FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Optimizing real-time physics simulation using GPGPU techniques

Ricardo José Santos Pereira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Jorge Manuel Gomes Barbosa

Co-Supervisor: Nelson Bilber Rodrigues

October 28, 2021

Optimizing real-time physics simulation using GPGPU techniques

Ricardo José Santos Pereira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Rosaldo Rossetti
External Examiner: Prof. André Pereira
Supervisor: Prof. Jorge Barbosa

October 28, 2021

Abstract

Modern-day game engines, primarily utilized in video game development, are seeing increasing adoption in other industries, due to their powerful capabilities for real-time computation and visualization of data. One such engine serves as the foundation for an application developed to simulate underwater environments and remotely operated vehicles used in subsea operations. The requirements for an interactive physics simulation, without loss of precision or stability, demand the authoring of custom logic which has a non-trivial impact on the application's performance. One possible avenue for optimization of this component of the application consists in exploiting the architecture of modern GPUs to massively parallelize the workload of the simulation.

This work presents a prototype engine with a rigid body physics simulator implemented both on the CPU and the GPU, with the latter using the SYCL framework, and tests the simulation in a variety of environments with different characteristics. The collision detection step shows satisfying results for simulations with a number of bodies in the order of thousands, while the collision resolution step shows promising results comparable with those on the CPU in situations with a high number of bodies, though some instabilities in simulation open up discussion for future research and improvements upon this work.

Keywords: real-time physics engine; rigid body simulation; parallel computing; general purpose computing on graphics processing units.

Resumo

Motores de jogo modernos, utilizados principalmente no desenvolvimento de videojogos, estão a ter crescente adoção em outras indústrias, devido à sua capacidade para computação em tempo real e visualização de dados. Um desses motores serve de base para uma aplicação que permite simular ambientes subaquáticos e veículos operados remotamente utilizados em operações submarinas. Os requisitos para uma simulação de física interativa, sem perda de precisão ou estabilidade, exigem a criação de lógica customizada que tem um impacto não trivial no desempenho da aplicação. Um possível caminho para otimização desta componente da aplicação consiste em tirar partido da arquitetura de GPUs modernas para paralelizar massivamente a carga de trabalho da simulação.

Este trabalho apresenta um protótipo de motor com um simulador de física de corpos rígidos implementado tanto no CPU como na GPU, este último utilizando a camada de abstração SYCL, e testa a simulação em vários ambientes com características distintas. A fase de deteção de colisões mostra resultados satisfatórios para simulações com um número de corpos na ordem dos milhares, ao passo que a fase de resolução de colisões mostra resultados promissores comparáveis aos do CPU em situações com um elevado número de corpos, embora algumas instabilidades na simulação abram a discussão para investigação e melhorias futuras sobre este trabalho.

Keywords: motor de física em tempo real; simulação de corpos rígidos; computação paralela; computação de propósito geral em unidades de processamento gráfico.

Acknowledgements

Firstly, I would like to thank both my supervisors: professor Jorge Barbosa, for the indispensable knowledge that he imparted relating to the field of parallel computing and graphics programming, and for his guidance on both the structuring of this thesis and on focusing on the elements of the work that were most important; and Nelson Rodrigues, a co-worker and friend, for his wisdom on how to properly read *and* write a scientific article, as well as to how to organize my work and, even more crucially, my time.

Next, I would like to thank my co-worker José Serra for his invaluable insights in physics simulation, which were instrumental in helping me achieve a working prototype.

I would also like to thank my co-workers and friends: Rui Pinto, Joana Pinho, Rui Figueirinha, and Matilde Freilão, for all of their patience and support during this venture; as well as two former co-workers, now friends, whom I deeply respect, and who guided me during my formative years and, in some capacity, contributed to this work: Cristiano Carvalheiro and Tiago Magalhães.

Additionally, I would like to thank the rest of the team at Abyssal for their enthusiasm, joviality, and for being a generally spectacular group of people.

Last, but certainly not least, I would like to thank some of my long-time friends: Nelson Lopes, Ricardo Carvalho, João Madureira, Leonardo Capozzi, and João Branco, for their fellowship and their uncanny ability to cheer me up during my most trying times by simply being them.

And of course, my parents and sister: José, Graça, and Bárbara, for everything, always.

RJSP

"All compounded things are subject to vanish.
Strive with earnestness!"

Siddhārtha Gautama Buddha

"Multithreading is just one damn thing
after,
before,
or simultaneous with another."

Andrei Alexandrescu

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem . . . 2
- 1.3 Objectives . . . 2
- 1.4 Implementation . . . 2
- 1.5 Organization . . . 3

2 Review of GPGPU and simulation concepts **5**
- 2.1 Fundamentals . . . 5
  - 2.1.1 GPU architecture . . . 5
  - 2.1.2 SYCL framework . . . 7
  - 2.1.3 Physics simulation . . . 9
- 2.2 Related work . . . 12
  - 2.2.1 Physics engines . . . 12
  - 2.2.2 Broad-phase sweep-and-sort . . . 12
  - 2.2.3 Broad-phase uniform grid . . . 13
  - 2.2.4 Narrow-phase separating axis test . . . 14
  - 2.2.5 Collision resolution mass splitting . . . 15
  - 2.2.6 Collision resolution constraint batching . . . 16
- 2.3 Summary . . . 16

3 Implementation of the physics simulation **19**
- 3.1 Engine structure . . . 19
- 3.2 GPU simulation details . . . 21
- 3.3 Collision detection . . . 23
  - 3.3.1 Broad-phase . . . 24
  - 3.3.2 Narrow-phase . . . 28
- 3.4 Collision resolution . . . 30
- 3.5 Summary . . . 36

4 Results **37**
- 4.1 Test environment . . . 37
- 4.2 Performance measurements . . . 42
  - 4.2.1 Random drifting environment . . . 43
  - 4.2.2 Layers falling environment . . . 47
  - 4.2.3 Spread bodies environment . . . 52
  - 4.2.4 Pyramid stack environment . . . 57
- 4.3 Discussion . . . 63

4.4 Summary 65

5 Conclusion **67**
5.1 Further work 68

A Compound types description **69**

B Glossary of mathematical symbols **71**

References **73**

List of Figures

2.1 Modern GPU architecture. Each pair of GPR/ALU designates an individual processing unit: the prefixes *s* and *v* distinguish between scalar and vector units, respectively. Adapted from [2]. 6
2.2 GPU work distribution. Common subdivisions of a range for a particular kernel invocation are shown. Adapted from [1]. 8
2.3 Parallel spatial partitioning. The first image shows cell indexing, while the second image shows cell attribution. Adapted from [18]. 14
2.4 Hyperplane separation theorem. The theorem states that if two convex sets are disjoint, then there exists at least one hyperplane between them. 15

3.1 Game engine loop. The game logic and simulation step execution flow is simplified for clarity. 20
3.2 Data layout alternatives. The figure details access to the components of a vector using one thread per body. The colors associate each statement to the corresponding access. 22
3.3 Data flow between CPU and GPU in SYCL simulator. Data labeled as constant may still be updated, for instance, when a body is added or removed from the simulation. 23
3.4 Projection of the extrema of two AABBs. The intersection of the blue and orange boxes can be seen as the smaller yellow box. 25
3.5 Bitonic merge sort execution example for 8 elements. The direction of the arrows indicates whether the order for those elements should be ascending or descending. 26
3.6 Partition sort example for 16 elements with work-group size of 4 elements. . . . 27
3.7 Face/vertex test example in two dimensions. The code sample demonstrates how to obtain a support vertex given a direction vector. 29
3.8 Example partitioning in 3 dimensions of 36 constraints with a maximum work-group size of 4. The thicker lines represent the division in grid cells, the thinner lines represent the division in cell groups. Parameters: $s_{gc} = 3$; $n_{wg} = 27$; $n_{wi} = 1$. 34
3.9 Algorithm for the dynamics computation. The GPU and the CPU version are similar in structure. 35
3.10 Algorithm for constraint solving on the GPU. The `precompute_constraints` method is similar in structure to this method. 35

4.1 Performance of broad-phase collision detection step in CPU with 100 bodies. . . 38
4.2 Performance of broad-phase collision detection step in CPU with 1000 bodies. . 38
4.3 Performance of broad-phase collision detection step in CPU with 10000 bodies. 39
4.4 Performance of narrow-phase collision detection step in CPU with 100 bodies. . 39
4.5 Performance of narrow-phase collision detection step in CPU with 1000 bodies. 40

4.6 Performance of narrow-phase collision detection step in CPU with 10000 bodies. 40
4.7 Performance of collision resolution step in CPU with 100 bodies. 41
4.8 Performance of collision resolution step in CPU with 1000 bodies. 41
4.9 Performance of collision resolution step in CPU with 10000 bodies. 41
4.10 Random drifting test environment. 43
4.11 Random drifting test broad-phase results with 100 bodies. 43
4.12 Random drifting test broad-phase results with 1000 bodies. 44
4.13 Random drifting test broad-phase results with 10000 bodies. 44
4.14 Random drifting test narrow-phase results with 100 bodies. 45
4.15 Random drifting test narrow-phase results with 1000 bodies. 45
4.16 Random drifting test narrow-phase results with 10000 bodies. 45
4.17 Random drifting test collision resolution results with 100 bodies. 46
4.18 Random drifting test collision resolution results with 1000 bodies. 46
4.19 Random drifting test collision resolution results with 10000 bodies. 47
4.20 Layers falling test environment. 47
4.21 Layers falling test broad-phase results with 100 bodies. 48
4.22 Layers falling test broad-phase results with 1000 bodies. 48
4.23 Layers falling test broad-phase results with 10000 bodies. 49
4.24 Layers falling test narrow-phase results with 100 bodies. 49
4.25 Layers falling test narrow-phase results with 1000 bodies. 50
4.26 Layers falling test narrow-phase results with 10000 bodies. 50
4.27 Layers falling test collision resolution results with 100 bodies. 51
4.28 Layers falling test collision resolution results with 1000 bodies. 51
4.29 Layers falling test collision resolution results with 10000 bodies. 51
4.30 Spread bodies test environment. 52
4.31 Spread bodies test broad-phase results with 100 bodies. 53
4.32 Spread bodies test broad-phase results with 1000 bodies. 53
4.33 Spread bodies test broad-phase results with 10000 bodies. 53
4.34 Spread bodies test narrow-phase results with 100 bodies. 54
4.35 Spread bodies test narrow-phase results with 1000 bodies. 54
4.36 Spread bodies test narrow-phase results with 10000 bodies. 55
4.37 Spread bodies test collision resolution results with 100 bodies. 55
4.38 Spread bodies test collision resolution results with 1000 bodies. 56
4.39 Spread bodies test collision resolution results with 10000 bodies. 56
4.40 Pyramid stack test environment. 57
4.41 Pyramid stack test broad-phase results with 55 bodies (5 layers). 58
4.42 Pyramid stack test broad-phase results with 385 bodies (10 layers). 58
4.43 Pyramid stack test broad-phase results with 1240 bodies (15 layers). 58
4.44 Pyramid stack test narrow-phase results with 55 bodies (5 layers). 59
4.45 Pyramid stack test narrow-phase results with 385 bodies (10 layers). 59
4.46 Pyramid stack test narrow-phase results with 1240 bodies (15 layers). 60
4.47 Pyramid stack test collision resolution results with 55 bodies (5 layers). 60
4.48 Pyramid stack test collision resolution results with 385 bodies (10 layers). 61
4.49 Pyramid stack test collision resolution results with 1240 bodies (15 layers). 61

List of Tables

4.1 Specifications for hardware used for the tests. 37
4.2 Results comparison for random drifting environment. 62
4.3 Results comparison for layers falling environment. 62
4.4 Results comparison for spread bodies environment. 63
4.5 Results comparison for pyramid stack environment. 63

Abbreviations

AABB	Axis Aligned Bounding Box
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FPU	Floating-Point Unit
GDS	Global Data Share
GPGPU	General-Purpose computing on Graphics Processing Unit
GPR	General-Purpose Register
GPU	Graphics Processing Unit
LDS	Local Data Share
PPU	Physics Processing Unit
RAM	Random Access Memory

Chapter 1

Introduction

1.1 Context

Modern game engines, mainly used for video game development, have been gaining traction across several other industries, such as architecture, movies and television, automotive manufacturing, as well as simulation, primarily due to their powerful capabilities for the representation of complex environments in real-time. Abyssal S.A. is a company in the subsea sector with several products that make use of one such engine for underwater structures and survey data visualization, as well as for the simulation of planned operations, the latter of which is the focus of this work. The Abyssal simulator project uses Unreal Engine, which integrates the PhysX engine, as a way to simulate realistic and physically accurate subsea environments, to assist the planning of operations, as well as the training of pilots of Remotely Operated Vehicles (ROVs), without the risks and costs associated with undertaking these operations in real life.

The area of computational physics employs numerical methods to obtain theoretical results for practical problems through successively more accurate approximations. Implementations of the methods used to create physics simulations, commonly designated *physics engines*, can be separated into *high-precision engines*, which favor precision of simulation over speed, and *real-time engines* that trade accuracy for better performance by simply providing a realistic approximation of the movement and interactions of the objects in some scene.

Dynamical simulation is a branch of computational physics that focuses on simulating systems of bodies, typically in three-dimensional space, according to Newton's laws of dynamics, through the use of time integration methods to model their behavior. This type of simulation is extensively used in computer animation to produce realistic-looking motion, as well as in video game development to obtain real-life-based physical behavior and interaction. The details of the simulation for each body depend on its characteristics, which are based on the requirements of the model, and their behavior can be described in terms of either solid or fluid dynamics. The scope of this work encompasses only the former, with a focus on rigid-body dynamics in the context of real-time engines.

In the real world, all solid bodies deform when in the presence of forces, but in many cases,

these are practically unnoticeable and do not significantly affect the bodies' motion. As such, they can be modeled as rigid bodies that retain their shape, and can only change their location and orientation relative to some referential (or each other), thus simplifying the implementation. For situations where the bodies must exhibit deformations when coming into contact with other objects, such as in the simulation of cloth, some different techniques must be employed, often with a higher computational cost, which is beyond the scope of this work.

Most physics simulations have some degree of parallelism that can be exploited since a non-trivial number of calculations can be made for each entity without requiring the synchronization of data between them. As such, this workload seems a prime candidate for GPU computation, as the architecture of these types of accelerators should capitalize on the parallelism of a simulation without requiring significant effort into modifying the logic of the implementation.

1.2 Problem

For the simulator to have realistic physics, there is a necessity for authoring custom logic for certain parts of the simulation. On the other hand, by forcing the engine to more accurately simulate some elements of the environment, some performance is naturally lost. As an attempt at improving the performance of the application, this work proposes researching whether the GPU can be effectively used for simulation, to reduce the load of the CPU, leaving it open to do other tasks that are independent of the simulation, and eventually increase the performance of the simulation by exploiting the parallelism of the architecture of GPUs. It is relevant to note that the current implementation of the PhysX engine available in Unreal Engine executes all simulation-related logic exclusively on the CPU.

1.3 Objectives

The main objective of this work is to establish the performance differences between rigid-body simulation on the CPU versus on the GPU, and determine whether the Abyssal simulator could benefit from moving the CPU simulation to the GPU, either partially or completely. If the GPU simulation displays significant performance improvements, while maintaining similar behavior to the CPU implementation, then this can be considered a viable potential avenue of development for the Abyssal simulator project. By establishing the CPU simulation as a baseline for performance, this work characterizes the algorithms involved, as well as the effort for their adaptation for execution on a many-core architecture, and by drawing a comparison between their performance across different environments and situations, provides reasoning for when such an adaptation is adequate.

1.4 Implementation

To achieve the proposed objectives, this work describes the implementation of a minimalist game engine prototype with modular subsystems responsible for the simulation of physics, with one

dedicated to the CPU simulation implementation, and another dedicated to the GPU simulation. At the same time, this work also has an exploratory component, as it will build the implementation of the GPU simulation upon a relatively young framework and experimental implementation thereof, named SYCL and ComputeCPP, respectively. This framework provides abstractions at the programming language level for heterogeneous computing, and this work employs it with the simultaneous goals of expanding on knowledge of the framework and overcoming some limitations of other well-established frameworks.

1.5 Organization

The remainder of this document is organized in four chapters: chapter 2 presents a review of some fundamental notions regarding GPU architectures and computing, as well as an overview of the SYCL framework and related concepts, followed by a high-level explanation of the main components that comprise a rigid-body physics simulation; chapter 3 describes the implementation of the prototype engine, starting by giving a brief description of the composition of the engine as well as its workings, and then going into detail regarding the simulation implementation, focusing in particular on the minutiae of GPU computing, and finally clarifying the algorithms used in each component of the simulation and their adaptation for GPU execution; chapter 4 demonstrates the results obtained from tests to the prototype implementation, initially characterizing the hardware used for the tests and the virtual environments where these were run, then laying out the results from each environment along with a brief description of these and any other behaviors observed during testing, and lastly elaborating on the results obtained; chapter 5 finishes by drawing some conclusions regarding the prototype and the results, framing them in the context that motivated this work, and finally providing a number of avenues of further work into this area of research.

Chapter 2

Review of GPGPU and simulation concepts

To contextualize the work presented here, several crucial concepts should be understood. Section 2.1 will explore some knowledge regarding modern GPU architectures (2.1.1) and introduces the SYCL framework and related concepts pertaining to GPGPU programming (2.1.2). Further on, a comprehensive overview of the components of a rigid-body simulator will be given (2.1.3), detailing some common procedures and how these interact, to provide the grounding necessary to understand how physics engines are integrated with real-time applications. Lastly, section 2.2 will discuss some modern physics engines with comparable capabilities (2.2.1) and present relevant literature that helps establish a basis for this work (2.2.2, 2.2.3, 2.2.4, 2.2.5, 2.2.6).

2.1 Fundamentals

2.1.1 GPU architecture

Graphics processing units feature a type of architecture that is, in some respects, the opposite of traditional processors. Being specialized for geometry manipulation and representation, the circuits are optimized to handle the processing of high amounts of data simultaneously while sacrificing speed in data access and divergence of execution flow between threads, therefore being better adapted to exploit data parallelism as opposed to task parallelism.

Figure 2.1 shows an example diagram of the architecture of a modern GPU. Typically, these feature a high number of cores, and each of these, while slower than a CPU core, include several integer and floating-point arithmetic units capable of processing large data blocks simultaneously, with floating-point units, in particular, being especially optimized as most graphical workloads chiefly require the processing of real numbers. On the other hand, there are few or no memory caches between the processor registers and local or global memory, making frequent data accesses slower [7] [12].

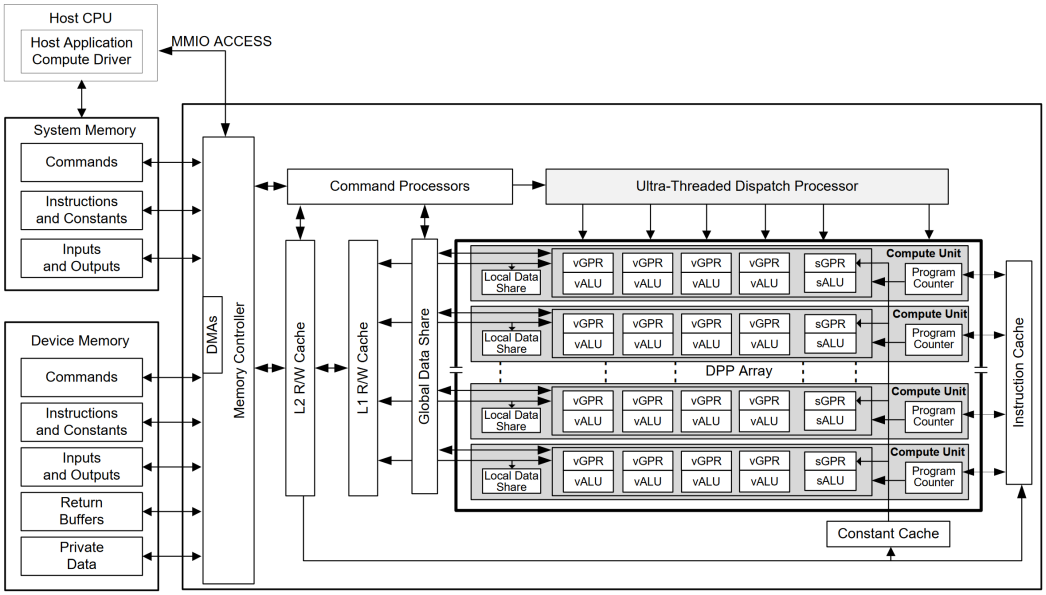

Figure 2.1: Modern GPU architecture. Each pair of GPR/ALU designates an individual processing unit: the prefixes *s* and *v* distinguish between scalar and vector units, respectively. Adapted from [2].

Going into more detail, the instructions of a GPU program, called *kernel*, are executed on a core by a collection of a fixed amount of threads (commonly either 32 or 64), designated a *warp* or *wavefront* (names given by Nvidia and AMD, respectively; this work adopts the former), usually in lock-step fashion, that is, with each instruction being executed by all threads simultaneously before the next instruction is processed. Most cores can execute a fixed number of warps concurrently, either running the same or different kernels. Additionally, most GPUs have specific hardware to efficiently handle the scheduling of these warps, and context switches between different kernels can happen at low or even zero cost. Also, by ensuring that the work on fetching and decoding instructions is done only once for each instruction in a warp, the hardware schedulers are able to somewhat reduce the latency of memory access for execution [2] [4].

Most GPGPU frameworks, however, abstract the execution of the warps, with kernels being programmed from the viewpoint of each thread, and simply allow specifying the total number of threads that should execute the kernel, leaving the attribution of the warps and the cores entirely to the GPU scheduler. It is still pertinent at times to carefully adjust the program to ensure that, for a particular invocation of a kernel, the number of threads is divisible by the warp size, in order to maximize throughput. Some frameworks allow for programming at the intra-warp level [1][4].

Branches or conditional statements present in a kernel are usually handled by keeping an *active mask* for each warp. Any thread for which an instruction should not be executed is marked in that warp as inactive; the thread will execute the instruction, but the result will be discarded. As such, handling divergence in a warp is usually an inefficient operation, with something such as an if-else statement resulting in code where both branches are executed sequentially: for the instructions

of each branch, the threads that would not have followed that branch are marked as inactive. It should be noted still that most schedulers are able to avoid spurious execution if all threads in a warp follow the same branch [2] [4].

Usually, a program that is designed to run some portion of the logic on a GPU has one or more kernels that are launched by first copying both the instructions and input data to the device memory, and then invoking the procedure. Copying data between host and devices has high latency, and as such, it should be done as infrequently as possible, otherwise, it will probably negate any performance improvements over a traditional CPU implementation. Therefore, such programs must be carefully structured to transfer as much data as possible at once, and hide latency by doing other work—and eventually submitting more work to the GPU—before accessing the results of any computation on a device [7].

Data that is transferred to a GPU will usually reside in the global data share (usually corresponds to the unit's dedicated RAM). Kernels can access any allocated data in this share, however, memory accesses are typically slower than on a CPU. To somewhat reduce this cost, each core on a GPU usually has a local data share which kernels may use as a faster memory pool for the duration of their execution. Additionally, data that does not change may be specified to be moved to a global constant cache, typically providing faster access than the global data share. Aside from these options, each warp also has a block of register memory available in each arithmetic unit which it can use for fast operation on frequently accessed data [1].

2.1.2 SYCL framework

The experimental work was done using a framework dedicated to heterogeneous computing named SYCL [1], which provides high-level abstractions for several types of hardware accelerators through some common interfaces. Following, this subsection will introduce some notions used by the framework, as well as clarify their relation and impact on some aspects of GPU devices that were discussed in the previous section.

The framework features a single source, multiple compilation passes design: the SYCL compiler compiles a source file and obtains the binary with the code intended to run on one or several devices, and a C++ compiler will compile the same source to produce code for the host, with the resulting binary files linked for the final executable file. During execution, the SYCL runtime ensures that the device code is invoked for the correct device at the correct points of the program [17] [7].

The SYCL programming model, built on top of the standard C++ language model, presents some general concepts that can be mapped to device characteristics in a way that is independent of hardware-specific terminology, composition, or behavior [17] [7].

Some of these are listed below:

- *Host* - general-purpose hardware, which may be connected to any number of *devices*, capable of running a SYCL application;

[1] SYCL, version 1.2.1, revision 7.

- *Device* - accelerator hardware, specialized for some specific type(s) of computation, consisting of one or several *compute units*;
- *Compute Unit* - collection of one or more *processing elements*, *e.g.*, a core;
- *Processing Element* - digital circuit capable of executing program instructions as arithmetic or logic operations, *e.g.*, an ALU or an FPU;

The kernels that are compiled to device code are launched on specific points of the application, but this launch is handled by the SYCL runtime, being abstracted away by a group of operations that couples the kernel definition with the data on which it operates, which it designates a *command group*[17].

The *range* on which a kernel operates is defined as the number of work-items that will execute said kernel, which may be further organized in terms of work-groups, which are equally-sized collections of work-items. This range may be specified in each command group as either just the number of work-items or the number of work-items and the size of the work-groups, with the requirement that the former is divisible by the latter. Considering the architecture of a GPU, a work-item usually maps to a single thread of execution in a warp, whereas a work-group may map to several warps on a core. Figure 2.2 shows an example of this hierarchy for a GPU device. It is also of importance to note that SYCL has no control over the scheduling of the warps for the execution of a particular kernel and that these may be scheduled to execute sequentially or concurrently—or a mix of both—with each other, depending on the capabilities and occupancy of the device, with this choice being controlled by the hardware. A particular kernel invocation may schedule warps across several cores [17][1] [7].

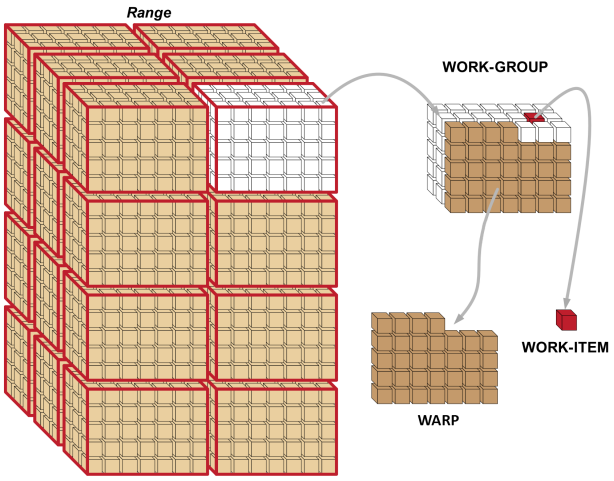

Figure 2.2: GPU work distribution. Common subdivisions of a range for a particular kernel invocation are shown. Adapted from [1].

The main advantage of organizing the range in work-groups is the possibility of synchronizing all work-items within individual work-groups on specific points. The way to achieve this is by

issuing barrier statements, which force work-items to await execution of all others in that work-group up to that point before continuing execution. This effectively allows thread communication within a kernel and thus accommodates for a larger variety of algorithms. However, depending on the size of the work-groups, and the number of work-items, some warps may contain threads that are inactive for the entire execution of a kernel, since their size is immutable. Excluding barriers, the only way to synchronize work-items is through the implicit synchronization that happens between kernel invocations. This means that to synchronize some datasets, some kernels must be programmed in a way that relies on multiple invocations of that kernel to synchronize changes to the dataset. Unfortunately, this is also significantly more computationally expensive than work-group synchronization [17].

Memory allocation is handled through *buffers*, either by wrapping existing data in a buffer or by creating a buffer with a specified amount of memory. A buffer does not define a location where the data will reside, but simply marks ownership of some memory, allowing the SYCL runtime to allocate memory and synchronize data wherever necessary between the host and any devices, depending on where the data is accessed. If the buffer creation wraps existing data from the host, the original memory location on which it resides is bound to the buffer for the duration of the lifetime of the buffer. On the destruction of the buffer, any changes to the data will be implicitly propagated so that the original data reflects said changes [17] [7].

To access data from the buffers, SYCL provides *accessors* which allow specifying where the data will be stored and accessed from, that is, the host or a device (in the latter case, they allow to further specify where in that device the data should be stored, *e.g.*, the global buffer, the constant buffer, the texture buffer, *etc.*), as well as the mode of access, *i.e.*, read-only, write-only, read-write, uninitialized-write-only, or uninitialized-read-write. In most cases, accessors are created from buffers, tying the allocation of the memory with the location of access to the data. If an accessor is created within a command group, it is associated with the devices on which it is submitted by the runtime, allowing it to keep track of the use of the data across different devices. However, to allocate memory on the local data share of a device, accessors are created as standalone objects within a command group and will manage access to that memory for the duration of the execution of the associated kernel. Internally, the runtime manages a graph of dependencies between kernels, characterized by the identity of the data and the mode of access, which it uses to determine the sequence of invocation of the kernels, as well as which kernels may be launched concurrently [17] [7].

2.1.3 Physics simulation

To aid the understanding of some of the concepts that will be discussed in further chapters, a treatment of rigid-body simulations will follow. Further mentions to bodies for the remainder of this work, when in the context of physics simulation, refer implicitly to rigid bodies. Since the representation of physical phenomena through graphics requires splicing time into discrete points, and the computation of these phenomena is reliant on numerical methods for the integration of forces and momenta of the bodies, each frame requires some parameter Δt which is the portion of time

that elapsed since the last calculated state of the simulation. Within simulations of rigid-body dynamics, there are three components at the core of almost any implementation: collision detection, collision resolution, and dynamics computation. These are necessarily executed sequentially, as the output of one serves as the input of the next. Different alternatives exist for how each of these components can be implemented, some of which will be examined further on.

The collision detection step analyzes the state of the world at a given moment to determine if the geometric shape of each body intersects any other body. Several different algorithms exist that fulfill this step, and these are separated into one of two types: *continuous* and *discrete* methods, also referred to as *a priori* and *a posteriori*, respectively. Continuous collision detection methods estimate the trajectories of the bodies and, relying on iterative methods and rolling back the simulation as necessary, attempt to determine the moment of collision as precisely as possible. These methods rely on extensive verification of all physical variables that might affect the motion of the bodies, as well as their contact. Discrete collision detection methods, on the other hand, analyze the position and geometry of pairs of bodies at each step to determine if they intersect, usually producing a list of pairs that are intersecting.

Continuous methods have the advantage of being more precise and stable, although that often comes at increased complexity and computational cost. Discrete methods are faster and do not typically need to be aware of the various physical variables that continuous methods require, but have the problem of not being physically correct, as they require an unrealistic situation to happen, which is that of the interpenetration of bodies, to detect the collision. Additionally, they have the disadvantage that if the Δt parameter is too large, or the velocities of the bodies are too high, or the geometry of the bodies involved is too small, situations may arise where two bodies pass through each other without the collision ever being detected. Despite these flaws, discrete methods are the most commonly used methods in game development, as they can still produce realistic-looking interaction, at the sacrifice of exactness, and due to their simpler implementation, they are the type of methods this work will focus on.

Regarding discrete collision detection, using the naive approach of checking each body against every other body, the number of checks necessary grows quadratically[2] with the number of bodies in the simulation, so several methods have been developed for reducing the complexity of the problem. Furthermore, most solutions involve separating this step into *broad-phase* and *narrow-phase* collision detection, executed sequentially in this order, to speed up computation.

The broad-phase utilizes some simple representation of the portion of space that a body occupies to quickly determine which pairs of bodies may be intersecting. Intuitively, if, at a given moment, the distance between two bodies is much larger than their dimensions, then these bodies cannot be intersecting. The characteristics of this representation usually allow for certain optimizations that reduce computational complexity and are much faster than the geometric checks required to detect interpenetration.

The narrow-phase works on the resulting pairs of bodies from the broad phase, and applies the necessary geometric checks to determine which of these pairs are effectively intersecting,

[2]Specifically, the number of checks needed for n bodies is given by $\frac{1}{2}(n^2 - n)$.

generating contact information for a pair if the bodies are determined to be intersecting, such as penetration depth (usually along the direction of minimal penetration), the contact normal, as well as a list of contact points. The next chapter will explore specific algorithms for each of these phases.

Following, the collision resolution step works on the result of the collision detection step, which is typically a list of pairs of intersecting bodies, together with information regarding the intersection and the Δt parameter, and calculates the changes in the physical properties of each body that would result from their contact, namely, velocity/momentum (simulations usually store only one of these quantities explicitly), position, and orientation. Again, different methods exist for how to compute this step, and this work will detail a specific one that uses an impulse-based contact model, commonly known as *sequential impulses*. In the sequential impulses method, collisions are modeled through *constraints* on the positions and velocities of the bodies, each one describing the pair of bodies in relation to one another, and defined in a way that does not allow interpenetration of the bodies.

Impulses are calculated as the effect of instantaneous forces F_r acting on a body, where F_r is the force resulting from the sum of external forces F_e and contact forces F_c. Thus, impulses Δp_r can be obtained for a frame at instant t through the following equation:

$$\begin{aligned} \Delta p_r &= \int_t^{t+\Delta t} F_r dt \\ \Delta p_r &= F_r \Delta t \end{aligned} \tag{2.1}$$

In each frame, the Δt parameter is small enough that, for the computation of the impulses, the approximation using the product between the forces and time yields effects that can be considered instantaneous for a real-time simulation. However, contact impulses are calculated by solving an inequality that enforces the constraints mentioned before, the details of which will be described in the next chapter. Additionally, effects such as bounce and friction are also modeled as a component of these constraints.

Lastly, the dynamics computation step computes the changes in velocities and momenta for each body resulting from the integration of applied forces at each frame with respect to the time elapsed since the last frame. Hence, considering the state of a body comprising its velocity v and position p at any moment t, then at each frame, the integration step to obtain the updated state can be described by the equation:

$$\begin{cases} v'(t) = v(t) + M^{-1} F_r \Delta t \\ p'(t) = p(t) + v'(t) \Delta t \end{cases} \tag{2.2}$$

Note that equation 2.2 is a particular method of numerical integration known as symplectic Euler (also known as semi-implicit Euler), where the updated position is dependent on the updated velocity of that frame [9]. Although other alternatives exist, this is the adopted approach for this work, and the next chapter will elaborate on this component further based on this method. It is

also worth noting that this step is commonly merged with the collision resolution step—and this work follows suit—since calculating the impulses that enforce the constraints requires that the velocities are updated first, as a result of the effects of the applied forces [9].

2.2 Related work

Following, some efforts in the application of heterogeneous computing models to existing physics engines will be covered. Afterward, relevant articles and research into the adaptation of the different components of a simulation for execution in graphics processing units will be reviewed. Further examination and clarification of some algorithms will be made in the next chapter.

2.2.1 Physics engines

Bullet[3] is an open source physics real-time engine with capabilities for rigid-body simulation and experimental support for execution of the entire physics step on GPU using OpenCL. Its implementation uses some of the algorithms that further chapters will analyze, and is in several ways similar to the prototype engine developed. According to the maintainers of this engine, performance between CPU and GPU counterparts is comparable if using a high-end GPU for simulating a large number of bodies, although they prescribe a case-by-case analysis to determine which version may be better suited for each use case. With this work, more concrete insights will be postulated to determine in what situations each component of a simulation, using a particular algorithm, may have a performance improvement when executed on a GPU [11] [10].

PhysX[4] is another open-source physics real-time engine, maintained by Nvidia, capable of several types of simulation, including rigid-body. Initially, PhysX was meant to be used as an API for interaction with a PPU, an accelerator specially built for physics computation, in many ways similar to a GPU. After limited adoption, it was rebuilt as a physics library with the capability for the execution of different tasks in CPU and Nvidia GPUs using CUDA capabilities. Despite having the ability to execute the entire rigid-body simulation in GPU, using heavily tested and validated methods, its restriction to Nvidia graphics cards limits developers who want portability across different systems [3].

2.2.2 Broad-phase sweep-and-sort

Liu *et al.*, 2010 [19] describe a collision detection algorithm for parallel broad-phase culling of rigid bodies on the GPU in real-time. They present an original hybrid sweep-and-sort (also known as sweep-and-prune) and spatial subdivision parallel algorithm that exploits spatial coherence between the bodies' positions within the time steps of a simulation.

The classic sweep-and-sort algorithm works by adopting an AABB for each body, o_i, projecting the extent of each box onto the x, y and z axes, and creating sorted lists of the intervals

[3] https://pybullet.org/wordpress/
[4] https://developer.nvidia.com/physx-sdk

$[min_{o_i}, max_{o_i}]$ formed by those projections for each axis. Each list is then "swept", with pairs of bodies that overlap being created if, for all lists, the intervals between the bodies overlap. The ordering of the lists is efficiently maintained across time steps by applying any sorting algorithm that is good for almost-sorted lists, taking advantage of the fact that bodies, and thus their bounding boxes, likely do not move significantly between consecutive steps. The parallel version is achieved by replacing the lists of intervals with lists of only the beginning of the interval of each body, min_{o_i}, and dividing the work of sweeping the lists to a body per thread, with each thread checking whether its body o_i intersects other bodies $o_j, j \neq i$ within its interval, checking the list and creating the pairs in the range $[min_{o_i}, min_{o_j}[$, where $min_{o_j} >= max_{o_i}$. Note that no duplicated pairs are created since each thread only checks against the minimums of the intervals of the bodies. The article then further describes the spatial subdivision algorithm, as well as the exploitation of motion coherence of the bodies for updating the list of pairs between time steps, and usage of principal component analysis with the position of each body to determine the optimal direction on which the sweep algorithm should be performed, projecting the box onto that axis instead of each of the x, y, and z axes. This review will not explore these components, as they are not as relevant to the prototype engine, although this work adopts a similar idea to the last one by using a single axis onto which the extrema of the boxes are projected.

This article describes several novel approaches to collision culling through adaptation of established sequential algorithms and goes into great detail about how each method in their solution is implemented. Furthermore, they display promising runtime results for tens of thousands of bodies in a scene, although the authors refer to certain steps of the algorithm as redundant for scenes with an amount of bodies below specific thresholds, as no performance gain is obtained from those steps in those situations.

2.2.3 Broad-phase uniform grid

Nguyen *et al.*, 2007 [20] dedicate a whole section on their book to physics simulation on the GPU, presenting algorithms for all phases of rigid-body simulation across several chapters. This book contains several useful examples of parallelization of traditional algorithms used in simulation, with an emphasis on real-time solutions, which are of particular interest to this work.

In particular, Le Grand, 2007 [18] describes a spatial partitioning algorithm using a uniform grid to process body intersections, showing how a parallel implementation for GPU may be achieved from the conventional sequential version. The traditional algorithm divides the world in a uniform grid, such that each cell is at least as large as the bounding volume of the largest body.

For the parallel version, the cells are scaled in such a way that any body assigned to a cell can interact with bodies in that same cell and at most 2^d adjacent cells[5], where d is the number of dimensions. This work designates a *cell group* to be the set of cells with the same index, with a total of 2^d cell groups for a split in d dimensions. To ensure that any potential pair is found only once in the parallel version, the processing of the cells is done in 2^d sequential passes, with each

[5] Two cells are considered adjacent if they share a face, an edge, or a vertex.

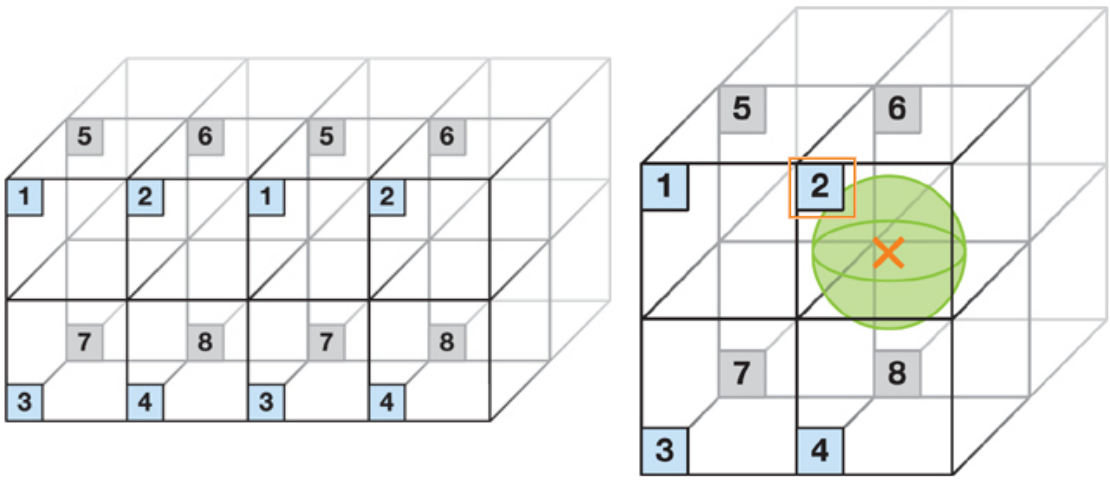

Figure 2.3: Parallel spatial partitioning. The first image shows cell indexing, while the second image shows cell attribution. Adapted from [18].

cell being assigned an index such that no two adjacent cells have the same index, as seen in figure 2.3. The article goes on further describing the rest of the algorithm, but it will be omitted here for brevity.

These ideas, although built for the broad-phase collision detection, turn out to also be useful for the collision resolution step, to determine the order in which to process the body pairs that collide, as an alternative to other methods. As it will be shown in subsection 2.2.6, some research by Harada, 2011 [14] into the problem of parallelization of collision resolution uses a technique that is similar in principle to this one, and this work adopts a mixture of the two for the prototype implementation.

2.2.4 Narrow-phase separating axis test

Gregorius, 2015 [13] describes a narrow-phase collision detection algorithm that can easily be adapted for GPU execution, which is based on the hyperplane separation theorem for convex polytopes as summarized by figure 2.4.

More specifically, the algorithm allows for pairs of arbitrary convex meshes to be tested with $O(f_1v_2 + f_2v_1 + e_1e_2)$ time complexity, where f_i, v_i, and e_i are respectively the number of faces, vertices, and edges of the body i, and $O(1)$ auxiliary space complexity. The algorithm requires no changes to the bodies it tests, and as such, it is embarrassingly parallel, as all potential collisions can be tested simultaneously. Additionally, when interpenetration is detected between two bodies, the algorithm is capable of generating contact information such as which faces and/or edges of each body are penetrating, used for generating contact points afterward, as well as the direction of minimum penetration, used as the contact normal, and the penetration depth.

Besides the collision test, the article further provides an overview of algorithms for contact generation and contact reduction, which use the information produced by the collision detection

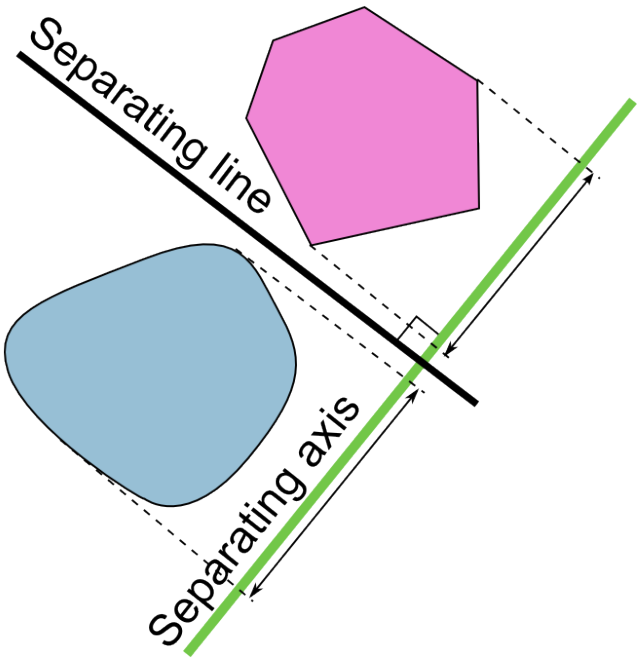

Figure 2.4: Hyperplane separation theorem. The theorem states that if two convex sets are disjoint, then there exists at least one hyperplane between them.

algorithm to create a list of contact points for the collision, later necessary for the collision resolution step.

2.2.5 Collision resolution mass splitting

Tonge *et al.*, 2012 [22] describe a novel approach to contact resolution by splitting the mass of bodies with multiple contacts before running the solver, which enables usage of a sequential algorithm with faster convergence to the solution, after which the results are joined using a parallel reduce algorithm to sum the resulting velocities before integrating. Their main goal is to have minimal jitter in situations with a high number of contacts while maintaining the real-time constraints of the simulation. The division of constraints for parallel processing is done using a contact painting algorithm which they do not describe in detail, but provide references for [15]. Of special interest is the fact that this method still models the collisions as a linear complementarity problem, but achieves high stability and precision, even with a very high number of contacts, in real-time. They additionally describe how to integrate a model of friction with their algorithm by alternating iterations between the solver for the non-penetration constraints and the solver for the friction constraints.

Despite the novel ideas for parallelization of the solver based on a mathematical approach, the implementation seems complex, even though it shows promising results, and this work foregoes this algorithm in favor of the one in the following subsection.

2.2.6 Collision resolution constraint batching

Harada, 2011 [14] describes a collision resolution implementation for constraint parallelization that runs entirely on the GPU, which it achieves by splitting the contact pairs into groups designated batches. All the constraints in a batch have distinct bodies among them, in other words, no two constraints in a batch contain the same body, and so a batch may be processed entirely in parallel.

Initially, a spatial partitioning of the constraints is made to separate them into disjoint groups, which are processed sequentially. This is where the uniform grid algorithm described by Nguyen *et al.*, 2007 [20] may be used, separating the bodies in cells, with cells in the same cell group being processed in parallel, and each cell group being processed sequentially. Afterward, the batch creation step is applied for each group of constraints, with each group being processed by a work-group, where each work-item operates on one constraint of that group. The algorithm attempts to attribute a batch to each constraint. At each iteration of the batching, each work-item attempts to mark both bodies that are colliding. If it succeeds, those bodies are not shared by other constraints, and thus that constraint is attributed to that batch and may be solved in parallel with other constraints in that batch. If it fails, it waits until the next iteration to try to mark the bodies and attribute a batch to the constraint. The number of iterations that are executed is chosen empirically, as there are performance trade-offs to consider, namely, a higher number of iterations will produce more batches, with potentially more constraints batched in that frame, at the cost of longer execution times for that frame. Lastly, when batch creation finishes, constraints are ordered with respect to their batches, so that there is better distribution of work across warps. The constraints are distributed, using a work-group for each cell, and solved in parallel for all the constraints within the same batch, but sequentially relative to the constraints in different batches within each work-group, and sequentially relative to different cell groups within all work-groups.

This paper presents a theoretically sound idea for the parallelization of arguably the most complicated and most computationally demanding part of a simulation. Some advantages of their approach, as described, include good data locality characteristics, due to the processing of constraint groups being local to each work-group through the usage of a spatial split algorithm (though it should be noted that this work's approach does not take advantage of this aspect), and the exploitation of the parallelism of the problem through the further separation of constraints in batches within each work-groups. As such, due to seemingly enabling a simulation to handle thousands of bodies, and apparent simplicity of implementation, this algorithm was adopted for this work.

2.3 Summary

This chapter presented an overview of GPU architecture and the SYCL framework, introducing some core concepts relevant for further discussion, namely, kernels as the procedures that execute on a GPU, the distribution of data in the GPU, the separation of the threads that execute a kernel

as several work-items distributed between one or more work-groups, and how these notions relate to accessing data and synchronizing changes to it. Afterward, a brief mention of established physics engines, referring to their capabilities and shortcomings in the field of GPU simulation, followed by a discussion of relevant articles into physics simulation on the GPU, listing essential algorithms, for each of the simulation phases, *i.e.*, broad-phase and narrow-phase collision detection and collision resolution, that are used in the prototype engine developed by this work, along with some other related research.

Chapter 3

Implementation of the physics simulation

The following chapter will delineate the prototype implementation, describing the structure of the engine in section 3.1 and some of the considerations that need to be taken when integrating the GPU physics simulation with the remaining computations on the CPU in section 3.2. Then, there will be an analysis of the particular algorithms used in the different phases of simulation, detailing the aspects of parallelizing the workload of a typical real-time simulation and adapting it for execution on a GPU device in sections 3.3 and 3.4.

3.1 Engine structure

Traditional game engines follow a common formula for structuring the application. Due to their necessity of constantly updating the screen with the latest render of the state of the world, the application is built in a way that certain events repeat in a loop, as represented in figure 3.1, where the execution of an iteration of that loop consists in a *frame* of the game engine.

The prototype engine developed for this work follows this structure, with different subsystems responsible for each part of the logic. To easily switch and compare between host and devices, two different subsystems were designed for the simulation step, one using native C++, the *native simulator*, and the other using the SYCL framework, the *SYCL simulator*, to represent and simulate the world in the CPU or the GPU, respectively. Each subsystem is controlled by a separate thread of execution, allowing the game logic and the simulation step to be executed concurrently. This feature is crucial for the SYCL simulator to take advantage of the massive parallelism found in GPU devices, which section 3.2 expands upon with more detail. During the engine initialization, a specific configuration is loaded from a file to determine which simulator subsystem it should use.

The implementation is structured such that there is a main engine class controlling the main game loop. The engine instances a singular world, which contains and manages entities. Every

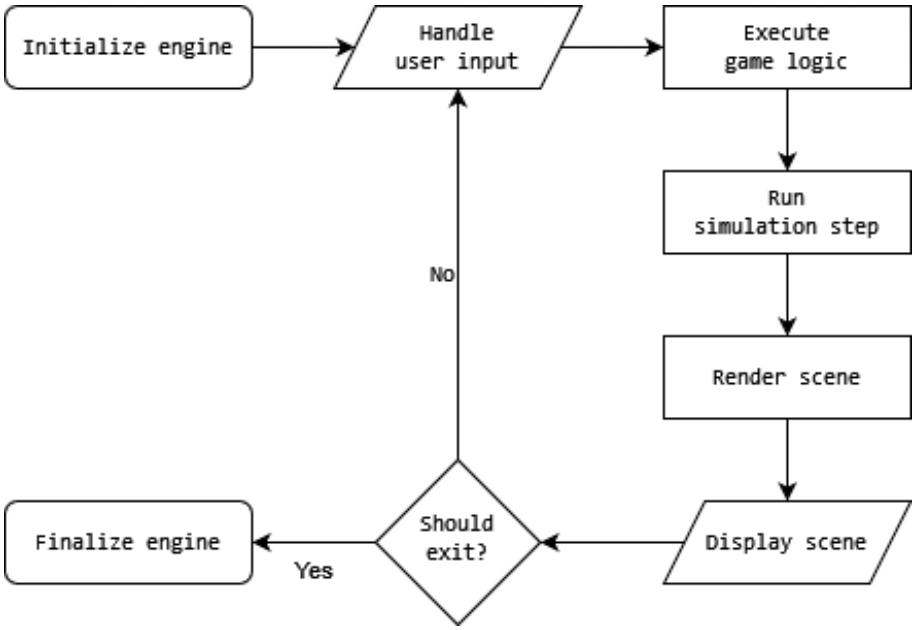

Figure 3.1: Game engine loop. The game logic and simulation step execution flow is simplified for clarity.

element that may exist in a world, including anything that comprises the environment, is an entity. Each entity contains both a physics body, responsible for the entity's simulation, and a graphics body, responsible for the entity's rendering, and may contain arbitrary logic that is executed at each frame, commonly called a *tick*, which is configured during the entity creation to run either before, during, or after the simulation step, or to not run at all. Both of the bodies in an entity are primarily containers for specific physics or graphics data and logic, which is used by the respective subsystems. During the entity initialization, specific logic may dictate whether these bodies are registered on the respective subsystems, as well as some of their characteristics for interaction and representation in the world. As such, whether an entity has or not any specific tick logic has no bearing on whether an entity is simulated and/or rendered.

Other features of the engine include subsystems for the handling of user input, such as keyboard and mouse events, handling operating system window events, setting and resetting timed events, loading and storing of engine configurations, logging of warnings and errors as well as frame and/or arbitrary data—crucial for profiling the performance of the engine, and detection and selection of devices (mainly for configuring the SYCL simulator). Besides these subsystems, the prototype uses its own set of custom compound types for describing mathematical constructs such as vectors, quaternions, matrices, *etc.*, which are fundamental elements for the representation of bodies within a three-dimensional environment. Appendix A contains details regarding the composition and nomenclature of some of these types. These are built in a way that allows them to be used in SYCL code without any additional concerns regarding data movement and synchronization.

As a way of having minimal impact on the frame duration, and thus in the results and stability

of the simulation, the engine renderer, responsible for drawing and displaying the world in each frame, uses a pipeline with minimal features to represent the bodies that exist on the world, with a simple lighting model built at the shader level, as well as mesh instancing, taking advantage of the fact that many or all of the bodies used in the tests have similar geometry.

3.2 GPU simulation details

To achieve a simulation on the GPU with a performance at least comparable to one on the CPU, some characteristics with regards to the architecture and capabilities of the GPU must be taken into consideration when building the simulator. In particular, there are some important aspects of GPGPU programming to consider, namely: data throughput, memory latency, and arithmetic intensity, all of which are intrinsically related. Ideally, to attain a competitive data throughput, that is, to process all input data and compute the updated state of each body that is being simulated within the time constraints of a frame, care must be taken to minimize the number of memory operations to the least necessary, thus reducing memory latency, and maximize the number of arithmetic operations per memory operation, increasing arithmetic intensity.

Memory buffers (in the global data share) should be accessed as rarely as possible to reduce memory latency, using private registers to store frequently accessed data. Whenever data must be synchronized between threads, the simulator should prefer algorithms where the data is divided among different work-groups, to allow using the local data share of each work-group to synchronize and access the data faster than using the global data share.

Due to the architecture of GPU devices, warps should ideally access data in continuous blocks, since instructions to access memory usually access arrays of several integral or floating-point numbers, as opposed to single numbers. This pattern of memory access is referred to as coalesced memory access. Non-coalesced memory accesses may force sequential accesses to memory from different threads in a warp, resulting in a source of inefficiency in the kernel. This particular optimization was not applied to the prototype, having been sacrificed for uniformity and simplicity of implementation, by reusing the compound types used by the native simulator. Figure 3.2 details the differences between an ideal data layout for coalesced memory accesses and the one adopted by the prototype subject to non-coalesced accesses, exemplifying access to an arbitrary vector quantity.

```
1 void apply_impulse(array<vector>& momentum, vector const& force, float
      delta_time)
2 {
3     unsigned thread_id = get_work_item_id();
4
5     momentum[thread_id].x += force.x * delta_time;
6     momentum[thread_id].y += force.y * delta_time;
7     momentum[thread_id].z += force.z * delta_time;
8 }
```

Non-coalesced memory access

Body 1			Body 2			Body 3			Body 4		
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z

Coalesced memory access

Body 1	Body 2	Body 3	Body 4	Body 1	Body 2	Body 3	Body 4	Body 1	Body 2	Body 3	Body 4
X	X	X	X	Y	Y	Y	Y	Z	Z	Z	Z

Figure 3.2: Data layout alternatives. The figure details access to the components of a vector using one thread per body. The colors associate each statement to the corresponding access.

Regarding other sources of latency, it is important to mention that there is a non-trivial overhead associated with the launching of kernels. Since kernel termination is the only way in SYCL of synchronizing data in the global data share across several cores, any implementation should prefer algorithms that require as little global synchronization of data as possible. Another source of latency comprises the transfer of data between CPU and GPU devices, and as such, one of the goals while implementing the simulator is to ensure that the least amount of data necessary is transferred at each frame. Thus, data that does not change or changes infrequently should ideally be allocated separately from data that changes often.

Since the prototype engine requires the results of a simulation step to be transferred back to the CPU to run any custom logic and update the graphics body of the entities, the simulator must organize the data in a way where work is constantly being submitted to the GPU, without waits for data transfers and kernel executions. During implementation, it was determined that the only way to achieve this is by allowing the simulation to run one frame ahead of the rest of the engine. In practice, this means that on each frame, the physics state of each entity on the CPU after the simulation step, and thus the state that is used to update the graphics body and rendered to the screen, corresponds to the previous frame of the simulation. Figure 3.3 shows the flow of data between CPU and GPU, with the world being rendered at the end of each frame with the data that is output from the simulation. This also implies that custom logic that changes according to the state of the physics body has an effective delay of at least one frame. The simulator was adapted to allow the simulation to run an arbitrary number of frames ahead of the engine.

The physics body utilized by the SYCL simulator differs from the native simulator one in that all data is maintained in several arrays managed by the simulator, instead of being managed by the body it belongs, with the body itself only allowing for simple indexing to the data that belongs

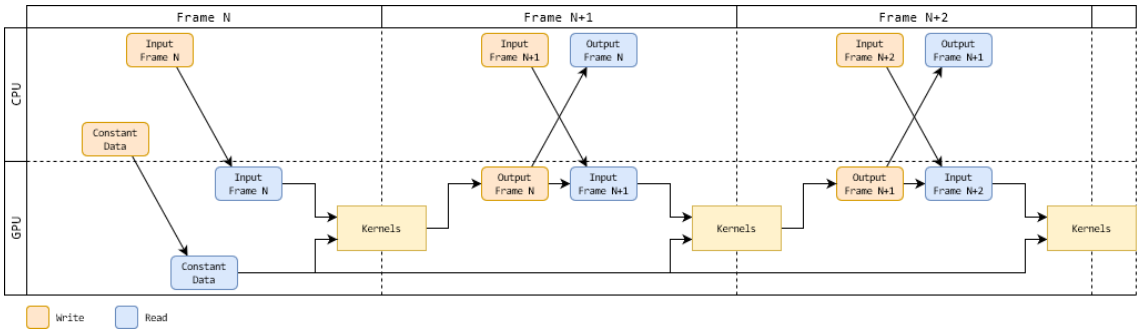

Figure 3.3: Data flow between CPU and GPU in SYCL simulator. Data labeled as constant may still be updated, for instance, when a body is added or removed from the simulation.

to the entity. This allows for faster logic to transfer this data between host and devices, as it can access the data in blocks instead of iterating through every body.

The simulator attributes one thread/work-item for each body or pair of bodies when handling collisions. Thus, the simulation is optimized for situations with high body counts, with the reasons for this choice being that these are the main types of situations that this work is attempting to optimize, and the high arithmetic intensity of the operations for each body in each frame can better offset the costs of device execution in low body counts simulations (when comparing to the costs of host execution).

One last note regarding the simulators is that both make use of a technique to improve stability known as substepping. This consists of the division of the frame Δt into n_{sub} smaller Δt_{sub} intervals, and the execution of the constraint solver n times using Δt_{sub} as the time parameter. The goal of this technique is to increase stability when the Δt parameter is relatively big, which happens when the frame rate of the application decreases, as they are intrinsically related. However, it is still dependent on the frame rate, as a higher n_{sub} implies a higher computation cost per frame, and as such, situations may occur where the cost of executing the solver n times exceeds that of the Δt that originated the substeps. It should also be noted that this implies the overhead of launching the collision resolution kernels n_{sub} times, as each substep must be executed sequentially relative to each other.

3.3 Collision detection

This section will describe the algorithms used by the simulators for the collision detection step of the simulation. While the algorithms are fundamentally the same and the implementations are similar between the simulators, this section will focus primarily on the details of the SYCL implementation, highlighting parts where it differs from the native implementation. The next subsections will separate the discussion between the broad-phase and the narrow-phase algorithms.

3.3.1 Broad-phase

The main algorithm adopted for broad-phase detection, responsible for quickly determining which bodies may be colliding and culling those that cannot be colliding from further checks, is the sweep-and-sort algorithm, briefly described in subsection 2.2.2. The algorithm utilizes an AABB for each body and projects the minimum and maximum of the box, represented by vectors, through the dot product onto an arbitrary axis, represented by a unitary vector. The idea behind the algorithm is that if two AABBs intersect, the projections of their extrema onto some axis with specific properties also intersect.

The following demonstration provides a reasoning for the algorithm, as well as for why the adaptation for parallel processing works. Appendix B contains a glossary of mathematical symbols used in the demonstration.

Using the following relations between two arbitrary points p and q:

$$p < q \Leftrightarrow p_x < q_x \wedge p_y < q_y \wedge p_z < q_z$$
$$p \leq q \Leftrightarrow p_x \leq q_x \wedge p_y \leq q_y \wedge p_z \leq q_z$$

Considering a box B as the set of points it contains, with minimum $min(B)$ and maximum $max(B)$, such that:

$$\begin{gathered} min(B) \in B, \forall c \in B(min(B) \leq c) \\ max(B) \in B, \forall c \in B(max(B) \geq c) \\ min(B) < max(B) \end{gathered} \tag{3.1}$$

Then for any two boxes P and Q, the following equivalence holds:

$$P \cap Q \neq \emptyset \leftrightarrow min(P) \leq max(Q) \wedge min(Q) \leq max(P) \tag{3.2}$$

Which is a necessary and sufficient test for detecting an intersection between two boxes.

Considering now two arbitrary points p and q such that $p \leq q$, and an axis a such that:

$$a_x \geq 0 \wedge a_y \geq 0 \wedge a_z \geq 0 \wedge a_x^2 + a_y^2 + a_z^2 = 1 \tag{3.3}$$

The following implication can be made:

$$p \leq q \rightarrow a \cdot p \leq a \cdot q \tag{3.4}$$

Then for any two boxes P and Q and axis a which conforms to the condition 3.3, starting from the expression 3.2 and applying the implication 3.4, using the notation $b_{min} = a \cdot min(B)$ and $b_{max} = a \cdot max(B)$ for an arbitrary box B, the following expression can be obtained:

$$min(P) \leq max(Q) \wedge min(Q) \leq max(P) \rightarrow p_{min} \leq q_{max} \wedge q_{min} \leq p_{max} \tag{3.5}$$

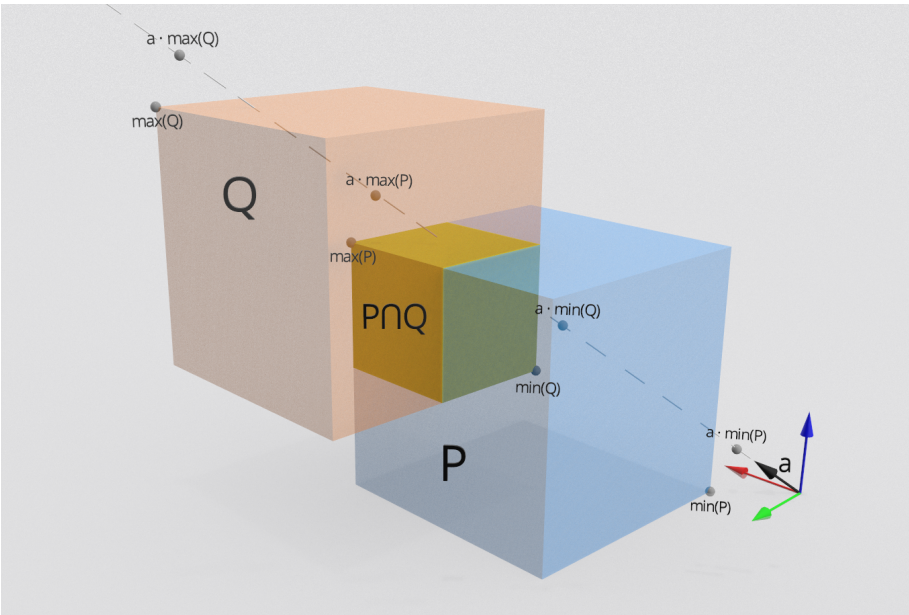

Figure 3.4: Projection of the extrema of two AABBs. The intersection of the blue and orange boxes can be seen as the smaller yellow box.

Expanding it further:

$$
\begin{aligned}
p_{min} \leq q_{max} \wedge q_{min} \leq p_{max} \Leftrightarrow \\
1 \wedge p_{min} \leq q_{max} \wedge q_{min} \leq p_{max} \Leftrightarrow \\
(p_{min} \leq q_{min} \veebar q_{min} < p_{min}) \wedge p_{min} \leq q_{max} \wedge q_{min} \leq p_{max} \Leftrightarrow \\
(p_{min} \leq q_{min} \wedge p_{min} \leq q_{max} \wedge q_{min} \leq p_{max}) \veebar (q_{min} < p_{min} \wedge p_{min} \leq q_{max} \wedge q_{min} \leq p_{max}) \Leftrightarrow \\
(p_{min} \leq q_{min} \wedge q_{min} \leq p_{max}) \veebar (q_{min} < p_{min} \wedge p_{min} \leq q_{max})
\end{aligned}
$$

$$
P \cap Q \neq \emptyset \rightarrow (p_{min} \leq q_{min} \wedge q_{min} \leq p_{max}) \veebar (q_{min} < p_{min} \wedge p_{min} \leq q_{max}) \tag{3.6}
$$

Noting that if $c \leq b_{min} \rightarrow c \leq b_{max}$, since $b_{min} < b_{max}$, then $c \leq b_{min} \wedge c \leq b_{max} \Leftrightarrow c \leq b_{min}$.

Thus, the expression 3.6 can be used to quickly determine whether two boxes cannot be colliding, or require a full test to determine whether they are colliding or not. By attributing a thread to each body, and keeping an ascending ordered list of the projection of the minima of the AABBs of all the bodies, the list can be swept by each thread starting from the minimum of their body p_{min}, testing each body up to the element where $p_{max} < q_{min}$. Since only those elements fulfill the condition 3.6, each thread tests only the bodies it needs to test using the expression in 3.2, and no thread creates repeated overlaps due to the exclusivity clause. Additionally, because bodies move relatively little between consecutive frames, the extrema of the corresponding AABBs, and thus their projection, also varies little, meaning that on any frame, the list from the last frame is often in an almost-sorted state, hence needing minimal updates to sort it.

In a purely sequential version, the ordered list traditionally keeps both the minimum and maximum of the boxes, though the condition for determining which boxes to test for intersection is similar. To sort the list on each frame, usually, a simple algorithm such as insertion sort is chosen, due to simplicity of implementation and better performance on almost-sorted lists, with $O(1)$ auxiliary space complexity and expected $O(n)$ time complexity [21] [6]. In parallel implementations, however, especially in a device such as a GPU, alternative algorithms must be used to take advantage of the architecture. Some alternatives include radix sort and bitonic sort, the latter of which is adopted by the prototype. This algorithm is based on a sorting network, an abstract device that is designed to work on a fixed number of values within a fixed number of operations. The implementation used by the prototype is known as bitonic merge sort, which sorts n elements in $O({log_2}^2(n))$ time. More specifically, for n elements, exactly $s = \frac{{log_2}^2(n)+log_2(n)}{2}$ steps are necessary to sort the list. Since at each step, a synchronization of the list of elements is necessary, in practice the algorithm is implemented through s kernel launches with a parameter indicating which step that kernel is executing. Due to the nature of the algorithm, n must be a power of two, so the implementation fills the list with arbitrarily giant values so that the first n elements corresponding to the actual number of bodies contains the sorted list of bodies [8].

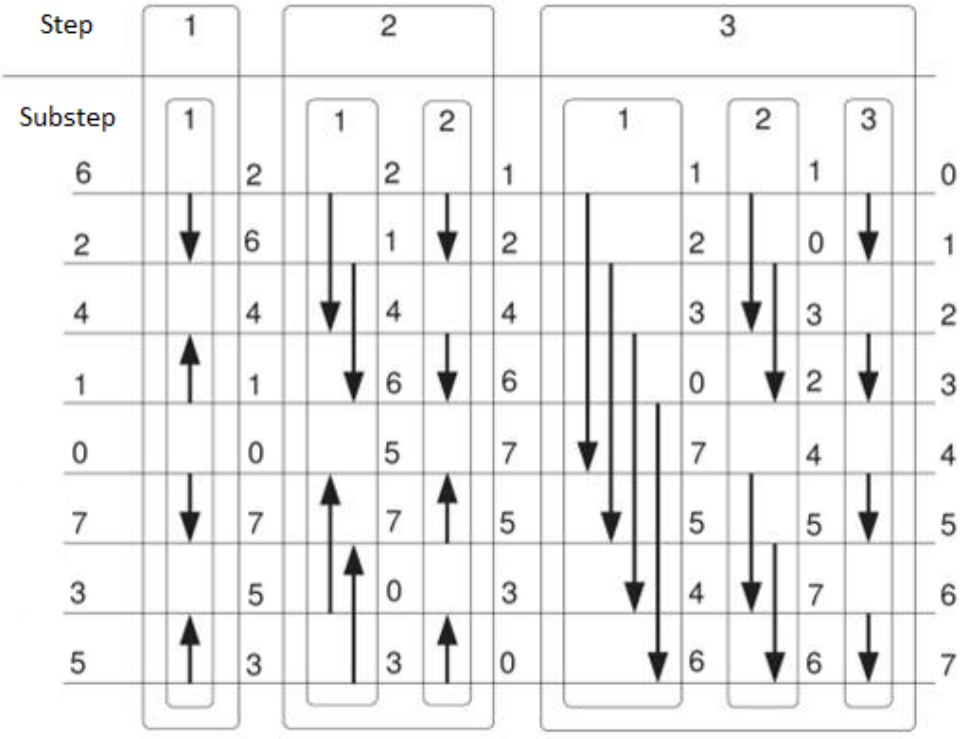

Figure 3.5: Bitonic merge sort execution example for 8 elements. The direction of the arrows indicates whether the order for those elements should be ascending or descending.

One problem with this algorithm is that for a very high number of bodies, a proportionally high number of kernel launches is necessary to sort the list, which incurs heavy performance penalties. As a way to solve this issue, the algorithm in the implementation was adapted so that the first frame

fully sorts the data and, and if no data is added to or removed from the simulation, subsequent frames divide the data of the previous frames into blocks with size less than or equal to the size of the work-group, launching kernels to sort each of those blocks of data by taking advantage of the almost-sorted nature of the data. The algorithm launches at most two kernels for these sorts of each block of data, offsetting the indexes where each work-group works on the second launch, so that each element can move at least $\frac{s_{wg}}{2}$ indexes, where s_{wg} is the size of the work-group, as seen in figure 3.6. Since threads within a work-group can synchronize data through the local data share using special barriers, these kernels avoid the need for multiple kernel calls, and thus the sort step becomes much more efficient. If at any point data is changed in the simulation, the simulator will simply discard the sort data, request a full sort for that frame, and request sorts for the blocks of data on subsequent frames.

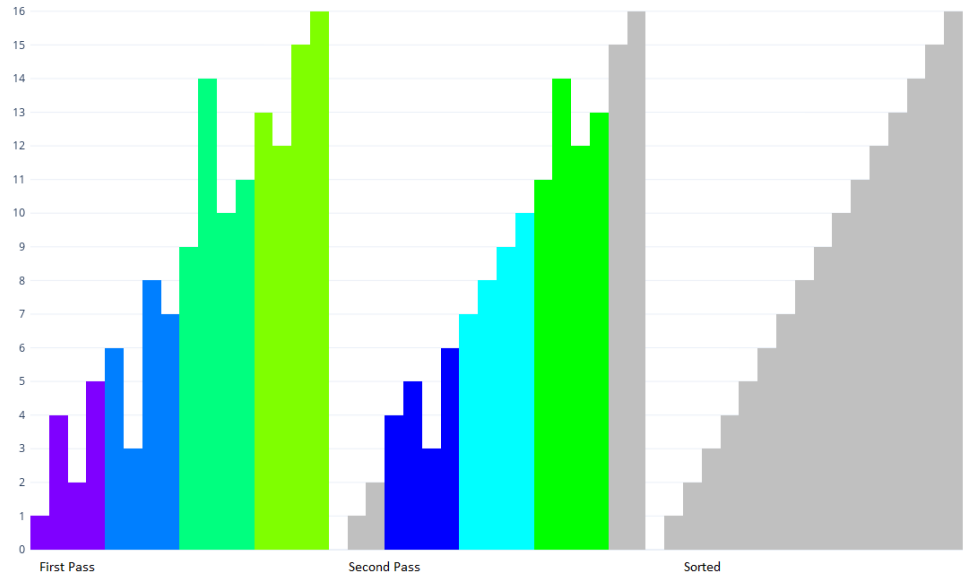

Figure 3.6: Partition sort example for 16 elements with work-group size of 4 elements.

In parallel with the sort-and-sweep algorithm, this step is also responsible for calculating the minimum and maximum of the extrema of the AABBs, as well as the maximum of the extents of the AABBs, of all the bodies. The simulator uses a parallel reduce algorithm to obtain these values, and this information is then used to partition the bodies using a uniform grid, as described in subsection 2.2.3, determining their cell index and cell group, to later use during the collision resolution step.[18]

The simulator allocates a list of overlaps with size $n \times c$, where n is the number of bodies and c is an estimate to the number of collisions, defaulting to some predefined value, with each body having c slots reserved for them. During the sweep of the sorted list, each thread tests the AABB of its corresponding body b_t with the AABBs of other bodies b_o, and if an overlap is found, the thread stores the index of the body b_o in the overlap list in the indexes reserved for its body, up to c overlaps. For each thread, any overlap beyond the c limit is not registered on the overlap list,

and hence is not considered for the remainder of the collision tests, but it contributes to a counter of the maximum number of overlaps per body c'. This c' is used in the next frame by the simulator to reallocate the overlap list if necessary (in particular, if $c < c'$), to accommodate that number of overlaps for that frame.

3.3.2 Narrow-phase

After obtaining the results from the broad-phase algorithm, the narrow-phase collision detection step is responsible for testing the geometry of each of the pairs of bodies. This step is similar between CPU and GPU, as it is embarrassingly parallel, and so this is where potentially more benefits can be gained from execution on the GPU, considering the massive difference in parallelism between the two types of processors.

Firstly, a thread is attributed to each body to update the respective geometry based on its position and orientation. Then, each thread analyses the list of indexes of overlaps for the corresponding body and tests the geometry of the body using the algorithm discussed in subsection 2.2.4. This algorithm features a high number of arithmetic operations, which increase with the complexity of the geometry, making it an even better candidate for execution in the GPU as opposed to the CPU. It is worth noting that this algorithm could theoretically display better performance if threads in the same warp are testing bodies with similar geometry, although no optimization was made in this regard for the prototype, with reasons for this choice being discussed in section 4.3.

The algorithm is based on the hyperplane separation theorem, which claims that if two convex sets are disjoint, then there exists at least one hyperplane between the sets. Hence, the algorithm tries to find such a hyperplane, by testing each of the components of the geometry of two bodies, and if it succeeds, then the bodies do not intersect. In practice, initially, the algorithm tests the faces of each body, comparing each face of one body to a support vertex of the other body. This support vertex is a vertex of the body obtained by a support function of the geometry, which gives the farthest vertex along a given direction, and the negation of the face normal of the other body is used as that direction for each test, as described in figure 3.7. If the signed distance of that vertex to the plane that contains the face is positive, then the plane that contains the face is a plane of separation between the bodies, and they are disjoint.

```
1  vector poly::support(vector const& dir)
2  {
3      vector farthest_vert;
4      float farthest_proj = -FLT_MAX;
5
6      for (unsigned int idx = 0; idx <
       vert_count; ++idx)
7      {
8          vector const& vert = vert_arr[idx];
9          float proj = vector::dot(dir, vert);
10
11         if (proj > farthest_proj)
12         {
13             farthest_vert = vert;
14             farthest_proj = proj;
15         }
16     }
17
18     return farthest_vert;
19 }
```

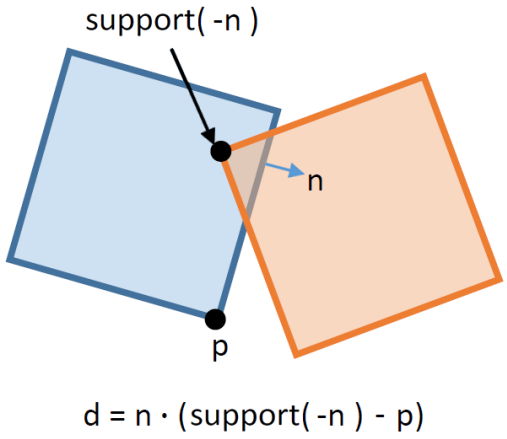

Figure 3.7: Face/vertex test example in two dimensions. The code sample demonstrates how to obtain a support vertex given a direction vector.

As it turns out, the test for three dimensions additionally requires certain pairs of edges, one from each body, to be tested, as the face/vertex tests may not be enough to find a plane when the bodies have certain orientations. The test is similar to the face/vertex test, using two vertexes belonging to each edge and comparing their distance along the axis resulting from the cross product of the vectors formed by the vertices of both edges (which corresponds to the normal of a plane containing either or both edges), however, only edges that form a face in the Minkowski difference of the geometry of the two bodies should be tested [13].

The algorithm keeps track of the face and distance of least penetration, that is, the face of a body with the greatest distance to the other body, which in the case of intersecting bodies is negative. If no combination of face and vertex, or edges, is found to have a positive signed distance, then the bodies are penetrating. However, if at any point a plane of separation is found, the algorithm finishes immediately, so situations in which the bodies are not colliding are when the algorithm is most efficient. One difference between the CPU and GPU implementation is that the CPU is capable of caching this plane of separation and reuse it in subsequent frames, since if the bodies remain disjoint, it is likely that the plane will still be a plane of separation. The GPU, however, does not cache this result, and repeats all the tests in each frame, at least until finding a plane of separation.

Lastly, after determining whether two bodies collide, each one is stored, along with information regarding the collision, in another overlap list, similarly allocated by the simulator[1], and contact generation is executed for all bodies. The contact generation step is again similar between

[1]The criterion for reallocating the broad-phase overlap list is also used for reallocating the narrow-phase overlap list, and both are reallocated at the same time.

CPU and GPU, with the main difference being that the face and vertex buffers are pre-allocated with a fixed size since dynamic allocation is disallowed on the GPU. This imposes a limit on the complexity of the geometry, which for many use cases is acceptable, as geometry for real-time physics tends to be simplified for performance. The results of contact generation are stored in the overlap list alongside the data of the corresponding contact.

3.4 Collision resolution

Once the collision detection step is finished, the collision resolution step follows, responsible for determining the effects of the collision on the bodies and updating their position depending on their velocity and the time elapsed since the last frame, based on the algorithms mentioned in subsection 2.1.3. The first operation consists in computing the impulses from the forces applied to the bodies, and update the momenta of the bodies. The solving of collisions itself involves using the contact information generated by the narrow-phase collision detection step, such as contact points, contact normal, and penetration depth, to compute a set of impulses that are applied to each of the bodies to simulate the impact. The sequential impulses algorithm is an iterative numerical method that calculates these impulses through successive approximations using equations that represent constraints on the position and velocity of the two bodies that are colliding. When the collisions are solved, the momenta resulting from the application of the contact impulses are integrated with respect to time to obtain the updated positions and orientations of each body for that frame.

Both the computing of impulses from forces and the velocities/momenta integration are embarrassingly parallel, and every body can be updated at the same time during these steps. The main difficulty in extracting parallelism from this step consists in ensuring no two threads update the same body at the same time during the collision solving, namely in the application of the computed impulses, and as such, the contacts must somehow be sequentialized depending on the bodies involved.

Detailing now the collision resolution algorithm, to understand why this sequentialization is needed and how it may be achieved, some concepts will be explored. The state of a body $s(t)$ can be described at each moment t in terms of velocity v, angular velocity ω, position p, and orientation θ:

$$s(t) = \begin{pmatrix} v(t) \\ \omega(t) \\ p(t) \\ \theta(t) \end{pmatrix} \tag{3.7}$$

For the prototype implementation, velocity, angular velocity, and position each are described by a vector, while orientation is described by a unit quaternion. Each body also has certain constants associated with it, such as mass m and rotational inertia I, given by a scalar and a 3×3

matrix, respectively. At each frame, the state of a body is updated using the state from the previous frame, the resulting forces f_r and torques τ_r acting on the body, and the elapsed time Δt between frames, through the following equations:

$$\begin{cases} v'(t) = v(t) + m^{-1}(f_r \Delta t) \\ \omega'(t) = \omega(t) + I^{-1}(\tau_r \Delta t) \\ p'(t) = p(t) + v(t)\Delta t \\ \theta'(t) = \theta(t) + \frac{1}{2} Q(t)\omega(t)\Delta t \end{cases} \tag{3.8}$$

Where $Q(t)$ is a 4×3 matrix that results from the time derivative of the angular position quaternion, such that:

$$Q(t) = \begin{bmatrix} -q_x(t) & -q_y(t) & -q_z(t) \\ q_w(t) & -q_z(t) & q_y(t) \\ q_z(t) & q_w(t) & -q_x(t) \\ -q_y(t) & q_x(t) & q_w(t) \end{bmatrix} \tag{3.9}$$

Equation 3.8 represents the necessary calculations at each frame for integrating the forces and velocities of a body with respect to time [5].

Describing now the constraint solver, for any collision between two bodies, their position and orientation can be represented by $s(t)$, while their linear and angular velocities can be represented by $v(t)$, and their mass and rotational inertia can be represented by the 12×12 matrix M, where E is the 3×3 identity matrix, such that:

$$s(t) = \begin{pmatrix} p_1(t) \\ \theta_1(t) \\ p_2(t) \\ \theta_2(t) \end{pmatrix} \in \mathbb{R}^{14} \tag{3.10}$$

$$v(t) = \begin{pmatrix} v_1(t) \\ \omega_1(t) \\ v_2(t) \\ \omega_2(t) \end{pmatrix} \in \mathbb{R}^{12} \tag{3.11}$$

$$M = \begin{bmatrix} m_1 E & 0 & 0 & 0 \\ 0 & I_1 & 0 & 0 \\ 0 & 0 & m_2 E & 0 \\ 0 & 0 & 0 & I_2 \end{bmatrix} \tag{3.12}$$

Additionally, the forces and torques acting on those bodies at a given frame can be described by:

$$F_r = F_e + F_c \tag{3.13}$$

$$F_e = \begin{pmatrix} f_{e_1} \\ \tau_{e_1} \\ f_{e_2} \\ \tau_{e_2} \end{pmatrix} \tag{3.14}$$

$$F_c = \begin{pmatrix} f_{c_1} \\ \tau_{c_1} \\ f_{c_2} \\ \tau_{c_2} \end{pmatrix} \tag{3.15}$$

For each collision between two bodies, the position of each body should be constrained so that their geometry does not interpenetrate. Then, for each contact point between two bodies, the following position constraint must depend on the position state s of the two bodies:

$$C(s) \geq 0 \tag{3.16}$$

Where $C(s)$ represents the distance between the two bodies along the axis of minimum penetration (typically this is the contact normal).

Additionally, to achieve a physically realistic behavior, there is also a need to constrain the relative velocity between the two bodies to impede them from interpenetrating. Thus, the velocity constraint can be obtained by taking the time derivative of equation 3.16, which follows:

$$\dot{C}(s) = Jv(t) = 0 \tag{3.17}$$

Where J is a 1×12 matrix known as the *Jacobian matrix* of the constraint. The contact forces must keep the constraints satisfied, but they should not change the energy of the system, so they can be described in terms of the Jacobian matrix, such that:

$$F_c = J^T \lambda \tag{3.18}$$

Since the Jacobian matrix is known for each of the constraints, the contact forces can be obtained if the λ parameter is found. It can be proven that the contact force does not work by computing the power P of the system through the following equation:

$$P = F_c v(t) = F_c^T v(t) = (J^T \lambda)^T v(t) = \lambda^T J v(t) = 0 \tag{3.19}$$

However, there may be a need to correct the position of the bodies, since these may be interpenetrating (and thus in violation of the position constraint), so the velocity constraint in those cases takes the form:

$$\dot{C}(s) = Jv(t) + b \geq 0 \tag{3.20}$$

Where b is known as the *bias velocity vector*. If b is not zero, the system's energy changes since the contact force must do work to correct the bodies from interpenetrating. Typically, the bias velocity vector is used to not only correct the positional drift of the two bodies, by computing a velocity in terms of the penetration depth and Δt, but to also calculate a bounce velocity, by computing the new velocity in terms of the relative velocity of the two bodies and some restitution coefficient e, where $0 \leq e \leq 1$.

In order for the constraints to be satisfied, the updated velocities must respect inequation 3.20, such that:

$$v'(t) = v(t) + M^{-1} F_r \Delta t = v(t) + M^{-1}(F_e + F_c)\Delta t \tag{3.21}$$

$$
\begin{aligned}
Jv'(t)+b=0 \Leftrightarrow & J(v(t)+M^{-1}(F_e+F_c)\Delta t)+b=0 \Leftrightarrow \\
& J(v(t)+M^{-1}F_e\Delta t)+JM^{-1}F_c\Delta t+b=0 \Leftrightarrow \\
& J(v(t)+M^{-1}F_e\Delta t)+JM^{-1}J^T\lambda\Delta t+b=0 \Leftrightarrow \\
& Jv'_e(t)+K\lambda\Delta t+b=0 \text{ with } \begin{cases} v'_e(t)=v(t)+M^{-1}F_e\Delta t \\ K=JM^{-1}J^T \end{cases} \Leftrightarrow \\
& K\lambda\Delta t=-(Jv'_e(t)+b) \Leftrightarrow \\
& \lambda\Delta t=-K^{-1}(Jv'_e(t)+b) \Leftrightarrow \\
& \lambda'=-K^{-1}(Jv'_e(t)+b) \text{ with } \lambda'=\lambda\Delta t
\end{aligned} \tag{3.22}
$$

$$
v'(t)=v'_e(t)+M^{-1}J^T\lambda' \tag{3.23}
$$

Where K is known as the *effective mass* of the constraint, and the velocity v'_e is computed from the external force. For each contact point of a collision, the position constraint is described in terms of the relative positions of the contact point to each body, the contact normal, and the penetration depth, and is then derived to obtain the velocity constraint and thus describe the Jacobian matrix of the constraint[9] [16].

Hence, to satisfy these constraints, the contact forces between two bodies must be computed at once by solving equation 3.22 and applying the contact impulses through equation 3.23. As such, it is important that the same body is not being updated by more than one thread at the same time, and so different collisions involving the same body must be solved sequentially. In the native simulator, this is solved trivially by simply requiring that all constraints in a cell are solved by the same thread, so that different cells are distributed between threads, instead of constraints or bodies. However, in the SYCL simulator, this form of partition of work requires knowledge of the dimensions of the uniform grid at the time of allocation of memory for the constraints, the acquiring of which is not viable for performance reasons, besides sacrificing a great amount of parallelism inherent to the device, and so a different scheme of division must be used.

Instead, considering d dimensions, the uniform grid partitioning is pre-computed using the number of estimated overlaps $n_{overlap}$ (the same one used for allocating the overlap lists used during the collision detection step) divided by the product of the maximum size of a work-group s_{wg} with the number of cell groups, obtained from 2^d, as explained in subsection 2.2.3, resulting in an initial estimate of the necessary number of work-groups i_{wg}, which is then used to calculate the number of cells per side of the grid s_{gc}. This number is then used to calculate both the actual number of work-groups n_{wg}, and the number of work-items per work-group n_{wi} corresponding to the maximum number of constraints processed per work-group, which must be less than or equal to the maximum size of a work-group $n_{wi} \leq s_{wg}$. This property is ensured by always rounding up divisions using a ceiling function. Additionally, the s_{gc} parameter is also used in the uniform grid partitioning algorithm during the broad-phase to calculate the correct cell index and cell group of each body. These numbers are obtained according to the formulas described by equations 3.24,

3.25, 3.26, and 3.27, and an example partitioning can be seen in figure 3.8.

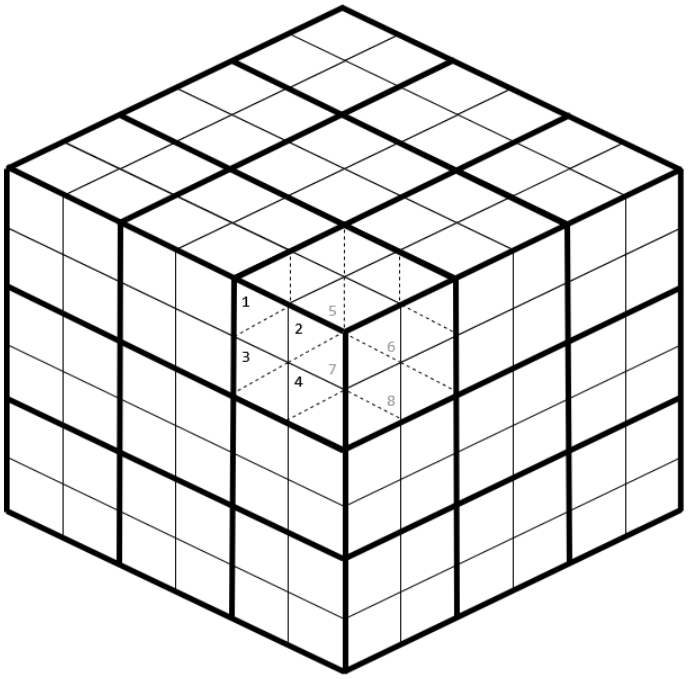

Figure 3.8: Example partitioning in 3 dimensions of 36 constraints with a maximum work-group size of 4. The thicker lines represent the division in grid cells, the thinner lines represent the division in cell groups. Parameters: $s_{gc} = 3$; $n_{wg} = 27$; $n_{wi} = 1$.

$$i_{wg} = \left\lceil \frac{n_{overlap}}{2^d s_{wg}} \right\rceil \tag{3.24}$$

$$s_{gc} = \left\lceil \sqrt[d]{i_{wg}} \right\rceil \tag{3.25}$$

$$n_{wg} = s_{gc}^d \tag{3.26}$$

$$n_{wi} = 2^{\left\lceil log_2\left(\frac{n_{overlap}}{2^d n_{wg}}\right)\right\rceil} \tag{3.27}$$

Characterizing now the collision resolution step as a whole, the general structure of the dynamics computation algorithm is described in figure 3.9.

Regarding the steps that comprise the dynamics computation, both the preparation and the solving of the constraints require sequentialization of the constraints. Hence, the constraint batching algorithm, described in subsection 2.2.6, is executed before the dynamics computation to prepare the constraints for parallel processing by sequentializing them as necessary to prevent concurrent modifications to the same body. Aside from the batching algorithm, constraints in each cell group are also sequentialized relative to other cell groups, with each one of these being processed by a different kernel launch. These two steps of sequentialization can be seen in figure 3.10.

```
void solver::compute_dynamics(unsigned int substep_count, float substep_delta)
{
    for (unsigned s = 0; s < substep_count; ++s)
    {
        compute_external_impulses(substep_delta);

        precompute_constraints(substep_delta);

        for (unsigned i = 0; i < iteration_count; ++i)
        {
            solve_constraints(substep_delta);
        }

        integrate_velocities(substep_delta);
    }
}
```

Figure 3.9: Algorithm for the dynamics computation. The GPU and the CPU version are similar in structure.

```
#define CELL_GROUP_COUNT 8

void solver::solve_constraints(float substep_delta)
{
    for (unsigned cell_group = 0; cell_group < CELL_GROUP_COUNT; ++cell_group)
    {
        launch_kernel(
            num_constraint_work_groups, num_constraint_work_items,
            [=]{
                unsigned group_id = get_work_group_id();
                unsigned thread_id = get_work_item_id();

                unsigned constraint_id = group_id * num_constraint_work_items + thread_id;
                auto& constraint = constraint_arr[cell_group][constraint_id];

                for (unsigned batch = 0; batch < batch_count; ++batch)
                {
                    if (constraint.batch_index == batch)
                    {
                        constraint.solve(mass_arr, momentum_arr, substep_delta);
                    }

                    get_work_group_barrier().wait();
                }
            }
        );
    }
}
```

Figure 3.10: Algorithm for constraint solving on the GPU. The `precompute_constraints` method is similar in structure to this method.

The implementation of the batching algorithm used by the prototype deviates slightly from the one described in the article. In particular, it uses a global array for marking bodies during batching, and instead allocates enough memory for $n \times c$ bodies, where n is the number of bodies and c is the number of cell groups, such that each constraint locks the bodies relative to the cell group that was attributed to the constraint, *e.g.*, if a collision is registered in body a between bodies a_0 and b_1, where the indexes represent the cell groups that each body belongs to, then the constraint attempts to mark bodies a_0 and b_0. The reasoning is that even if two separate constraints in different cell

groups try to mark the same body during an iteration, they should be able to succeed since they will be processed sequentially anyways due to belonging to different cell groups. Additionally, it skips the batch sorting after the batching step, due to the high local memory requirements it imposes for situations with many constraints, thus sacrificing some data locality when running the solver.

One alternative implementation that this work briefly explored involved further simplifying the constraint batching algorithm by not dividing the constraints based on the size of each work-group, and instead use a global buffer with atomic operations to attribute batches to each constraint. This implied using kernel launches to synchronize work-groups relative to each batch during constraint solving, which had a significant negative impact on performance and led to this approach being discarded by this work.

Looking at the constraint solver implementation, it is worthy of mention that the algorithms used by the native and SYCL simulators differ slightly. Namely, the CPU keeps a temporary cache of previous constraints, associated with the bodies, and reuses them to provide an estimate of the impulses that will be calculated through numerical approximation. This technique is known as warm starting, and is important for increasing the stability of real-time simulation, as usually there is no possibility of running iterations until convergence to a solution within the time constraints of a frame. The native simulator implements a naive, simplified version of this technique, and the SYCL simulator lacks this optimization entirely.

3.5 Summary

This chapter described the structure and main components of a game engine, which the prototype follows, referring to some key components, such as the renderer responsible for representing the virtual world, the input managers responsible for consuming and routing the user's input to other parts of the engine, and the simulators, responsible for changing the state of that world and the entities that exist in it, by representing them as rigid bodies susceptible to forces and torques. Then, the particulars of the GPU simulator were presented, mentioning some core logic, such as the necessity of simulating one frame ahead of the rest of the state of the engine to reduce the latency caused by data transfers between CPU and GPU. Lastly, a more comprehensive overview of each algorithm used in the different phases of a simulation was given, clarifying some aspects of adapting these algorithms for execution on the GPU.

Chapter 4

Results

The following chapter will delineate the performance results obtained from multiple executions, with different configurations, across several environments. Multiple devices were used for performing the tests, and the different configurations for the engine will be detailed in section 4.1. The performance measurements will be shown in section 4.2, occasionally summarizing the findings and establishing a comparison between the devices, and finally, a discussion of these results will be presented in section 4.3, elaborating on the findings and drawing some conclusions regarding the algorithms.

4.1 Test environment

The tests were performed with the use of a high-end AMD CPU and three high-end Nvidia GPUs of different generations. The specifications of the hardware used for the tests are summarized in table 4.1.

Device Name	Nr. of cores	Clock frequency	GDS size	LDS size
AMD Ryzen 9 3950X	32[1]	3.50 GHz	32 GB	N/A
NVIDIA GTX 970	13	1.253 GHz	4 GB	48 KB
NVIDIA GTX 1060	10	1.759 GHz	6 GB	48 KB
NVIDIA RTX 2070 SUPER	40	1.785 GHz	8 GB	48 KB

Table 4.1: Specifications for hardware used for the tests.

Additionally, all GPUs possess a maximum work-group size of 1024 work-items/threads.

In all of the results presented henceforth, due to the implementation of the engine, the initial seconds may display incongruous behavior with the rest of the data, because the engine forces a limit on the number of entities spawned per frame, and so spreads the spawns across frames.

[1]This processor has 16 physical cores, but 32 logical cores

This is especially prevalent in environments with a great number of entities, and each spawn has a non-trivial cost since each requires allocation of several data structures along with registration of these in several places in the engine, namely the simulator itself. Aside from this situation, all the other subsystems have a trivial impact on the performance of the engine, and as such, it can be assumed that the simulation is solely responsible for the variation of the frame rate.

Before presenting the results of each test, it is important to characterize the performance of the native simulator with various amounts of dedicated threads. The results presented in figures 4.1 to 4.9 were obtained from tests on the second environment (which will be described further on) and show the performance of each component for 1, 8, 16, or 32 threads dedicated to the simulator. The reason for using this environment was due to the high number of moving bodies and collisions per frame, which imply heavy workloads on all components of the simulation.

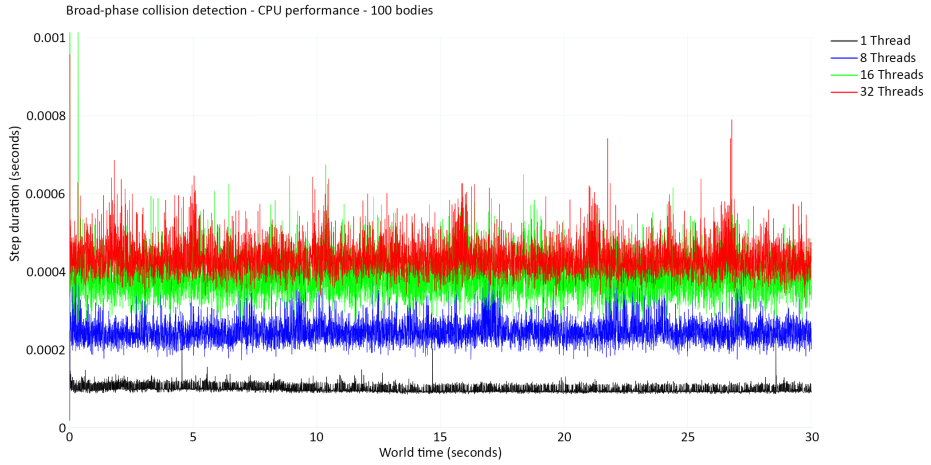

Figure 4.1: Performance of broad-phase collision detection step in CPU with 100 bodies.

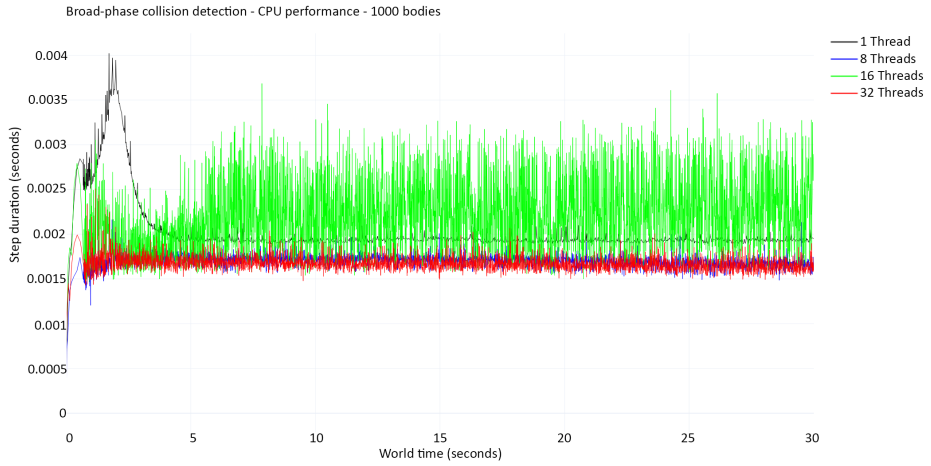

Figure 4.2: Performance of broad-phase collision detection step in CPU with 1000 bodies.

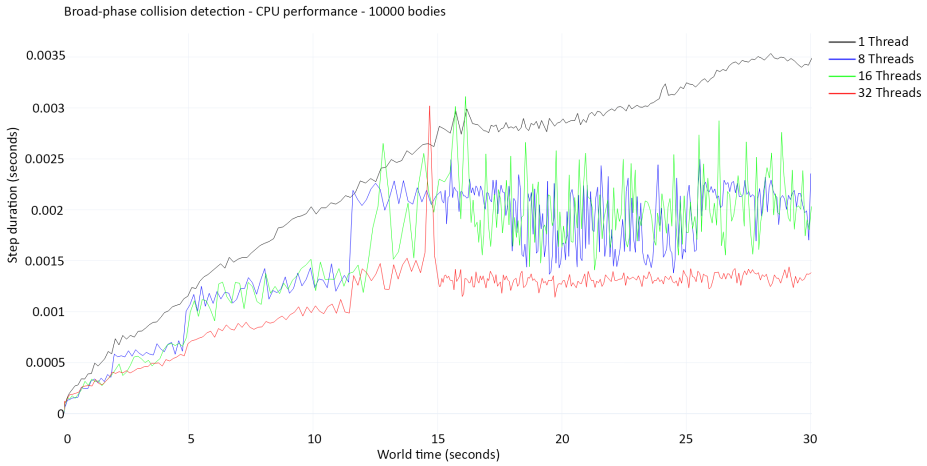

Figure 4.3: Performance of broad-phase collision detection step in CPU with 10000 bodies.

Analyzing the results for the broad-phase step, it can be seen that the algorithm benefits from a higher number of threads at higher body counts, however, on average, 8 threads seem to display results comparable to a larger amount of dedicated threads.

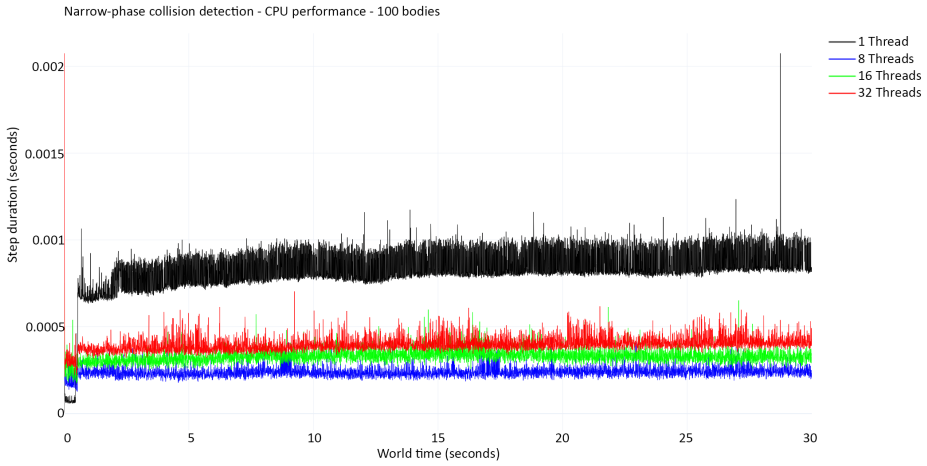

Figure 4.4: Performance of narrow-phase collision detection step in CPU with 100 bodies.

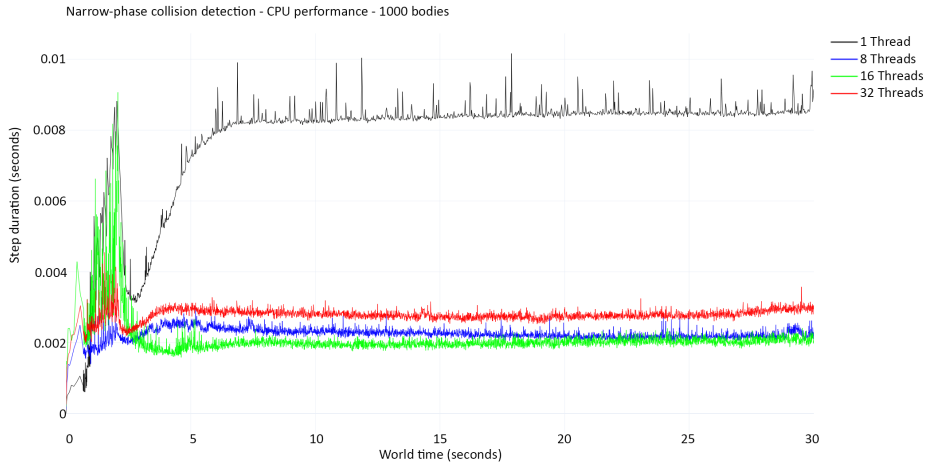

Figure 4.5: Performance of narrow-phase collision detection step in CPU with 1000 bodies.

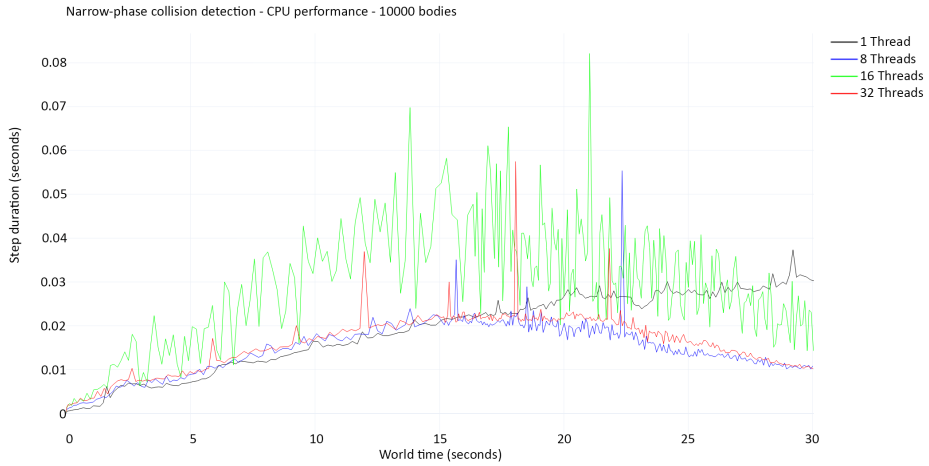

Figure 4.6: Performance of narrow-phase collision detection step in CPU with 10000 bodies.

Referring now to the narrow-phase step results, it can be seen again that a higher number of threads improves performance at higher body counts, with 8 threads striking a fine balance between machine parallelism and the characteristics of the algorithm.

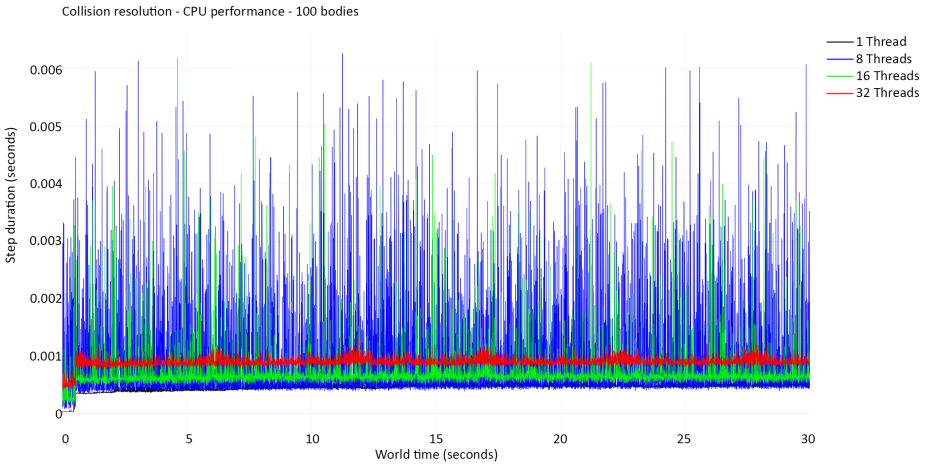

Figure 4.7: Performance of collision resolution step in CPU with 100 bodies.

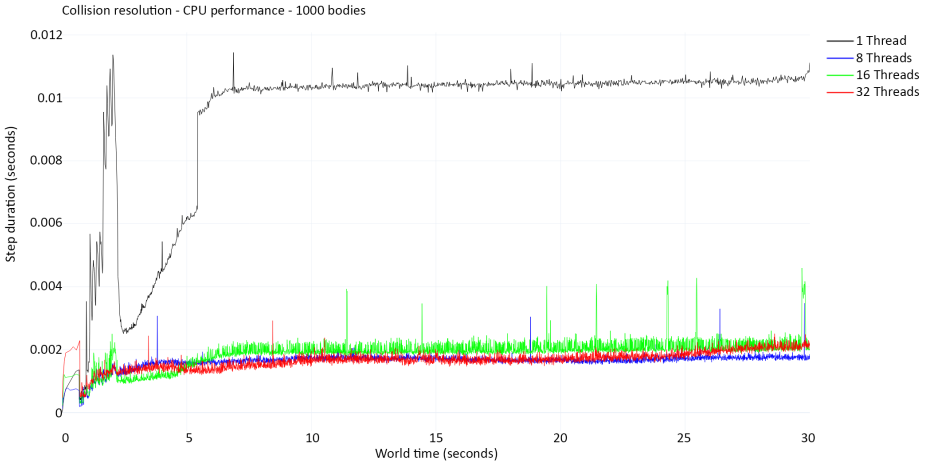

Figure 4.8: Performance of collision resolution step in CPU with 1000 bodies.

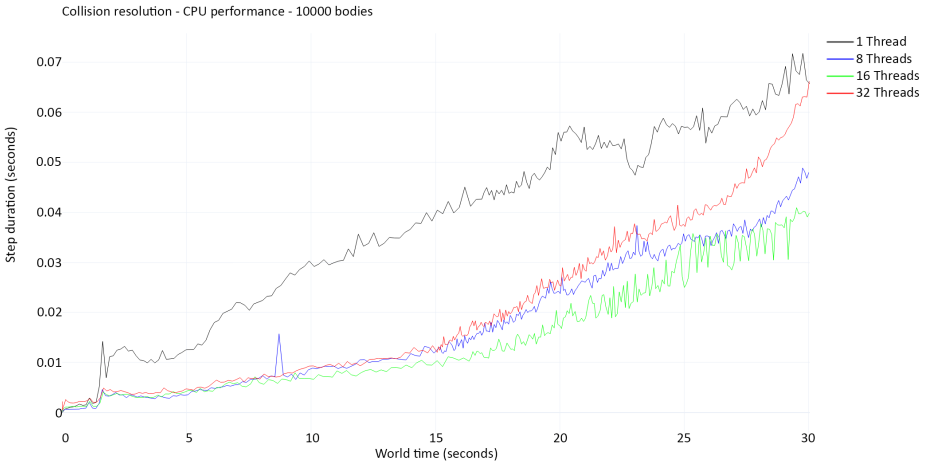

Figure 4.9: Performance of collision resolution step in CPU with 10000 bodies.

Lastly, for the collision resolution step, the tendency is again for performance to improve in a multi-threaded environment, though the difference between performance for various numbers of dedicated threads seems to be less significant for this component.

For the most part, it can be seen that all components benefit from a higher number of threads dedicated to the simulator, regardless of the number of bodies being simulated, though in the case of the tested processor a point of diminishing returns is reached for a number greater than 8 or 16 threads for the narrow-phase and the collision resolution, depending on the number of bodies simulated, above which the simulation shows worse performance for a higher number of threads.

Only the broad-phase shows better results with 32 threads on higher body counts, with 8 threads showing more consistent results overall across all other steps.

As such, the results for the tests presented henceforth will show only the CPU results with 8 threads dedicated to the simulator.

4.2 Performance measurements

To test the performance and stability of the simulators, four different environments were prepared, with different characteristics concerning the interactions between the bodies. In each environment, tests were run with a varying number of bodies in the world. The bodies used in all tests were cubes or cuboids (in the case of the ground) with the same dimensions, mass, and rotational inertia. The results of the execution of each test were sampled at the end of each frame, for thirty seconds of simulation time, thus the number of samples varied depending on the environment and the test configuration. The graphs for each environment were obtained by simulating with 100, 1000, and 10000 bodies, with the exception of the fourth environment, which was run with 55, 385, and 1240 bodies instead, and are presented as a comparison of the execution time of each device, per component of simulation, per number of simulated bodies. The SYCL simulator was configured to run two frames ahead of the engine for all tests, to potentially reduce the impact of data transfers on the results of the tests. As a side note, no results for the GTX 970 were obtained for simulation with 10000 bodies due to the low memory capacity of the device.

4.2.1 Random drifting environment

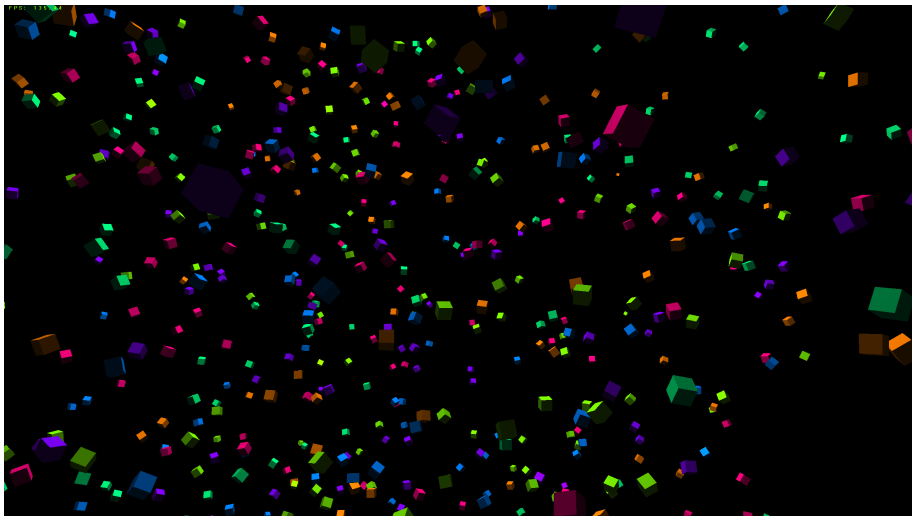

Figure 4.10: Random drifting test environment.

The first environment comprised a world with no gravity and bodies with random positions and orientations spread around the world, with random forces and torques applied to the bodies at arbitrary intervals. This environment has a low number of collisions, but highly variable positions of the bodies, so it mainly tested the performance of the broad-phase collision detection, which is more susceptible to alterations to the bodies' positions.

With regards to the broad-phase collision detection step, the results can be seen in the figures 4.11, 4.12, and 4.13.

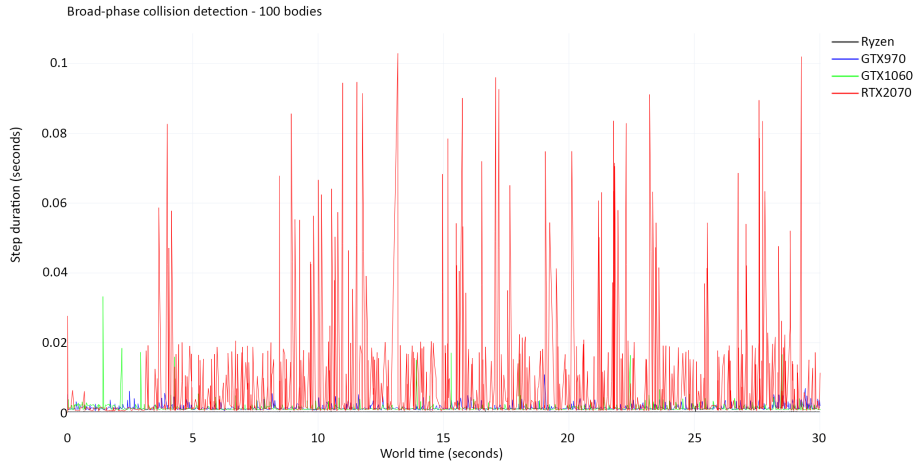

Figure 4.11: Random drifting test broad-phase results with 100 bodies.

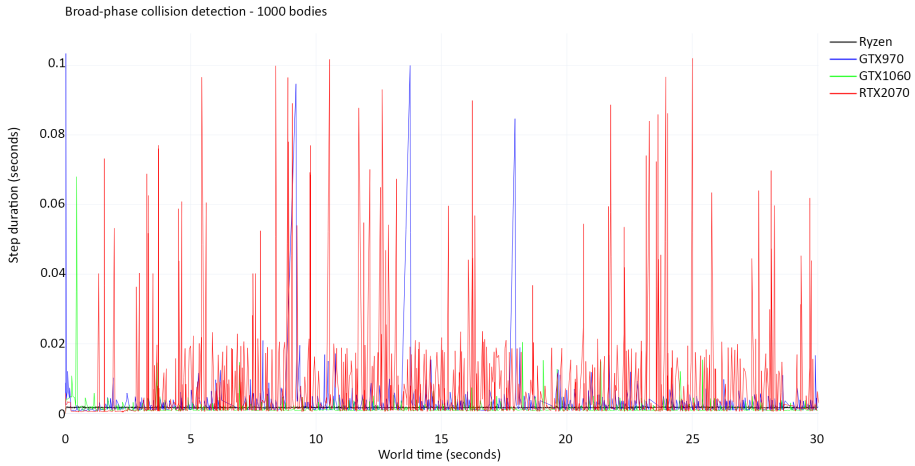

Figure 4.12: Random drifting test broad-phase results with 1000 bodies.

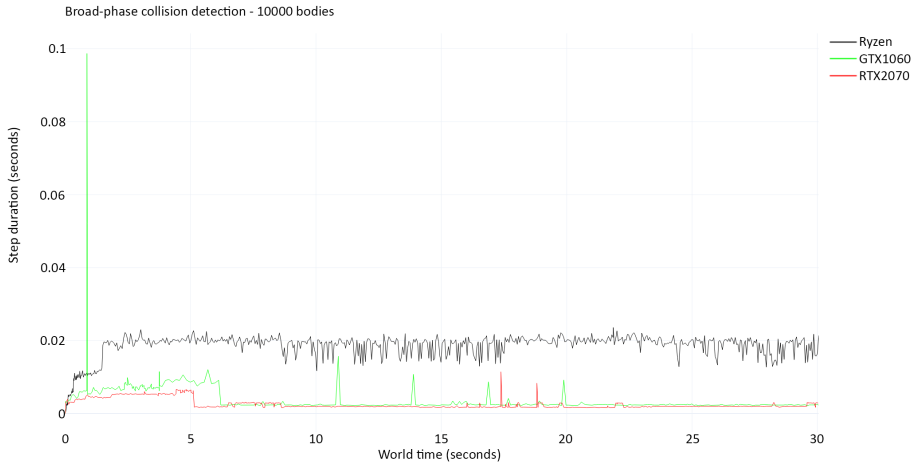

Figure 4.13: Random drifting test broad-phase results with 10000 bodies.

For 100 bodies, the CPU is better than any GPU, with the RTX 2070 even showing several spikes in execution times due to low arithmetic intensity, unable to compensate for the latency of memory transfers. For 1000 bodies, the GPUs begin to be similar or better performing than the CPU, though the RTX 2070 still suffers from the problem of low arithmetic intensity, with many spikes in execution times. For 10000 bodies, both GPUs are capable of surpassing the CPU with no issues.

Looking now at the narrow-phase collision detection step, the results can be seen in the figures 4.14, 4.15, and 4.16.

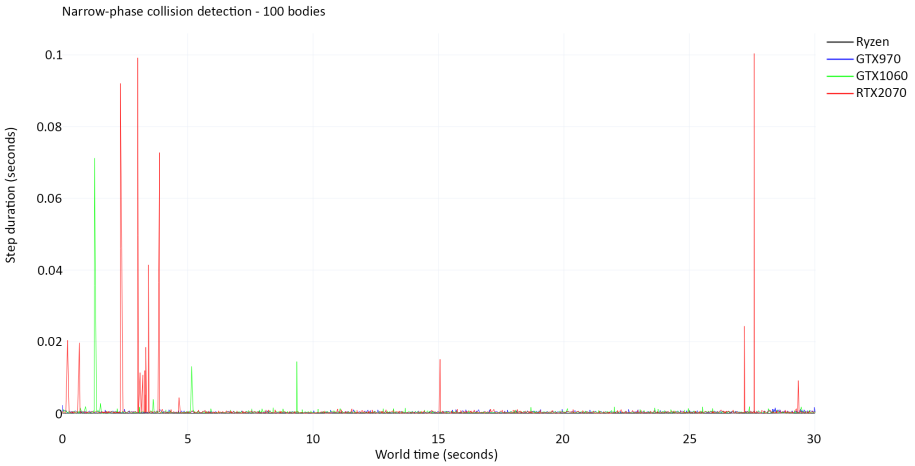

Figure 4.14: Random drifting test narrow-phase results with 100 bodies.

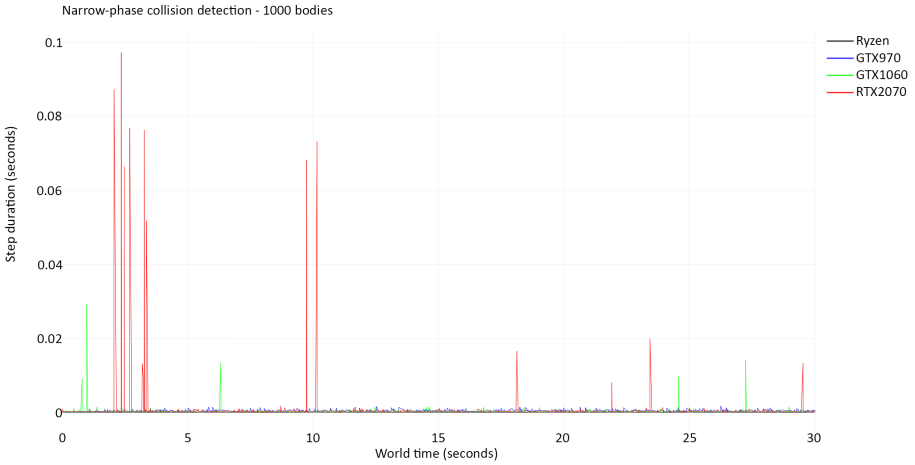

Figure 4.15: Random drifting test narrow-phase results with 1000 bodies.

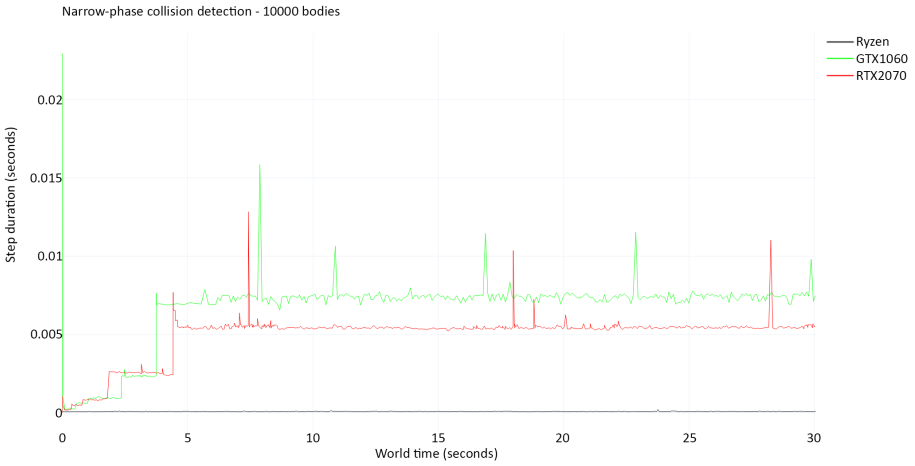

Figure 4.16: Random drifting test narrow-phase results with 10000 bodies.

In this test, due to the characteristics of the environment and how the narrow-phase step works in the SYCL simulator, no GPU was capable of surpassing the performance of the CPU, since situations with no collisions in the CPU result in a no-op for the narrow-phase. Since the GPU handles the workload through a parallel check to all bodies, it still requires checking each body to determine there is no collision to submit to the narrow-phase.

Finally, with respect to the collision resolution step, the results can be seen in the figures 4.17, 4.18, and 4.19.

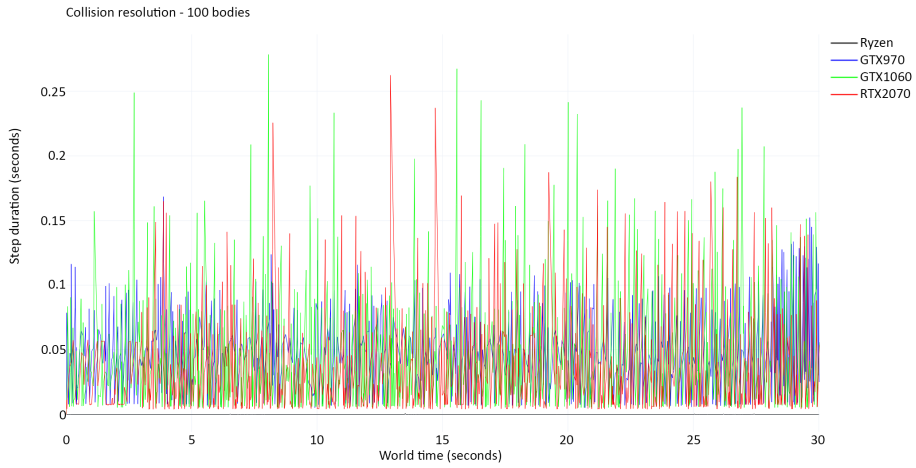

Figure 4.17: Random drifting test collision resolution results with 100 bodies.

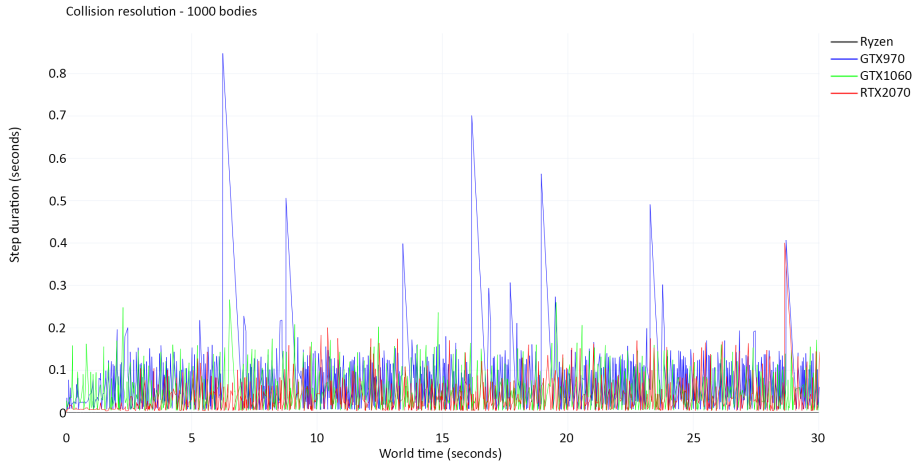

Figure 4.18: Random drifting test collision resolution results with 1000 bodies.

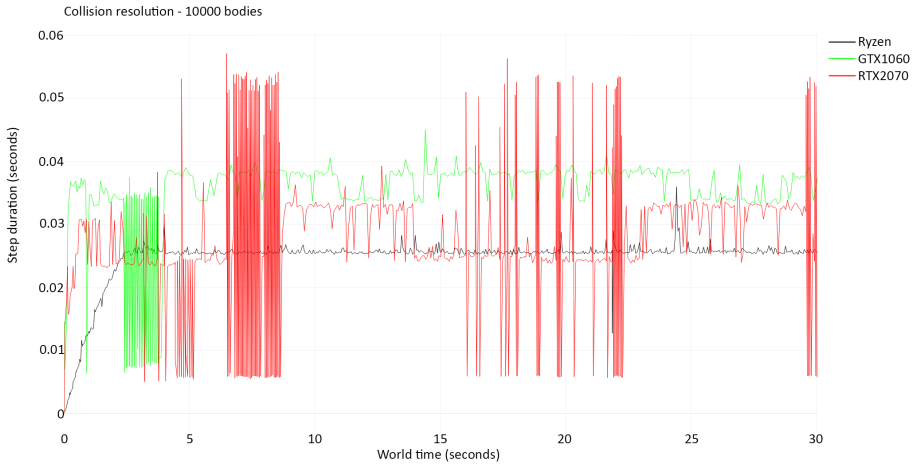

Figure 4.19: Random drifting test collision resolution results with 10000 bodies.

For 100 and 1000 bodies, the CPU performs better than any GPU, with these subject to the same execution spikes as the ones in the broad-phase step due to low arithmetic intensity in a situation with no collisions. For 10000 bodies, however, the RTX 2070 is on par with the CPU version, though it still shows occasional execution time spikes, with the GTX 1060 performing slightly worse but more consistently, as, despite the high number of bodies having an impact on the management work that the CPU has to do even with no collisions, the GPU has a non-trivial overhead associated with the execution of certain kernels even with no work needing to be done.

4.2.2 Layers falling environment

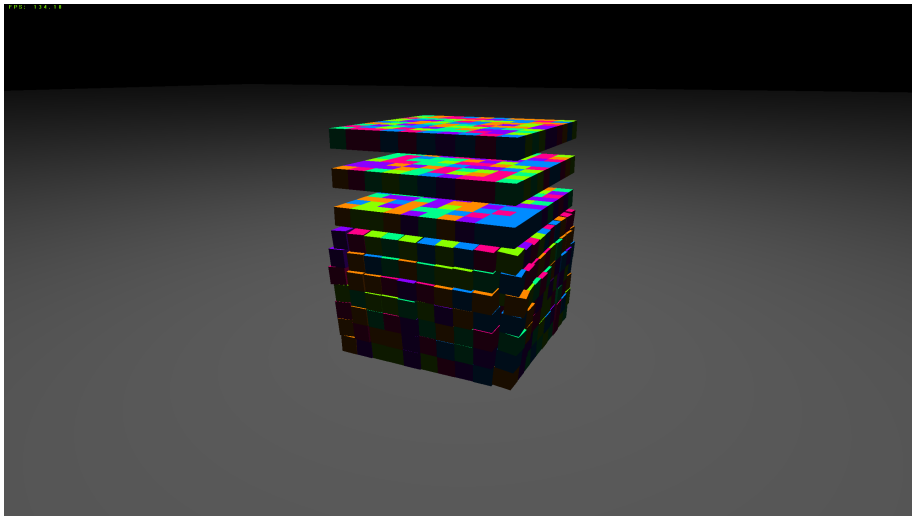

Figure 4.20: Layers falling test environment.

The second environment comprised a world with an immovable body with dimensions $1000 \times 1000 \times 1$ in the x, y, z axes, at $x = 0, y = 0, z = 0$, to function as the ground, and several movable bodies, subject to gravity, spawned side-by-side in a 10×10 disposition in the xy plane, in multiple

groups stacked vertically with increasing z, with the first group of bodies at $z = 1$. The bodies would proceed to fall to the ground, with each group falling on top of the groups below it. This environment has a high number of collisions, with each body colliding with at least two bodies below and above it, so has a higher impact on the narrow-phase and collision resolution steps than on the broad-phase, at least while the bodies remain close to each other. At high body counts, the bodies tend to bounce and spread out, so the execution times of the broad-phase tend to increase as well. Besides collisions, it also tests the stability of the simulation with respect to energy conservation, due to the high number of bouncing bodies, as well as to how well the simulator responds to an increasing number of collisions over time at high body counts.

The results for the broad-phase collision detection step can be seen in the figures 4.21, 4.22, and 4.23.

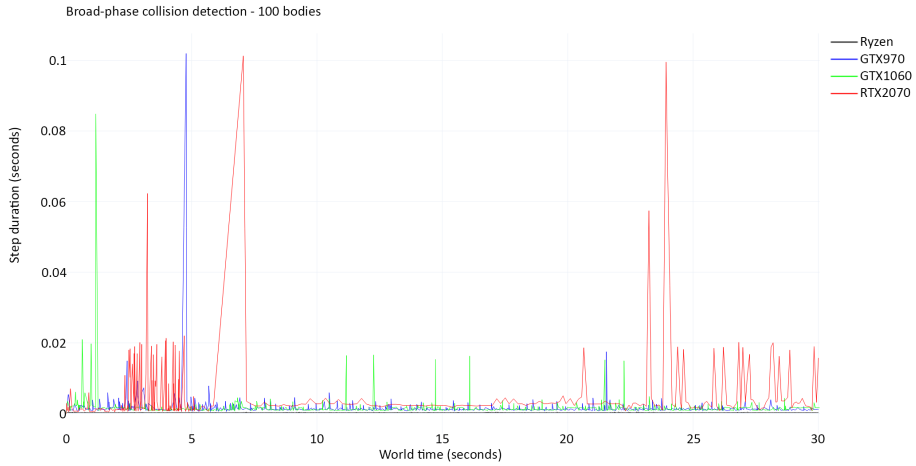

Figure 4.21: Layers falling test broad-phase results with 100 bodies.

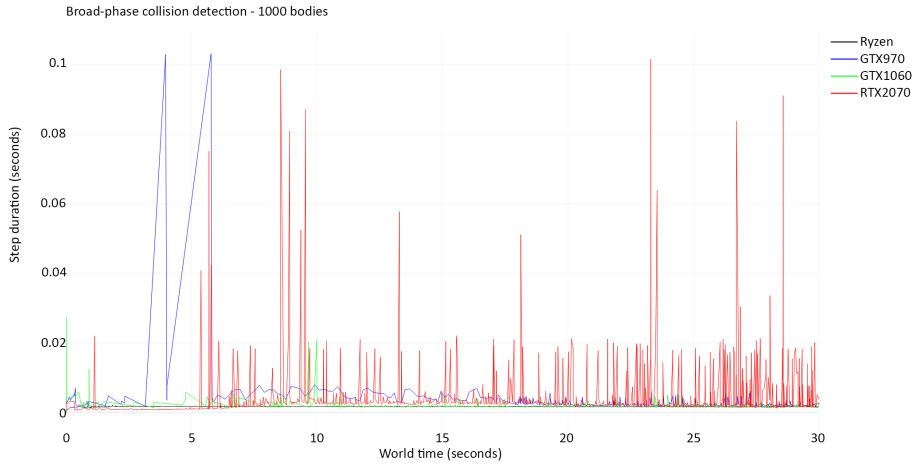

Figure 4.22: Layers falling test broad-phase results with 1000 bodies.

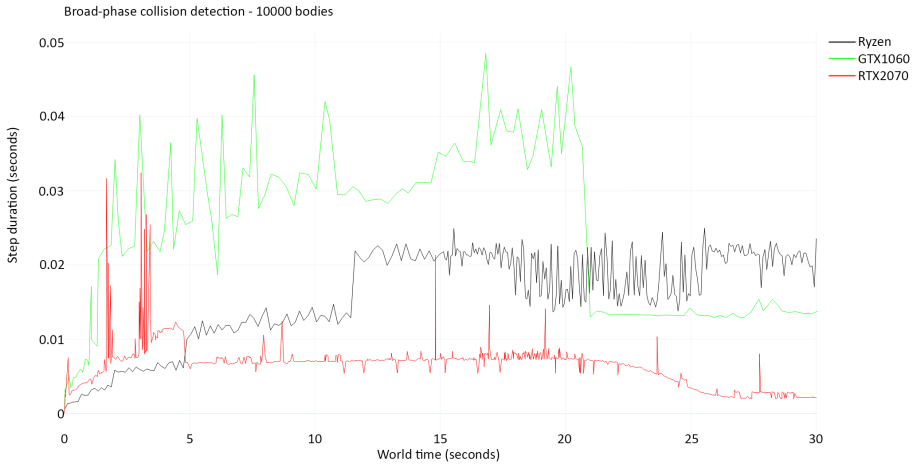

Figure 4.23: Layers falling test broad-phase results with 10000 bodies.

For 100 and 1000 bodies, the GPU and CPU implementations show comparable results, though the RTX 2070 stays behind the remaining devices, and again displays significant execution spikes, especially in the 1000 bodies execution run. As for the 10000 bodies execution, the RTX 2070 shows better execution time throughout the test, with the GTX 1060 staying behind the CPU until the 20 seconds mark, at which point the execution times drop below those of the CPU. This likely occurs due to the spawning of the bodies until the 20 seconds mark, delayed by the fact that the simulation is computationally heavy and thus decreases the frame rate, dragging out the spawns along time.

Next, the results for the narrow-phase collision detection step can be seen in the figures 4.24, 4.25, and 4.26.

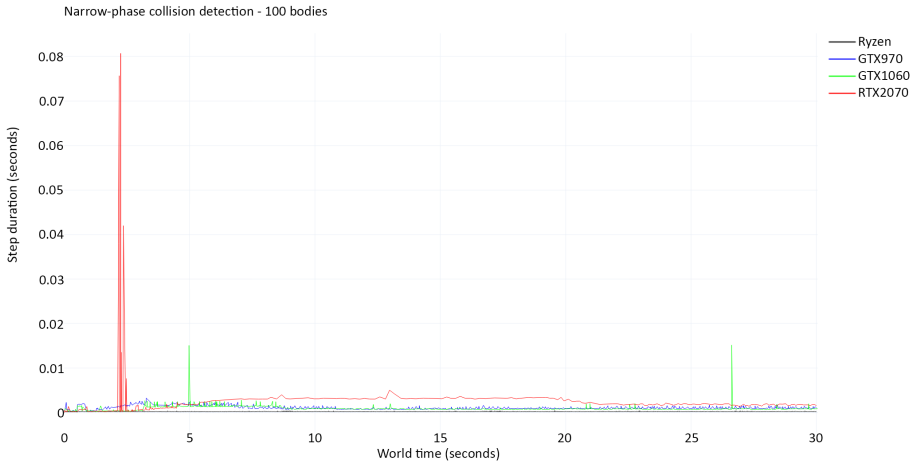

Figure 4.24: Layers falling test narrow-phase results with 100 bodies.

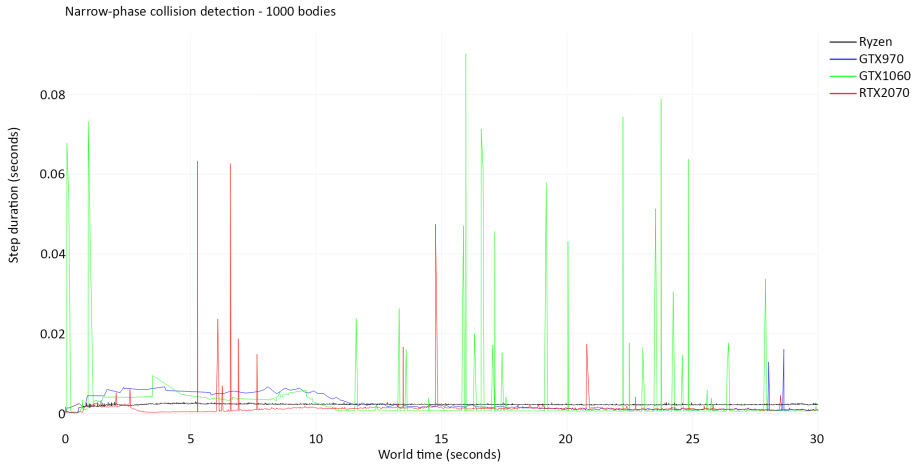

Figure 4.25: Layers falling test narrow-phase results with 1000 bodies.

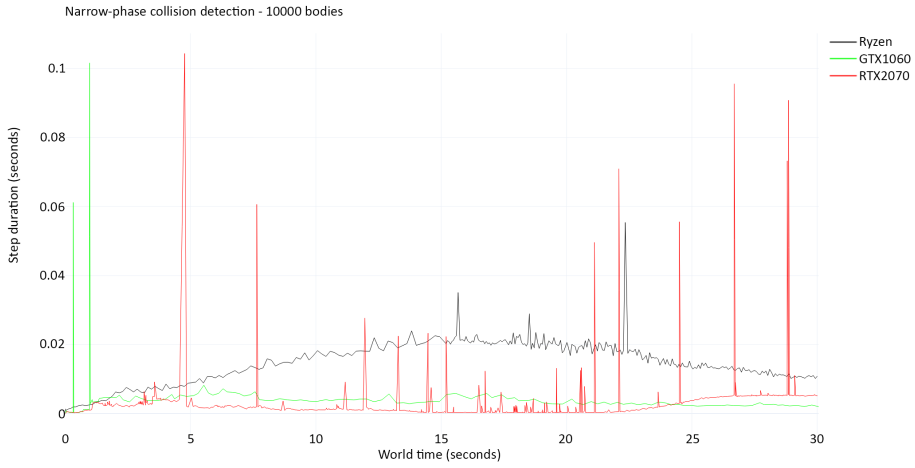

Figure 4.26: Layers falling test narrow-phase results with 10000 bodies.

With 100 bodies, all GPUs perform similarly, slightly worse than the CPU, with the RTX 2070 showing a decrease in execution time at a certain point to be closer, albeit slower, to the remaining devices. The 1000 bodies test shows better performance by the GPUs in relation to the CPU, although still similar, though the GTX 1060 displays a considerable amount of execution spikes. At 10000 bodies, both GPUs defeat the CPU in terms of execution time, with the RTX 2070 showing occasional spikes and a drop in the performance roughly at the 25 seconds mark to behind the GTX 1060, which shows a more constant execution time throughout the test.

Lastly, the collision resolution step results can be seen in the figures 4.27, 4.28, and 4.29.

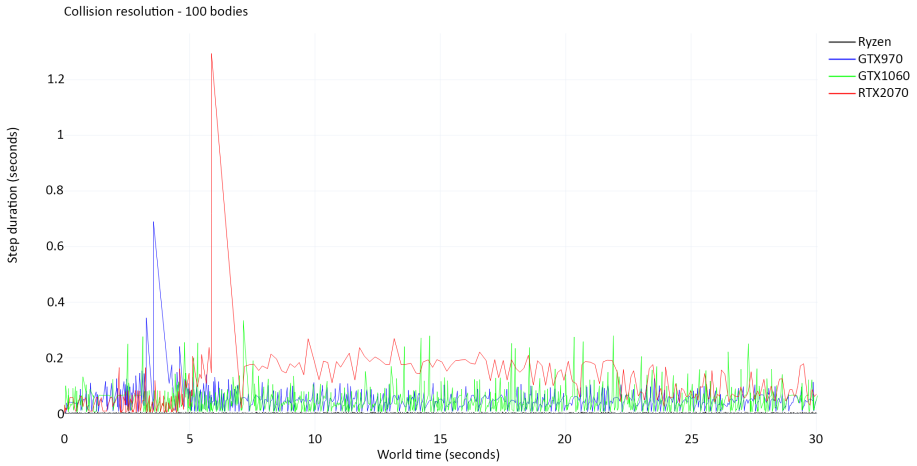

Figure 4.27: Layers falling test collision resolution results with 100 bodies.

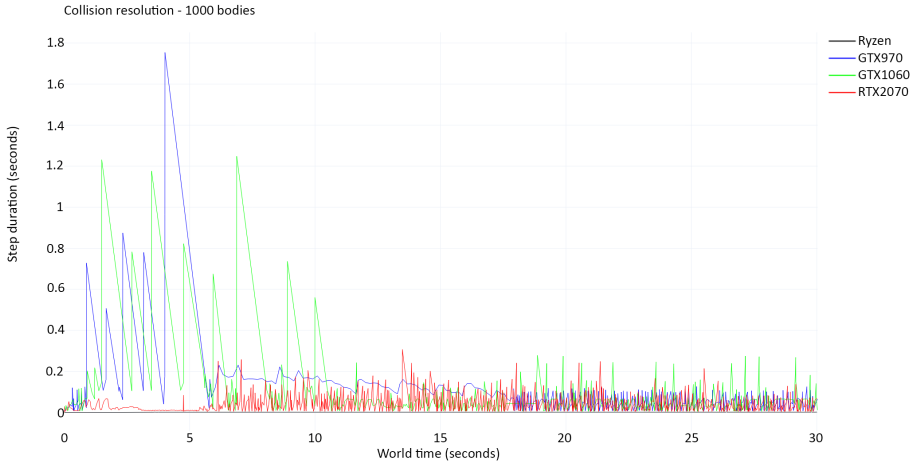

Figure 4.28: Layers falling test collision resolution results with 1000 bodies.

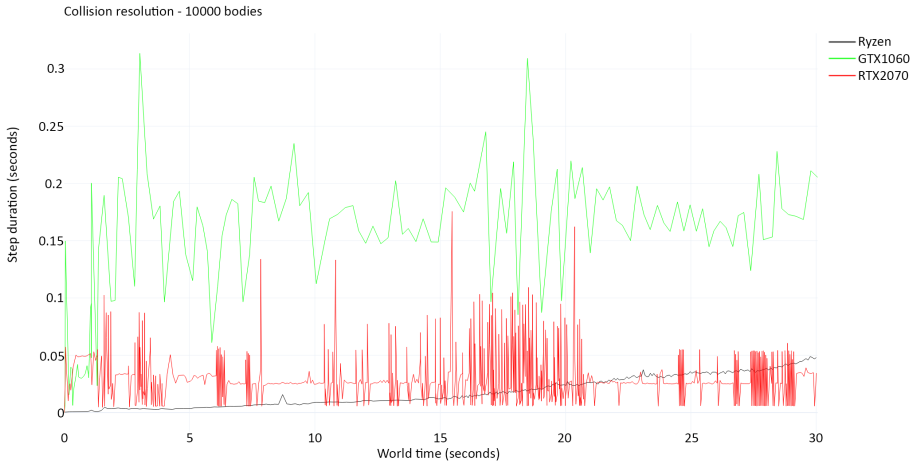

Figure 4.29: Layers falling test collision resolution results with 10000 bodies.

For 100 and 1000 bodies, the performance of the GPUs is significantly worse than the CPU, with several spikes in execution on all devices, reaching peaks of 200 milliseconds. When simulating 10000 bodies, only the RTX 2070 is capable of obtaining good execution times on average, despite frequent spikes reaching 150 milliseconds, roughly beating the CPU at the 20 seconds mark, while the GTX 1060 stays with performance similar to the 100 and 1000 bodies tests.

It should be noted at this point that the environment displays different behavior between the CPU and GPU implementations, in particular, in the test with 1000 bodies, while the bodies are capable of stacking and keeping some structure (as seen in figure 4.20) in the CPU version, in the GPU version they bounce off each other and spread across the ground. Some reasons for this behavior will be explored in section 4.3.

4.2.3 Spread bodies environment

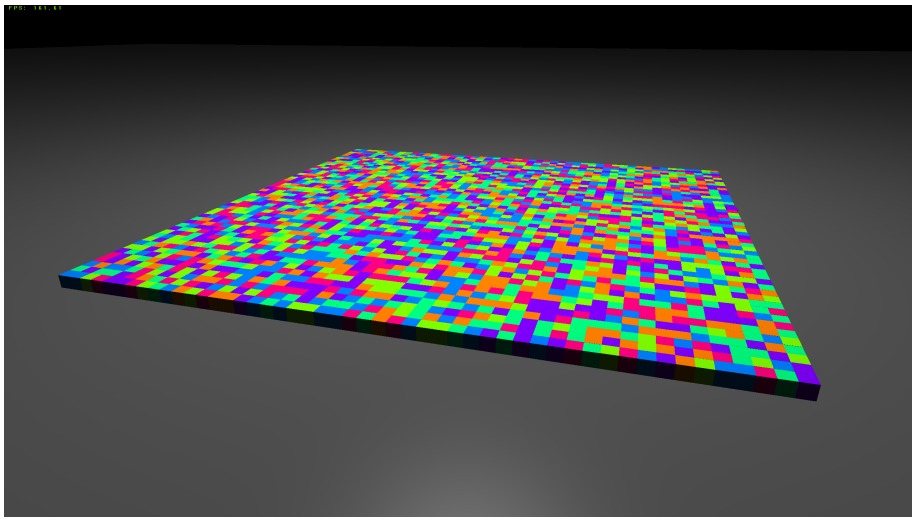

Figure 4.30: Spread bodies test environment.

The third environment consists of a world similar to the second, with an immovable body with the same dimensions at $z = 0$ functioning as the ground, and several movable bodies, subject to gravity, spawned side-by-side in a $n \times n$ disposition in the xy plane, in a single group at $z = 1$. This environment has a moderate number of collisions, with all bodies colliding at least with the ground body. The broad-phase execution times should remain constant, as the bodies maintain their positions relative to one another. The narrow-phase execution times should be moderate, as each body should not collide with one another, speeding up the test due to caching of previous results, but at least one contact with the ground should be detected for each body. The stability of the simulation is also tested with respect to the behavior of the collision resolution phase, as the bodies should not collide with those that are by their side.

Regarding the broad-phase collision detection step, the results can be seen in the figures 4.31, 4.32, and 4.33.

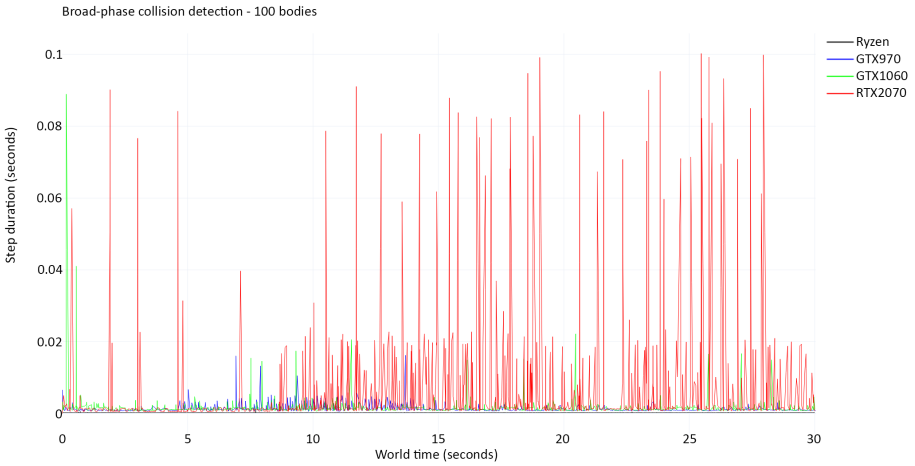

Figure 4.31: Spread bodies test broad-phase results with 100 bodies.

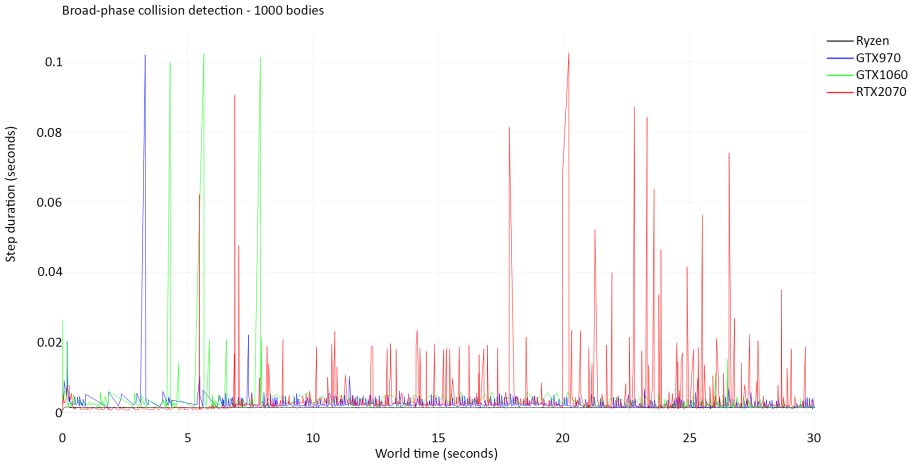

Figure 4.32: Spread bodies test broad-phase results with 1000 bodies.

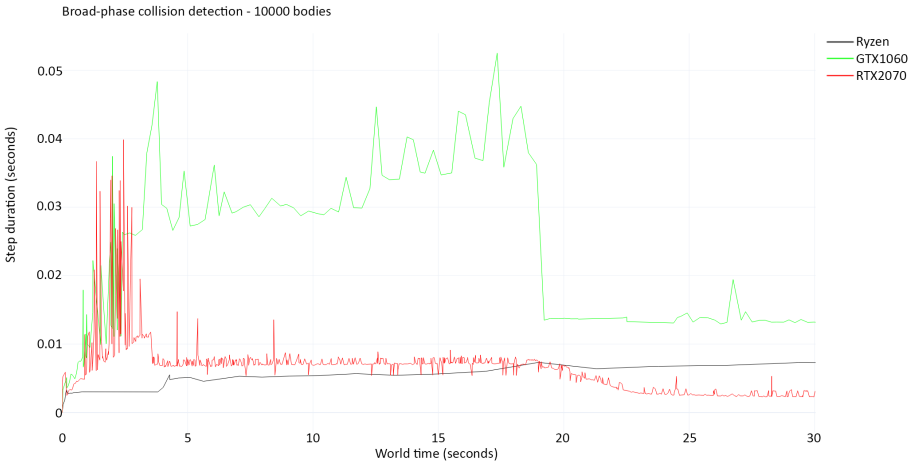

Figure 4.33: Spread bodies test broad-phase results with 10000 bodies.

For 100 and 1000 bodies, the CPU surpasses the GPU in performance, with the RTX 2070 again showing many spikes in execution time, especially in the 100 body test. As for the 10000 bodies test, only the RTX 2070 can surpass the CPU, keeping up with it up until the bodies finish spawning around the 20 seconds mark, after which the execution times drop below the CPU. The GTX 1060, on the other hand, while dropping execution times significantly once the bodies finish spawning, still performs moderately worse than the CPU or the RTX 2070.

Now, the results for the narrow-phase collision detection step can be seen in the figures 4.34, 4.35, and 4.36.

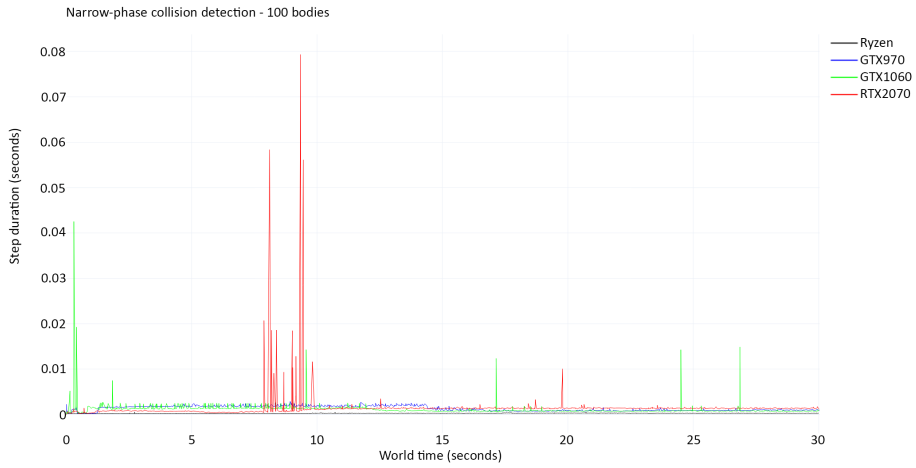

Figure 4.34: Spread bodies test narrow-phase results with 100 bodies.

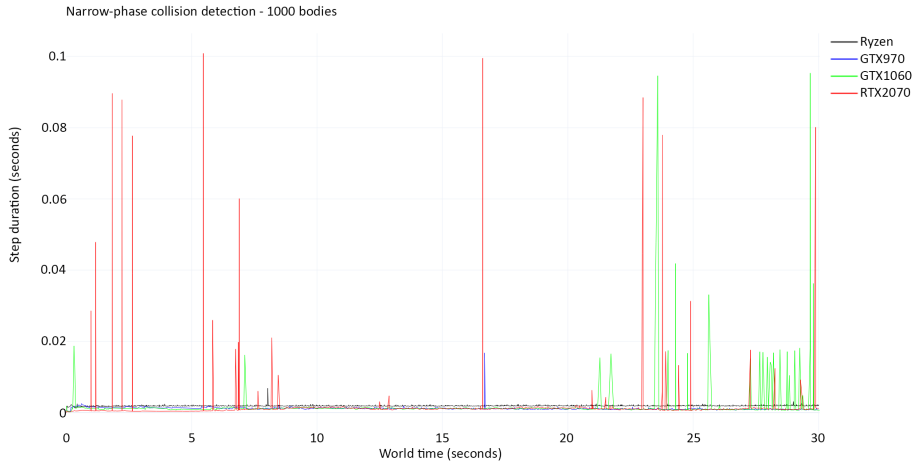

Figure 4.35: Spread bodies test narrow-phase results with 1000 bodies.

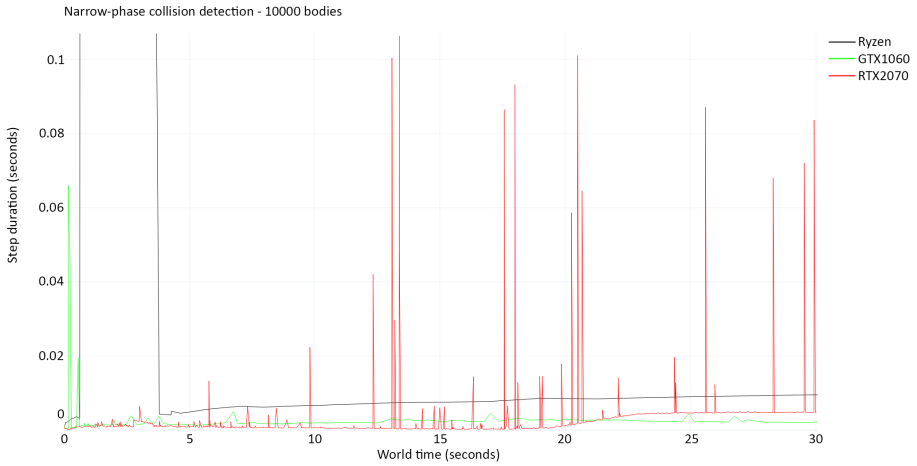

Figure 4.36: Spread bodies test narrow-phase results with 10000 bodies.

The CPU performs better in the 100 bodies test, while the GPUs defeat the CPU for 1000 and 10000 bodies, noting still some execution spikes by the RTX 2070 on all tests, and by the GTX 1060 at the end of the 1000 bodies test. It is interesting to note that the RTX 2070 dips behind the GTX 1060 in the 10000 bodies test from the 20 seconds mark onward, similarly to the narrow-phase test in the second environment with the same number of bodies.

Finally, the collision resolution step results can be seen in the figures 4.37, 4.38, and 4.39.

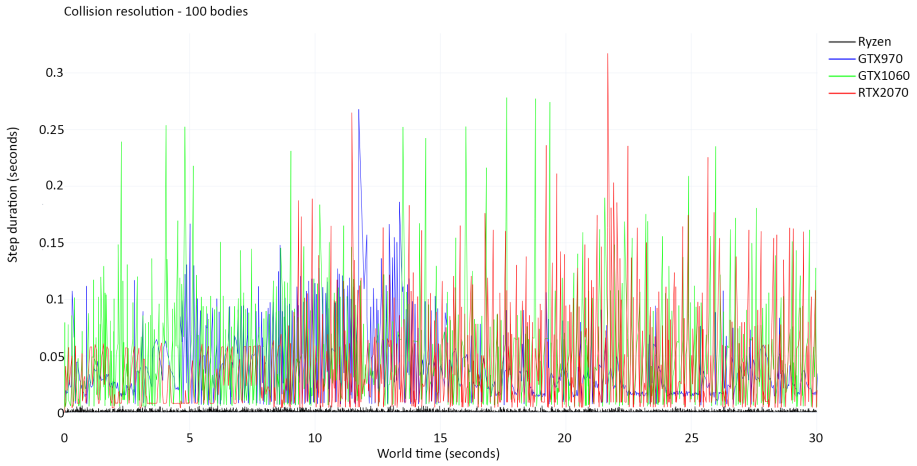

Figure 4.37: Spread bodies test collision resolution results with 100 bodies.

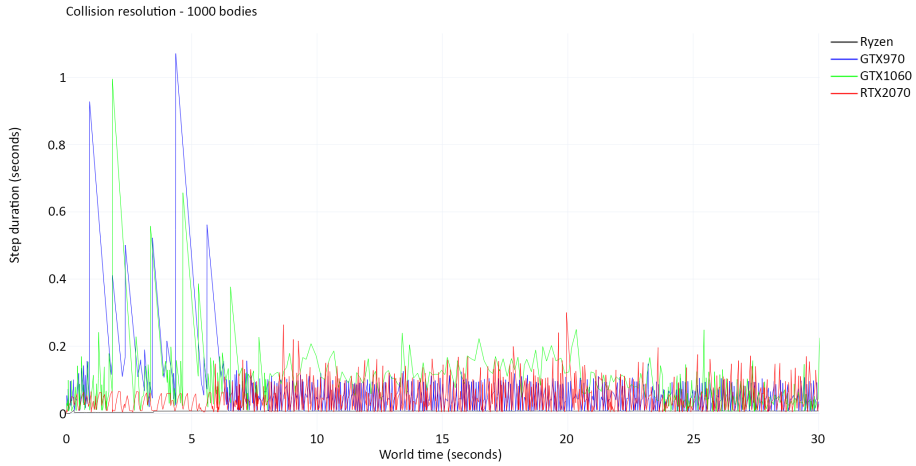

Figure 4.38: Spread bodies test collision resolution results with 1000 bodies.

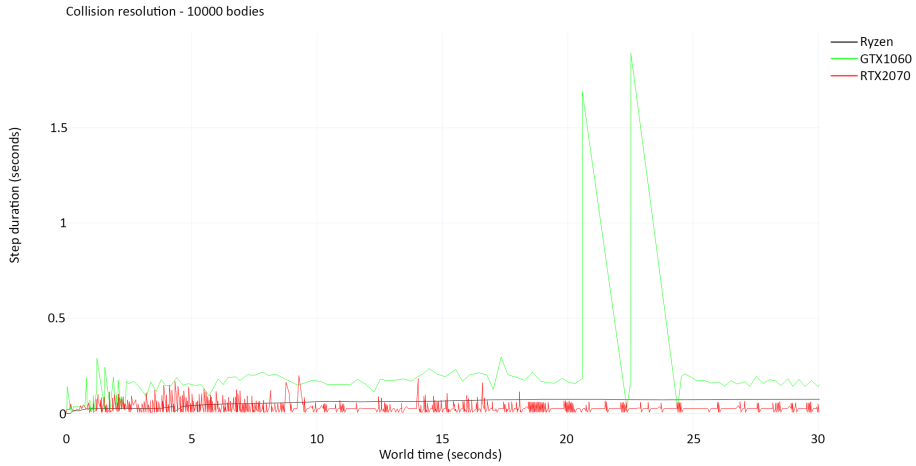

Figure 4.39: Spread bodies test collision resolution results with 10000 bodies.

For 100 and 1000 bodies, the CPU far surpasses the GPU implementation, with these showing several spikes in execution time, reaching 300 milliseconds in some situations, though the GTX 970 displays more consistent behavior and performance than the other two. With 10000 bodies, the RTX 2070 surpasses the CPU version, however still displaying occasional periods with spikes in execution time. The GTX 1060, on the other hand, performs worse than the CPU, despite being more consistent than the RTX 2070, with two exceptions during the test where it seemingly froze for a few seconds. It is worth noting that these freezes were reoccurring across multiple tests.

Similar to the second test, this environment also displays differing behavior between the CPU and GPU implementations. With any number of bodies, while in the CPU version, the bodies will remain still, in the GPU version, they will randomly start falling and penetrating the ground, bouncing back a few moments later when collision resolution forces the correction. One other problem that can be detected is that two adjacent bodies may sometimes start penetrating and,

instead of repelling each other, they will attract for a brief moment before violently repelling. Possible reasons for these behaviors are given in section 4.3.

4.2.4 Pyramid stack environment

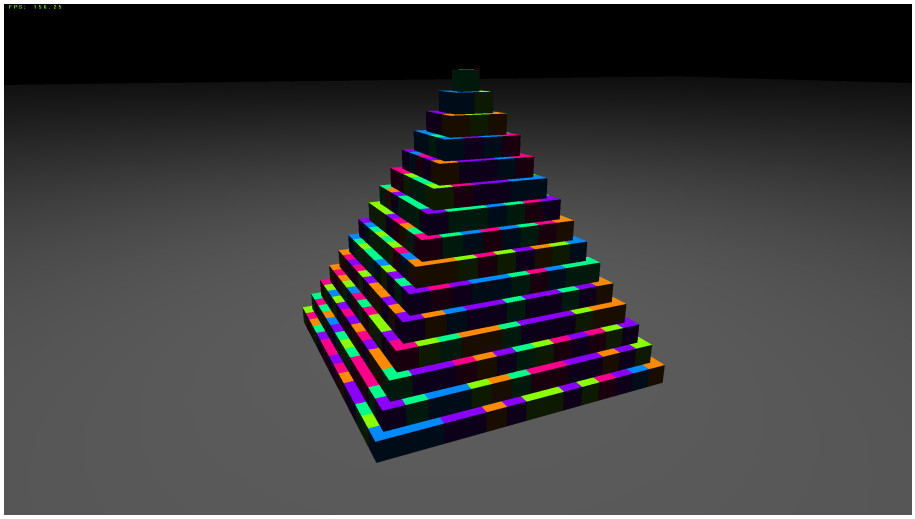

Figure 4.40: Pyramid stack test environment.

The fourth environment consists of a world with an immovable body similar to the previous two worlds, with the same dimensions at $z = 0$, and a stack of bodies arranged in a pyramid-like formation. This environment has a high number of collisions, with each body colliding with the same bodies around it, constantly over time. The broad-phase execution times should not vary as the bodies should remain static, but should be high due to the high amount of overlap caused by the position of the bodies relative to one another. On the other hand, the narrow-phase and collision resolution steps should have higher execution times due to the high number of collisions in each frame. This test also checks the stability of the simulation with respect to the accuracy of the collision solver, due to the high number of interdependent contacts between bodies.

The results with respect to the broad-phase collision detection step can be seen in the figures 4.41, 4.42, and 4.43.

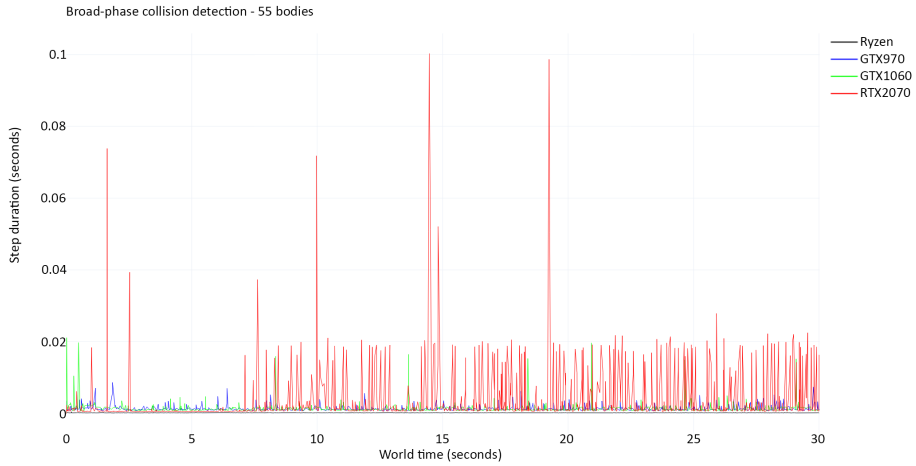

Figure 4.41: Pyramid stack test broad-phase results with 55 bodies (5 layers).

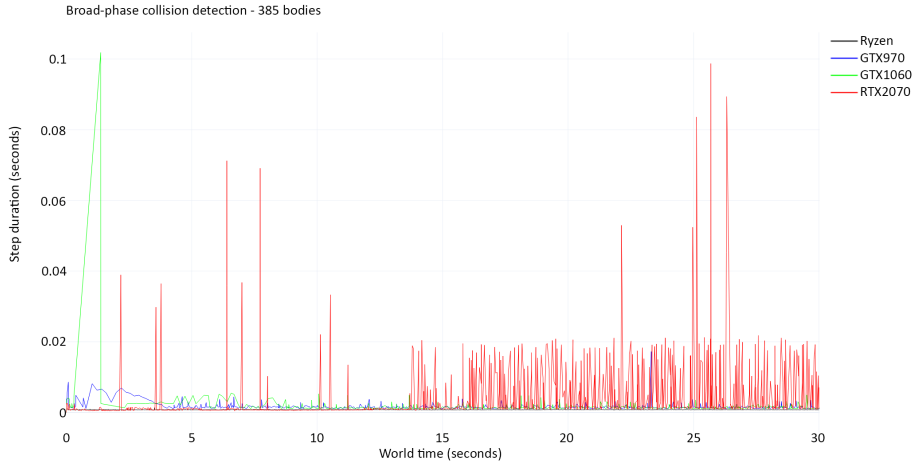

Figure 4.42: Pyramid stack test broad-phase results with 385 bodies (10 layers).

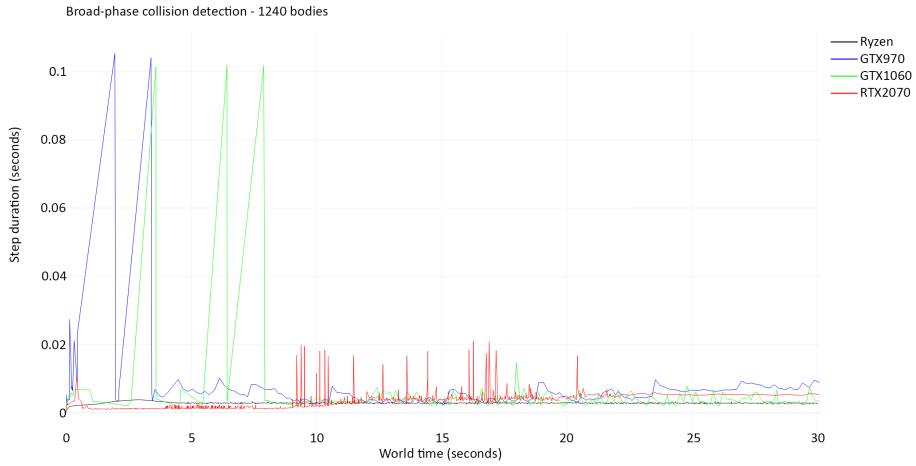

Figure 4.43: Pyramid stack test broad-phase results with 1240 bodies (15 layers).

For all tests, the CPU defeats the GPU version, with the GPU staying relatively close performance-wise to the CPU version, although the RTX 2070 continues to experience several spikes in execution, with these reducing in frequency and amplitude only on the 1240 bodies test.

Next, the narrow-phase collision detection step results can be seen in the figures 4.44, 4.45, and 4.46.

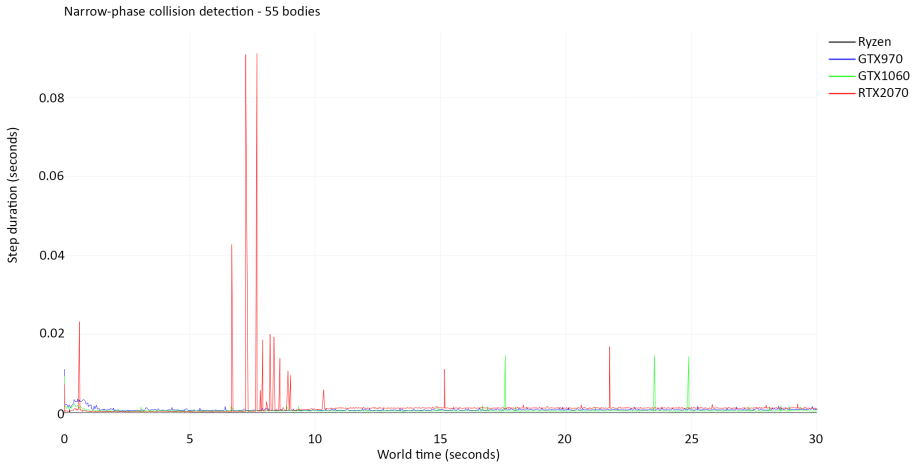

Figure 4.44: Pyramid stack test narrow-phase results with 55 bodies (5 layers).

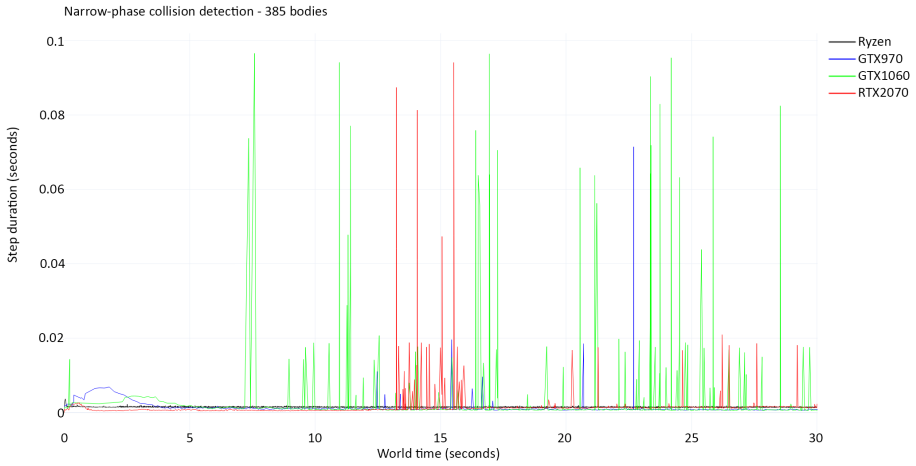

Figure 4.45: Pyramid stack test narrow-phase results with 385 bodies (10 layers).

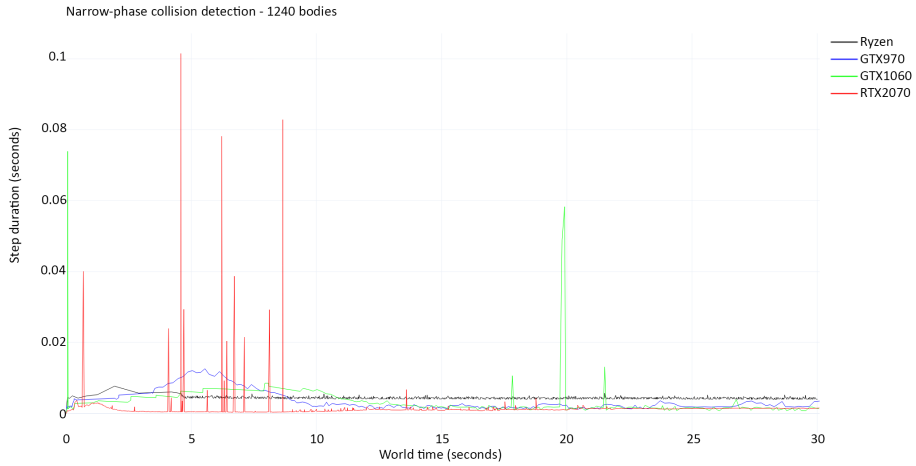

Figure 4.46: Pyramid stack test narrow-phase results with 1240 bodies (15 layers).

For 55 bodies, the CPU performs better than all GPUs, though these stay close behind the CPU, with no great difference between them, and only occasional spikes in execution. For both 385 and 1240 bodies, the GPUs are capable of surpassing the CPU in performance, especially in the latter case, though in the former test all graphics suffer occasional execution spikes.

At last, the collision resolution step results can be seen in the figures 4.47, 4.48, and 4.49.

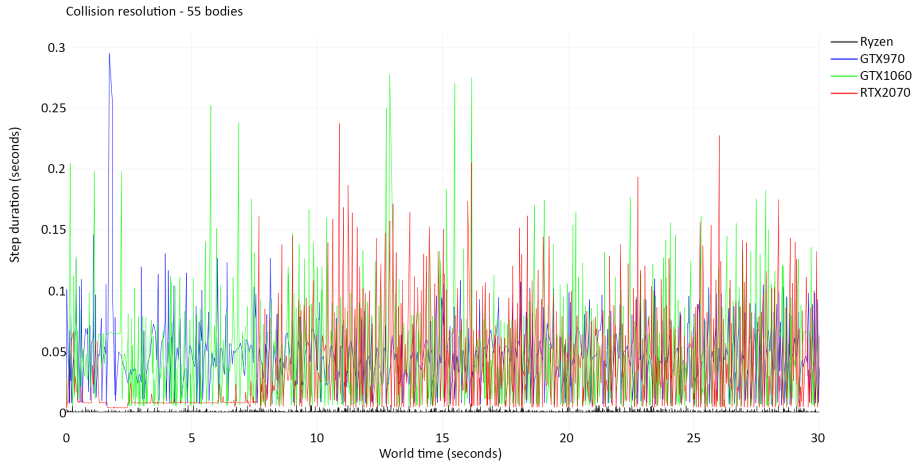

Figure 4.47: Pyramid stack test collision resolution results with 55 bodies (5 layers).

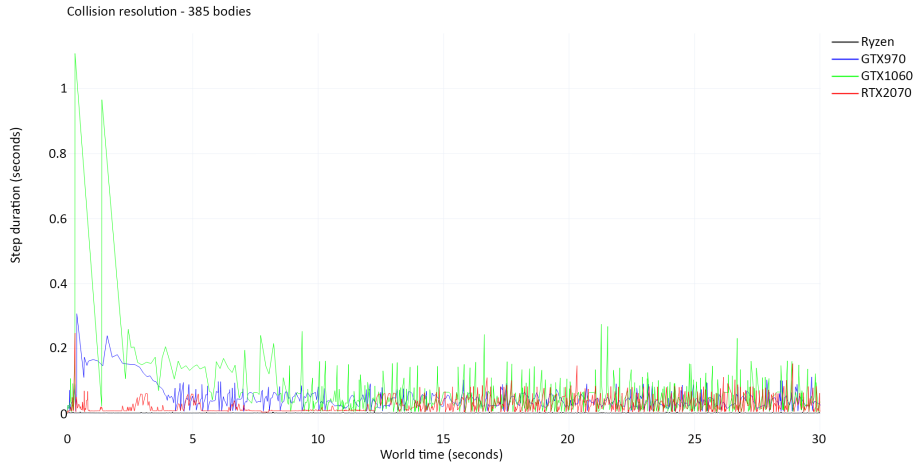

Figure 4.48: Pyramid stack test collision resolution results with 385 bodies (10 layers).

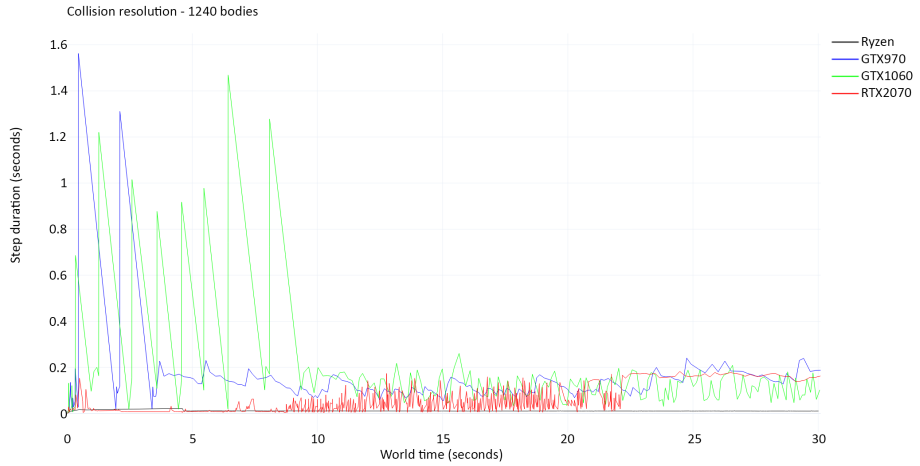

Figure 4.49: Pyramid stack test collision resolution results with 1240 bodies (15 layers).

Across all tests, no GPU can perform better than the CPU, with all GPUs showing heavy execution spikes in the 55 and 385 bodies test, while in the case of the 1240 bodies test, despite showing more consistent results, all perform far worse than the CPU. In the last test, while the RTX 2070 somewhat keeps up with the CPU for the first 20 seconds of execution, albeit with frequent execution spikes, from that point forward its performance dips significantly and stays consistently there, at roughly 175 milliseconds, with no apparent reason for doing so. Both the GTX 970 and the GTX 1060 show similar average performance in all tests. It should also be noted that on average, the GPUs perform better on the 55 and 385 bodies tests than on the 1240 bodies test.

Finally, it is worth noting that, similarly to the second test, the CPU version can maintain the structure of bodies, while in the GPU case, they will tend to randomly interpenetrate and then bounce off each other, destroying the structure.

To summarize the obtained results, the tables 4.2, 4.3, 4.4, and 4.5 present the average step duration of each device, with the best average of each step within a certain configuration marked in bold. All values are in seconds.

	Broad-phase	Narrow-phase	Collision resolution
Ryzen - 100 bodies	**0.000246**	**0.000058**	**0.000153**
GTX 970 - 100 bodies	0.001384	0.000362	0.050560
GTX 1060 - 100 bodies	0.001025	0.000312	0.044387
RTX 2070 - 100 bodies	0.004499	0.000306	0.032672
Ryzen - 1000 bodies	0.001681	**0.000056**	**0.000560**
GTX 970 - 1000 bodies	0.002391	0.000433	0.073422
GTX 1060 - 1000 bodies	**0.001085**	0.000246	0.049511
RTX 2070 - 1000 bodies	0.004457	0.000242	0.029814
Ryzen - 10000 bodies	0.017590	**0.000068**	**0.023008**
GTX 1060 - 10000 bodies	0.003649	0.006053	0.034333
RTX 2070 - 10000 bodies	**0.002537**	0.004846	0.027526

Table 4.2: Results comparison for random drifting environment.

	Broad-phase	Narrow-phase	Collision resolution
Ryzen - 100 bodies	**0.000245**	**0.000238**	**0.000957**
GTX 970 - 100 bodies	0.001307	0.001078	0.049294
GTX 1060 - 100 bodies	0.001295	0.000874	0.044654
RTX 2070 - 100 bodies	0.001990	0.001523	0.076552
Ryzen - 1000 bodies	**0.001702**	0.002242	**0.001626**
GTX 970 - 1000 bodies	0.002951	0.001143	0.066739
GTX 1060 - 1000 bodies	0.001929	**0.000670**	0.047458
RTX 2070 - 1000 bodies	0.002254	0.001030	0.035962
Ryzen - 10000 bodies	0.016786	0.015179	**0.020881**
GTX 1060 - 10000 bodies	0.022044	0.003340	0.145451
RTX 2070 - 10000 bodies	**0.006344**	**0.002059**	0.028928

Table 4.3: Results comparison for layers falling environment.

	Broad-phase	Narrow-phase	Collision resolution
Ryzen - 100 bodies	**0.000222**	**0.000249**	**0.000913**
GTX 970 - 100 bodies	0.001011	0.001232	0.036410
GTX 1060 - 100 bodies	0.001335	0.001084	0.049150
RTX 2070 - 100 bodies	0.001419	0.001083	0.035376
Ryzen - 1000 bodies	**0.001403**	0.001872	**0.006503**
GTX 970 - 1000 bodies	0.002517	0.001095	0.056704
GTX 1060 - 1000 bodies	0.002665	0.000975	0.075728
RTX 2070 - 1000 bodies	0.002016	**0.000893**	0.045172
Ryzen - 10000 bodies	**0.003809**	0.004567	**0.031709**
GTX 1060 - 10000 bodies	0.021737	0.002038	0.136037
RTX 2070 - 10000 bodies	0.006322	**0.001781**	0.032750

Table 4.4: Results comparison for spread bodies environment.

	Broad-phase	Narrow-phase	Collision resolution
Ryzen - 55 bodies	**0.000166**	**0.000243**	**0.000391**
GTX 970 - 55 bodies	0.001248	0.000896	0.047200
GTX 1060 - 55 bodies	0.001183	0.000636	0.044827
RTX 2070 - 55 bodies	0.000878	0.000915	0.026247
Ryzen - 385 bodies	**0.000766**	0.001480	**0.001350**
GTX 970 - 385 bodies	0.001447	0.000924	0.039375
GTX 1060 - 385 bodies	0.001409	**0.000780**	0.050702
RTX 2070 - 385 bodies	0.000832	0.000895	0.015920
Ryzen - 1240 bodies	0.002767	0.004342	**0.010591**
GTX 970 - 1240 bodies	0.004992	0.002393	0.116537
GTX 1060 - 1240 bodies	0.003748	0.002079	0.109537
RTX 2070 - 1240 bodies	**0.002470**	**0.000851**	0.023300

Table 4.5: Results comparison for pyramid stack environment.

4.3 Discussion

With respect to the collision detection step, it can be seen that, in the case of the first environment, the component that consumes almost all of the time is the broad-phase, as almost no collisions happen in each frame. Both GPUs tend to perform equally or better than the CPU in the broad-phase when the number of bodies is high enough and these move significantly. In contrast, in the third environment, where the broad-phase has comparatively little work to do, the CPU is capable of performing better than some GPUs, even at high body counts, though it is likely that even

higher body counts would change this tendency. Regarding the narrow-phase step, expectably, all GPUs perform worse than the CPU in the first environment, since no collisions turn this operation into a no-op on the CPU. However, across all other tests, the GPUs perform better at higher body counts, though with a much smaller requirement for the number of bodies than the broad-phase, with better results on the GPU being seen even at only hundreds of bodies. This is likely due to both the embarrassingly parallel nature of the algorithm, as well as to the comparatively higher arithmetic intensity of the narrow-phase relative to the broad-phase. Finally, looking at the collision resolution step, it can be seen that across all tests this is the most computationally expensive step of the physics simulation, regardless of the environment and characteristics of the hardware, with execution times significantly increasing when the number of bodies reaches the tens of thousands. Additionally, it can be determined that the GPUs can only match the CPU performance on tests with a very high number of bodies, in the order of tens of thousands, performing worse at any amount lower than that, regardless of the environment. Thus, this work was not able to match some of the results mentioned by the article from which the algorithm was adapted [14], though it is important to note that some crucial modifications were done to this implementation which may have negatively influenced the results, as described in section 3.4.

Looking now at some of the behaviors mentioned in the second, third, and fourth tests, one possible explanation that may arise is that the collision detection fails momentarily, causing the bodies to fall and penetrate the ground or each other, however, no such failure of collision detection was found during testing. Also, this explanation seems inconsistent with the attractive behavior occasionally exhibited by bodies. Alternatively, a more plausible explanation for both problems is that these are caused by the variance of the frame rate and hence variance of the Δt parameter which, when being passed to the collision solver, wrongly calculates the resulting impulses, compounded by the fact that the GPU simulation does not implement warm starting, as the implementations of the solver should otherwise be identical. This may also be aggravated by the chosen simulation scheme, where work is only submitted to the GPU periodically and is dependent on the frame rate of the whole application, instead of being an independent system that keeps the GPU busy with frequent blocks (using a fixed, independent Δt) of constant work, which could improve the consistency of the results.

It is relevant to note that the implementation sacrifices data locality in many cases for simplicity of implementation. Theoretically, this has a negative impact on performance due to the characteristics of the GPUs, relating to how they access memory (see 3.2). In practice, this only has a noticeable effect on the collision resolution step, as it can be verified through the varying step duration across several frames, likely during access to the constraints, as these are not sorted with respect to how they are accessed (their ordering should follow their spatial distribution in the world). It also affects, to a lesser extent, the narrow-phase step, which shows more variance of duration at higher body counts, likely due to access to the geometry of bodies, which exists in a global buffer, and not per collision instance. While some of these issues are solvable, such as constraint ordering, others are not so simple, namely the geometry distribution. There is no clear solution aside from copying geometry information to each potential collision, which would have

its drawbacks as the memory requirements for supporting the algorithm would grow significantly.

Aside from the varying step duration, several results from different tests show execution time spikes in GPUs, which can considerably affect a real-time simulation. These do not seem consistent with the problems regarding the behavior of the simulation, and as such remain an unsolved issue with the implementation, even after extensive debugging. It is possible that a more consistent workload being requested to the GPU over time could reduce or even erase these spikes, though another potential suspect for these problems might be the framework implementation itself. Being a black box, only allowing limited or no control over the scheduling of kernels and data movement, it is plausible that the SYCL runtime manages these aspects suboptimally with respect to the requirements of a real-time simulator.

Considering all aspects, it can be determined that there is potential for an effective simulation on the GPU, even if partial, as both phases of the collision detection step show promising results at high body counts. The collision resolution step would require some more analysis to determine whether the instabilities and any inefficiency in the simulation could be solved through some different algorithm and/or simulation scheme.

4.4 Summary

This chapter presented the results obtained from multiple tests on the developed prototype, beginning by listing the hardware used for the tests along with some of their specifications, briefly detailing the format and parameters of the tests, and then characterizing the performance of the simulator on the CPU when using various amounts of dedicated threads, referring to results obtained from one of the environments, and providing an optimal number of threads which was utilized for the remaining tests, along with the reasoning for its choice. Afterward, each environment was described and the results of all phases of simulation on each environment on the different devices were listed, with a short comparison of the results of each phase, detailing at key points some additional observations of behaviors of the bodies detected in some environments. Lastly, a discussion of the obtained results was provided, analyzing these and determining in which situations the GPU simulation was capable of performing better than its CPU counterpart. Particularly, both the broad-phase and narrow-phase collision detection showed better performance on the GPU in most or all environments when the number of bodies is high enough. At the same time, some of the behaviors mentioned during the description of the results were clarified, or a possible reason for their existence was provided when their source was unknown.

Chapter 5

Conclusion

Computational physics is a complex area of research with several sub-fields, with research conducted mainly focusing on high-precision simulation, such as in realistic animation and the authoring of physically accurate models in the manufacturing industry. The use of GPU devices and distributed systems to solve some of these problems is not new, with several articles describing such solutions. However, real-time physics simulation has been given comparatively less attention, with this particular sub-field remaining somewhat unexplored. Although there have been ventures into adapting real-time physics simulations for execution on the GPU, most of these see little adoption by applications at large, so these implementations tend to fall into disuse. By falling behind on technological advancements, these implementations lose opportunities for more efficient solutions to arise, from which new, innovative applications could surge.

This work presents a compelling reason for exploring this type of solution, grounding it on an existing application with real-world impact, and attempts to explore recent developments in GPGPU techniques to optimize real-time physics simulation, to not only improve the application but also to encourage discussion and research into this particular area of computation. By using an independent implementation based on established algorithms along with a new framework for heterogeneous computation, some positive results show that both this framework and this particular form of optimization of dynamical simulation have the potential for adoption in both new and existing applications, namely, the adopted broad-phase collision detection algorithm displays better performance on the GPU compared to the CPU when the bodies number in the tens of thousands, while the narrow-phase collision detection algorithm exhibits even better results, with the GPU implementation surpassing the CPU for environments with frequent collisions when the bodies number only in the order of the thousands. On the other hand, the results from the collision resolution algorithm on the GPU, while being on par with the CPU for tens of thousands of bodies, also display some undesirable behavior that affects the stability of the simulation, and thus further research is necessary to improve this subject.

5.1 Further work

Some considerations for future work consist of various optimizations to the physics simulation, such as implementing an effective contact caching scheme, which could also help with implementing warm starting to improve stability in the constraint solver.

Besides contact caching, implementing some form of body sleeping, which skips constraint solving and time integration for bodies with energy/velocity below a certain threshold, would be beneficial for performance, as it is a common optimization adopted by many real-time engines.

Additionally, researching ways into improving data locality on the SYCL simulator, such as adapting data structures to better support coalesced memory accesses, and ordering constraints before running the solver, could have a significant effect on the performance of these algorithms.

Alternatively, seeking different algorithms, especially for collision resolution, that could provide better performance and/or stability, could constitute a line of research. In particular, the parallelization of constraints is still an underexplored problem, and one of the main obstacles to capitalize more effectively on the parallelism of a simulation. Some of the problems seen with instability and execution times spikes, especially prevalent with the collision solver, would also require some research in order to determine their source.

Yet another option would be implementing a different simulation type: a scheme commonly known as asynchronous physics, where the simulation runs independently from the rest of the application and instead is simply provided with forces/torques and occasionally queried for the state of the world, keeping constant work on the GPU with fixed time steps, could help improve performance and stability through uniformity and persistence of work.

The possibility of a hybrid solution, using both the CPU and the GPU for the simulation, should also be considered, playing into each devices' strengths, *i.e.*, moving collision detection into the GPU and use those results for the collision resolution step, though some way to either remove the need for the frame delay on the engine or to map the results correctly to each frame would need to be developed.

Finally, one more solution that could improve the frame rate is removing the need to move the data back to the CPU by using shared buffers between the simulation and the renderer. Since the main reason for moving data back to the CPU after simulating is to prepare it for rendering, which requires moving it again to the GPU, removing this redundancy by simply processing the results in whatever form they exist in the buffers after simulation would likely reduce the total execution time of each frame, though this would require some form of communication between the SYCL framework and the rendering framework.

All these results will be taken into consideration when reviewing the Abyssal simulator, and at least part of this solution may potentially be adapted and integrated into the application, along with some of the ideas presented in this section, to improve the performance of the current simulation and remove some of the limitations imposed by the real-time constraints of the application.

Appendix A

Compound types description

- Vector – type composed of three floating-point numbers, x, y, and z, referred to as the components of the vector;
- Quaternion – type composed of four floating-point numbers, w, x, y, and z, referred to as the components of the quaternion, with w being the *real* component of the quaternion, and x, y, and z being the *imaginary* components of the quaternion;
- Matrix – type composed of arbitrary number of floating-point numbers, usually specified as $M \times N$ number of elements, capable of operations with vectors when specified as a 3×3 matrix;
- AABB – type composed of two vectors, *min* and *max*, referred to as the minimum and maximum of the AABB, respectively, or both as the extrema of the AABB;

Appendix B

Glossary of mathematical symbols

$\emptyset$ denotes the empty set;

$\in$ denotes set membership: $x \in A$ means that x is an element of A;

$\notin$ denotes set non-membership: $x \notin A$ means that x is not an element of A;

$\cup$ denotes set-theoretic union: $A \cup B$ is the set formed by all the elements of A and B;

$\cap$ denotes set-theoretic intersection: $A \cap B$ is the set formed only by the elements of both A and B;

$\neg$ denotes logical negation: $\neg A$ is true if A is false;

$\wedge$ denotes logical and: $A \wedge B$ is true if A and B are both true;

$\vee$ denotes logical or: $A \vee B$ is true if either A, B, or both, are true;

$\veebar$ denotes logical exclusive or: $A \veebar B$ is true if either A or B are true;

$\forall$ denotes universal quantification: $\forall x A$ means that A is true for all possible values of x;

$\exists$ denotes existential quantification: $\exists x A$ means there exists at least one value of x for which A is true;

$\rightarrow$ denotes material conditional: $A \rightarrow B$ means that if A is true, then B is also true;

$\leftrightarrow$ denotes material biconditional: $A \leftrightarrow B$ means that if A is true, then B is also true, but if A is false, then B is also false.

References

[1] AMD APP SDK OpenCL User Guide. Technical report, 2015.

[2] AMD "Vega" Instruction Set Architecture Reference Guide. Technical report, 2017.

[3] GPU Rigid Bodies — NVIDIA PhysX SDK 4.0 Documentation, 2018.

[4] CUDA C++ Programming Guide. Technical report, aug 2020.

[5] David Baraff. An Introduction to Physically Based Modeling: Rigid Body Simulation I-Unconstrained Rigid Body Dynamics Rigid Body Simulation. *IN AN INTRODUCTION TO PHYSICALLY BASED MODELLING, SIGGRAPH '97 COURSE NOTES*, page 32, 1997.

[6] David Baraff. An Introduction to Physically Based Modeling: Rigid Body Simulation II-Nonpenetration Constraints. *IN AN INTRODUCTION TO PHYSICALLY BASED MODELLING, SIGGRAPH '97 COURSE NOTES*, page 38, 1997.

[7] João Bispo, Jorge G Barbosa, Pedro Filipe Silva, Cristian Morales, Mirko Myllykoski, Pedro Ojeda-May, Milosz Bialczak, Mariusz Uchronski, Adam Wlodarczyk, Peter Wauligmann, et al. Prace best practice guide 2021: Modern accelerators. Technical report, PRACE aisbl, 2021.

[8] Ian Buck and Tim Purcell. Chapter 37. A Toolkit for Computation on GPUs | NVIDIA Developer, 2004.

[9] Daniel Chappuis. Constraints Derivation for Rigid Body Simulation in 3D. Technical report, 2013.

[10] Erwin Coumans. *Bullet - GPU rigid-body simulation using OpenCL.* 2015.

[11] Erwin Coumans. *Bullet User Manual*. 2015.

[12] J. Darlington, M. Ghanem, and H. W. To. *Structured parallel programming*. 2012.

[13] Dirk Gregorius. Robust Contact Creation for Physics Simulations, 2015.

[14] Takahiro Harada. A parallel constraint solver for a rigid body simulation. In *SIGGRAPH Asia 2011 Sketches, SA'11*, page 1, New York, New York, USA, 2011. ACM Press.

[15] Hans Christian Hege and Hinnerk Stüben. Vectorization and parallelization of irregular problems via graph coloring. *Proceedings of the International Conference on Supercomputing*, pages 47–56, jun 1991.

[16] Ben Kenwright and Graham Morgan. Practical introduction to rigid body linear complementary problem (LCP) constraint solvers. In *Algorithmic and Architectural Gaming Design: Implementation and Development*, pages 159–201. IGI Global, 2012.

[17] Ronan Keryell, Maria Rovatsou, and Lee Howes. SYCL™ Specification. Technical report, 2020.

[18] Scott Le Grand. Chapter 32. Broad-Phase Collision Detection with CUDA | NVIDIA Developer, 2007.

[19] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. Real-time collision culling of a million bodies on graphics processing units. In *ACM SIGGRAPH Asia 2010 papers on - SIGGRAPH ASIA '10*, volume 29, page 1, New York, New York, USA, 2010. ACM Press.

[20] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[21] Pierre Terdiman. Sweep-and-prune. Technical report, aug 2007.

[22] Richard Tonge, Feodor Benevolenski, and Andrey Voroshilov. Mass splitting for jitter-free parallel rigid body simulation. *ACM Transactions on Graphics*, 31(4):1–8, aug 2012.