Hochschule
Augsburg University of
Applied Sciences

Bachelorarbeit

Fakultät für
Informatik

Studienrichtung
Informatik

Simon Kerler

Generische N-Körper Simulationsengine mit GPU-Unterstützung

Prüfer: Prof. Dr. Peter Rösch
Abgabe der Arbeit am: 19.01.2015

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Bachelorarbeit:
Simon Kerler
Haupstraße 4
86877 Walkertshofen
simon_kerler@web.de

Zusammenfassung

Die Simulation und Visualisierung von N-Körper Systemen findet sowohl in der Forschung als auch in der Computergrafik häufig Anwendung. Neueste Entwicklungen im Bereich der Grafikhardware erlauben es, diese einfach zu parallelisierenden Simulationen auf der GPU auszuführen. Viele der verfügbaren Lösungen wurden jedoch für Spezialfälle entwickelt und bieten deshalb wenig Flexibilität bei der Definition von Partikelattributen und -verhalten. Zudem werden sie häufig mit Hilfe der CPU oder veralteten GPGPU-Techniken berechnet.

Diese Bachelorarbeit stellt eine Simulationsengine vor, die diese Nachteile vermeidet. Sie wurde in C++ geschrieben und verwendet OpenGL 4.3 als Render- und modernes GPGPU-Framework. Ihre objektorientierte und generische API ermöglicht es, N-Körper Systeme jeglicher Art zu modellieren. Hierbei werden Partikelattribute in GPU-Puffern gespeichert während ihr Verhalten mit Compute Shadern definiert wird. Die Engine wurde so konzipiert, dass Unabhängigkeit von Plattform und Hersteller, Erweiterbarkeit, Flexibilität und Benutzerfreundlichkeit gewährleistet ist. Quellcode für Shader kann wiederverwendet werden, um Code-Duplikationen zu vermeiden. Des Weiteren verwaltet die Engine all ihre Ressourcen implizit.

Für den Entwurf der Engine wurden verschiedene GPU-Architekturen und GPGPU Frameworks analysiert. Zudem wurden zwei moderne Spieleengines (Irrlicht3D, Ogre3D) und mehrere Partikel- und N-Körper Simulationen untersucht, um die Struktur der Engine zu entwerfen.

Um die Korrektheit der Implementierung sicherzustellen, wurden zwei Gravitationssimulationen erstellt. Zusätzlich wurden zwei Benchmarks geschrieben, um die Performanz des Render- und Update-Prozesses zu messen. Diese wurden auf einer NVIDIA GeForce GTX660, GeForce GTX780 Ti und Quadro 4000 ausgeführt. Durch Analyse der Ergebnisse konnten mögliche Performanzengpässe aufgezeigt und Optimierungen der Compute Shader demonstriert werden.

Das Ergebnis dieser Bachelorarbeit ist eine funktionsfähige N-Körper Simulationsengine mit GPU-Unterstützung, die alle ursprünglichen Anforderungen erfüllt. Lediglich die Benutzerfreundlichkeit kann im Moment nicht objektiv beurteilt werden.

Abstract

Simulation and visualisation of n-body systems have many use cases in scientific research and computer graphics. Recent developments of graphics hardware allow these simulations to be executed on the GPU due to the parallel nature of the n-body problem. Many of the available solutions are hard coded to a special purpose and do not grant much flexibility for the definition of particle attributes and behaviour. Most often, updating a simulation is done on CPU or by using deprecated ways of GPGPU.

This thesis proposes a simulation engine which negotiates these shortcomings. It is written in C++ using OpenGL 4.3 as render and modern GPGPU framework. Its object-oriented and generic API allows the modelling of all kinds of n-body systems with particle attributes stored in GPU buffers and behaviour encoded in compute shaders. The engine was designed to be cross-platform, independent of hardware vendors, extensible, flexible and easy to use. Shader source code can be reused and all resources are managed by the engine implicitly.

For the creation of the engine, different GPU architectures and GPGPU frameworks were analysed. Two modern game engines (Irrlicht3D, Ogre3D) and various particle and n-body simulations were examined to design the architecture of the engine.

To verify implementation, two gravitational simulations were created. Two benchmarks to profile the render and compute process were implemented, as well. They were executed on a NVIDIA GeForce GTX660, GeForce GTX780 Ti and Quadro 4000 with results being examined to show possible bottlenecks and compute shader optimisations.

The result of this thesis is a working n-body simulation engine with GPU support that fulfils all of the initial requirements. Only the usability cannot be assessed objectively at the moment.

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

.....

Augsburg, den 19.01.2015

Contents

List of Figures

List of Tables

Listings

1. Introduction

Simulating physical processes and rendering the results on screen is a common task for both, scientific research and entertainment media like computer games. To model complex systems, for example fog, fluids or galaxies, they are represented by a set of *particles* with a number of attributes and behaviours [2]. If particles can interact with each other, the term *n-body system* is used [3].

Use cases for n-body simulations are fluid dynamics [4], [5], [6], plasma [7], [8] and astronomical simulations [9], [10]. They are also useful for prototyping in industries for example to examine soiling of cars [11] or to analyse bottlenecks of escape routes of a building in case of a panic [12].

Since all particle-particle interactions have to be calculated when an n-body system is updated, the complexity of these simulations is $O(n^2)$ which leads to a quadratic runtime as particle count increases [13]. Different methods of improvement have been developed, for example the Barnes-Hut algorithm with a complexity of $O(n \log n)$ [14].

On the other hand, recent developments of graphics cards enabled them to be used not only for graphical applications but also for general purpose computing (*GPGPU*). This makes it possible to use the highly parallel compute cores of the GPU to update n-body systems since these can be easily parallelised. This theoretically reduces the runtime of the best case to $O(n)$ if n cores are used.

Many of the available simulations (see section 1.2) either restrict the particles to certain attributes or have a fixed way of how particles behave. To model and simulate all kinds of systems, a generic way to describe and update them is necessary. Some engines provide a way to model the behaviour, but updating is performed on the CPU or by using a deprecated way of GPGPU that exploits the rendering pipeline to perform computations. In some cases, particle-particle interactions are not possible and so these engines cannot be used to simulate n-body systems.

1.1. Objectives of the Thesis

To overcome all of these drawbacks mentioned above, a generic n-body simulation engine using a modern GPGPU framework has to be created which is addressed by this thesis.

First of all, the state of the art has to be evaluated on which the implementation of the engine is based. Besides related work covered in section 1.2, this also includes detailed understanding of the architecture of graphics chips and the basics of GPGPU which is why they have to be introduced. Different frameworks could be used for this thesis: OpenCL, CUDA, DirectCompute and OpenGL 4.3. These have to be compared to each other based on the requirements imposed on the engine (see below) to choose the

most appropriate one. The usage of the selected tool has to be examined to understand how it can be integrated into the engine.

The engine must grant possibilities to model any kind of n-body system with flexible particle attributes and update routines for scientific purposes or to create visual effects. Updating of particles must be executed on the graphics card to benefit from its parallel processing power.

It must be possible to render the current state of the simulated system interactively using a 3D rendering framework like Direct3D or OpenGL. Since this thesis mainly focuses on GPGPU, the basics of 3D graphics and of the selected rendering framework cannot be covered because they are beyond its scope.

In addition, several requirements had been taken into account when the engine was designed. The most important ones are described in the following.

Vendor independency and cross-platform. The engine has to be designed vendor independent and cross-platform to address as many target systems as possible. This has to be kept in mind for the choice of render and GPGPU framework as well as for third party libraries.

Reusable GPU code. It must be possible to reuse code that is executed on the graphics cards to avoid code duplication. This allows common functionality to be shared between GPU programs and applications.

Flexibility and extensibility. To be able to create advanced simulations and visual effects, the engine must not restrict its potentials. This means that, although high level functions are used most of the time, access to the low level API has to be exposed to the end user as well. Despite abstracting the view on the used frameworks, directly calling their functions should be possible since not all features can be abstracted by the engine. This introduces additional flexibility and the engine can be extended easily to adapt the needs of a certain simulation.

Easy to learn and to use. The engine must provide an easy to learn and easy to use API so newcomers are able to use it, as well. Creating a new simulation should be possible with few function calls. To give visual feedback as soon as possible, standard implementations for all mandatory resources should be available.

Object-oriented API. Since object-oriented APIs are state of the art in terms of game engines [15], the created solution should use the OOP paradigm as well.

Implicit resource management. The user should not be bothered with the management of resources created by the engine. Instead, they should be destructed implicitly when they are no longer in use.

1.2. Related Work

There are many publications and software available related to the topic of this thesis: Game engines provide a generic architecture, work covering particle systems give many basics used for these simulations and publications covering n-body simulations in particular are a good source for special cases and optimisations.

This section provides a general overview of work taken into account for the design and implementation of the simulation engine.

Game Engines

Some of the most powerful 3D rendering systems are part of game engines. As video games become more and more realistic, not only better hardware is required but also the performance of the software has to be optimised. They are also a good source for generic software architecture since games of all different kinds can be created with the same engine.

For this thesis, two representative open source engines have been examined: Irrlicht3D [16] and Ogre3D [17]. Both are based on similar design principles but differ in their implementation. Many concepts of the engine use these as a guideline. When the simulation engine is discussed in chapter 4, parallels between its architecture, Irrlicht3D and Ogre3D are drawn where necessary.

Particle Systems

In 1983, Reeves introduced the concept of "particle systems - a method for modeling fuzzy objects such as fire, clouds, and water"[2]. N-body systems can be considered as special types of particle systems in which their elements interact with each other [3] and the number of particles is constant in most cases, i.e. they do not have a birth-death lifecycle. Therefore, many of the techniques and architectural design patterns as well as algorithms for movement integration or collision detection and response introduced for particle systems can be applied to n-body systems. This is also the reason why this thesis uses the terms "n-body" and "particle" interchangeably.

The game engines Irrlicht3D and Ogre3D contain CPU based particle systems which can be added to the scene graph. The system is modelled by using *affectors* that influence a particle, e.g. attract them or change their colour.

In 2004, two GPU based particle systems have been proposed by Kipfer et al. (the *UberFlow* engine) [18] and Lutz [19] respectively. Both store attributes in textures and use fragment shading to update the particles. Methods for particle motion, collision detection, collision response and sorting by distance are addressed as well.

McAllister published an OpenGL-style API for particle systems in 2000 [20] which inspired many other solutions like the two mentioned above. Particles have the same attributes that have been proposed by Reeves and are modified with *actions* which are building blocks to model their behaviour. They can be chained in a so called *action list* which is executed in a way comparable to multi-pass rendering to model complex

particle effects. Neither attributes nor actions can be customised which is stated as future work. Porting the API to GPU is also suggested future work.

A different approach was created by Krajcevski and Reppy in 2011 [21]. They introduced a declarative language to encode particle attributes and behaviour as well as the rendering of the particle system. This description is translated into an internal representation which can be interpreted by the CPU or further compiled to OpenCL or GLSL code that uses fragment shading. In contrast to the UberFlow engine and to Lutz's solution, they do not support particle-particle interactions.

In [22], a gravitational particle system implemented with Direct3D is described. It uses new features introduced with DirectCompute to achieve a simulation solely based on the graphics card without the need to exchange data between CPU and GPU.

The OpenGL Programming Guide presents a particle system using compute shaders [1, ch. 11] whereas the OpenGL Cookbook shows a transform feedback based implementation [23, ch. 8].

In [24], another OpenGL compute shader based particle system is introduced which is compared to an OpenCL implementation.

NVIDIA GameWorks™ have released some open source samples which demonstrate various visual effects implemented with OpenGL [25]. Some compute shader based examples are provided one of which is a particle simulation.

N-Body Simulations

The third type of useful sources are papers that cover n-body simulations in particular. In the majority of cases, these are hard coded examples focused on a certain topic or framework.

In [26], the capabilities of OpenGL compute shaders are evaluated. One of the included examples demonstrates the implementation of an n-body simulation.

A good source for the implementation and optimisation of a brute-force gravitational n-body simulation is presented in [13]. This implementation is based on CUDA but shows concepts that can be mapped to other frameworks.

Another brute force method named "force splatting" is outlined in [3]. This technique uses multiple rendering passes to calculate the influence of one particle on all other particles per pass. By using additive alpha blending, the new results are accumulated with the forces computed in previous rendering calls. After all passes completed, each texture pixel stores the force that applies on one of the particles.

Besides force splatting, [3] also presents a method how particles can interact with their environment, e.g. bounce off meshes placed in the scene, and, vice versa, how the environment can react to particles, e.g. colour a mesh when it is hit by a particle.

1.3. Structure of the Thesis

Chapter 2 gives an introduction to GPGPU by discussing the architecture of modern graphics cards as well as frameworks that allow to use them for general purpose computing.

Since OpenGL was chosen for the engine of this thesis, chapter 3 describes how its compute shaders are used.

Chapter 4 covers the simulation engine in particular. The different abstraction layers and components are described and advanced functionality is introduced.

To demonstrate how the engine can be used, chapter 5 presents two gravitational n-body simulations as examples. The first one uses randomly distributed particles and demonstrates how shared memory can be used for optimisation. The second one simulates our solar system as an example of a scientific use case producing quantitative results.

In chapter 6, the performance of the render and update processes of the simulation engine are evaluated. The optimisation technique applied to the first gravitational simulation is profiled, as well.

Chapter 7 concludes the thesis by summarising the achieved results and proposing future work.

2. Introduction to GPGPU

In this chapter, an introduction to GPGPU is given. At first, a deeper look at the architecture of modern graphics hardware is taken. Afterwards, GPGPU frameworks are discussed by describing their basic concepts and outlining pros and cons of available solutions.

2.1. GPU Architectures

Developing massively parallel programs for devices like graphics cards requires detailed understanding of the underlying hardware architecture. Especially optimisations regarding memory usage make the programmer think about how to optimally use caches or shared memory.

The earliest GPUs implemented fixed functionality which was aimed at graphics processing only. They provided functions for geometry transformations and special operations like texturing or lighting. New developments enabled programmability of previously fixed vertex and pixel manipulation stages leading to the introduction of so called shaders. These gave developers much more flexibility in their applications and also the ability to use GPUs for general purpose calculations by using textures as input and frame buffer rendering or transform feedback as output of their algorithms (see also section 2.2.2)[27]. The next generation of graphics processors went even further and reduced fixed functionality to special cases like tessellation, viewport transformations etc. [28], [29], [30]. Instead, highly programmable compute cores are used for graphics processing as well as general purpose computations [27].

The next two sections give an overview of NVIDIA and AMD GPU architectures. Since the software for this thesis was created on a *NVIDIA GeForce GTX 660*, this architecture is mainly focused on.

2.1.1. NVIDIA CUDA Architecture

First Generations of CUDA

In 2006, NVIDIA released the *GeForce 8800* which was based on their new GPU architecture called *G80*. This first generation of *CUDA* (*Compute Unified Device Architecture*) removed the separate pixel and vertex pipelines and replaced them by so called *streaming multiprocessors* (*SM*). Each SM consists of multiple cores called *CUDA cores* which provide a uniform basis for vertex, pixel and compute programs. Multiple threads can run concurrently on one SM. During each clock cycle, one instruction is decoded

and executed on all of these threads. This kind of execution model is called *SIMT* (*single-instruction multiple-thread*) [31, Chapter 4.1].

Aside from that, *shared memory* and *barrier synchronisation* techniques are used for inter-thread communication on one SM. [32, p. 4]

Introduced in 2008, *GT200* (codename *Tesla*), the second generation of CUDA, focused on performance improvements and additional features like double precision floating point support which is required for scientific computations [32, p. 4].

Fermi

A major redesign of CUDA was done with the introduction of *Fermi* in 2009. Some of the key concepts have been the improvement of inter-thread communication (new cache layout, faster atomic operations), introduction of tessellation and a new streaming multiprocessor. The full set of changes and improvements can be found in [28] and [32].

Figure B.1 shows the block diagram of the Fermi based *GF100* whereas Figure B.2 depicts the architecture of a Fermi SM. Since naming and functionality basically stayed the same, a description of components is given in the next paragraph.

Kepler

Kepler is the code name of the architecture released by NVIDIA in 2012 which is based on Fermi. Since the GeForce GTX 660 belongs to the Kepler family of *GK104* GPUs, a deeper look into this architecture is taken hereafter.

As depicted in Figure 2.1, the GPU consists of several main components:

- The *Host Interface* communicates with the system via PCI-Express 3.0 [32, p. 7][29, p. 5] by reading CPU commands [28, p. 11].
- The *GigaThread Engine* copies data from host memory to the frame buffer [28, p. 11] and dispatches work (grouped as thread blocks) to SM units. [28, p. 11], [32, p. 7]. It also handles redistribution of work e.g. after tessellation or rasterisation [28, p.11] and when *Dynamic Parallelism* [33, p. 5] is used.
- There are four *Memory Controller*s which are used to access DRAM. Each of them is combined with a 128kB L2 cache (512kB in total) and eight ROP units (see below) [29, p. 12].
- A *Raster Operation Unit* (*ROP unit*) [28, p. 4] is used for pixel based operations like blending and antialiasing but also features atomic operations for memory access [28, p. 12]. These are depicted above and below the L2 cache in Figure 2.1.
- The GPU is further subdivided into four *Graphics Processing Clusters* (*GPC*s). Apart from ROP units, a GPC contains all logic required for graphics processing [28, p. 12][29, p. 6]. To accomplish this task, they contain a *RasterEngine* and two streaming multiprocessors.

- The *Raster Engine* is part of the GPC. As name indicates, it is used for the rasterisation step in the graphics pipeline, including back face culling and depth tests [28, p. 14].

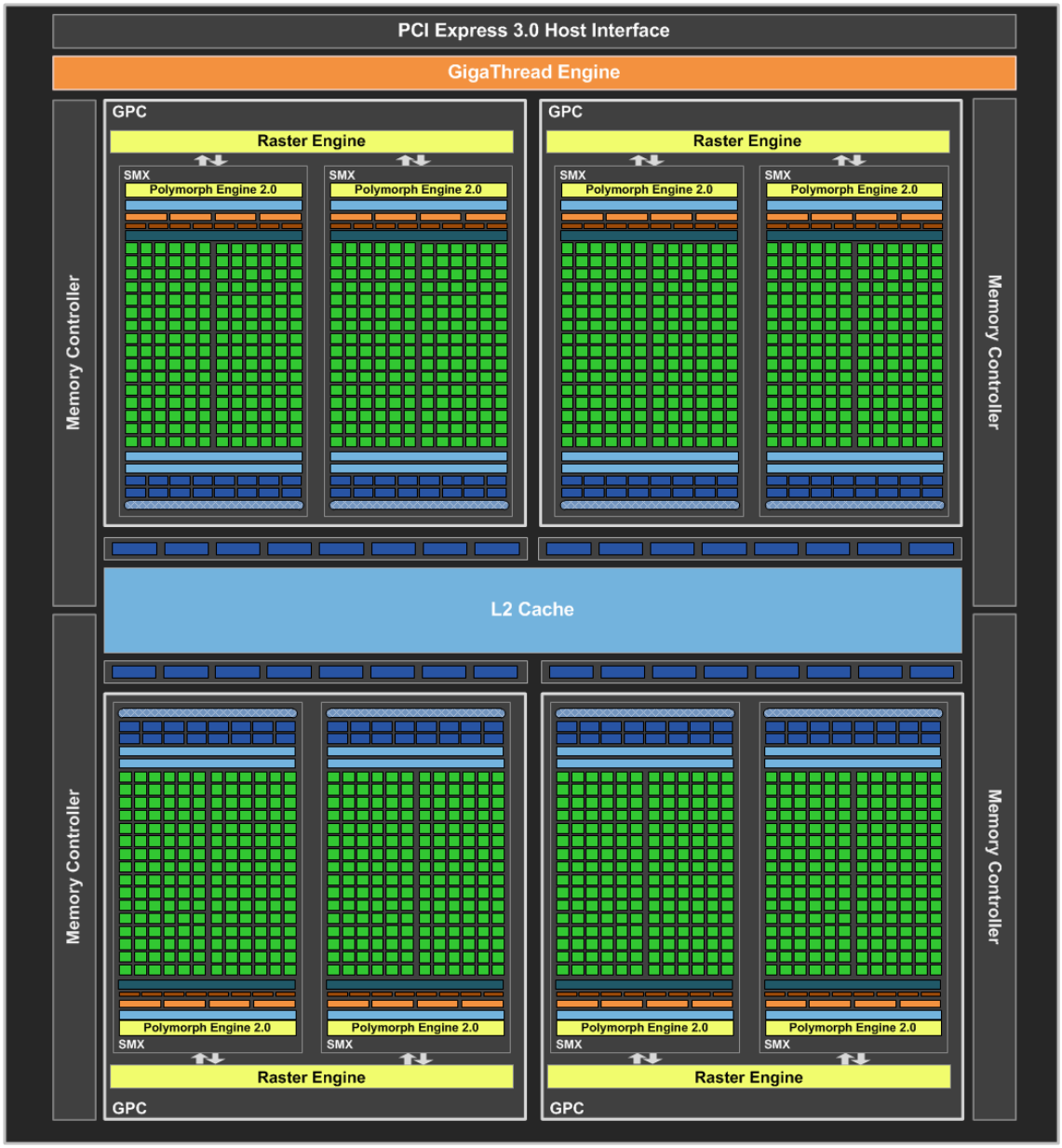


Figure 2.1.: NVIDIA GeForce GTX680 block diagram [29]

As explained previously, each GPC features two streaming multiprocessors which are called *SMX* in Kepler architecture. Each SMX executes a number of so called *warps* which are groups of 32 threads [33, p. 13 et seq.]. There can be a maximum of 64 active warps per multiprocessor [33, p. 7].

The components of the SMX are shown in Figure 2.2 and are described below.

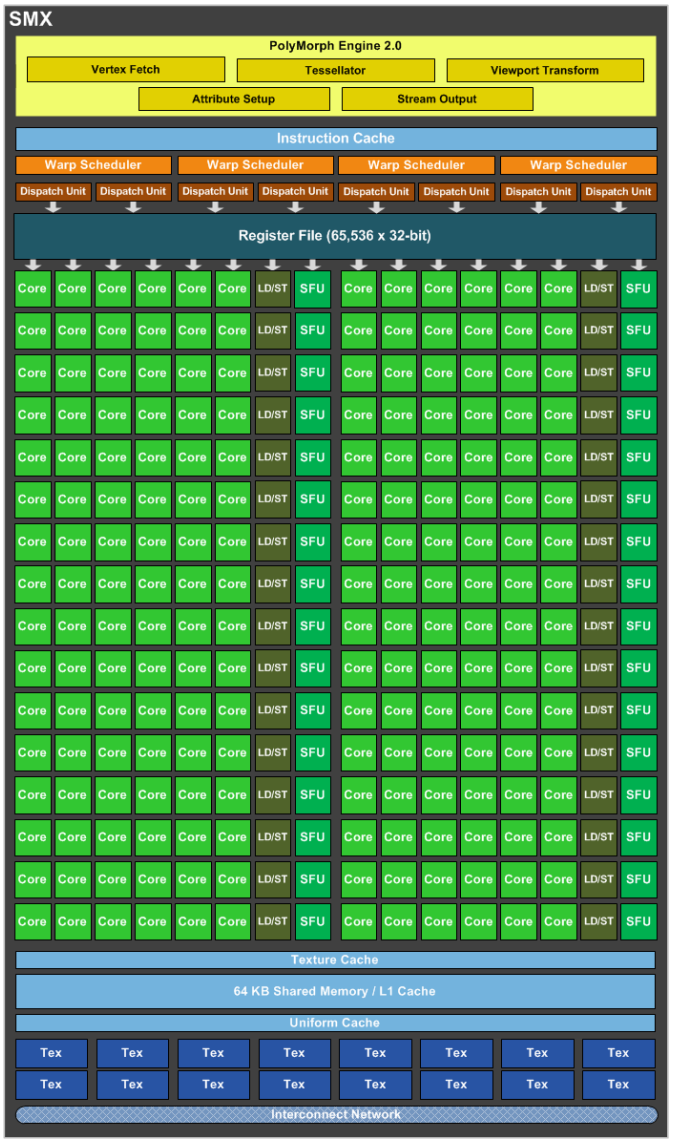


Figure 2.2.: NVIDIA GeForce GTX680 SMX block diagram [29]

- A SMX contains four *Warp Schedulers* with two *Dispatch Units* each. This structure enables scheduling of two independent instructions per warp and per clock cycle, e.g. a floating point operation and a texture fetch. All schedulers inside a multiprocessor share one *Instruction Cache*.
- There is one *PolyMorph Engine* per SMX [28, p. 13 et seq.] which contains five of the fixed functionality stages of the graphics processing pipeline like vertex fetch or tessellation. After each of these steps, the result is passed on to the *CUDA*

cores which execute the appropriate shader (e.g. vertex shader after vertex fetch). The outcome is handed back to the PolyMorph Engine which continues execution until the pipeline has been fully processed. The overall result is then passed to the RasterEngine of the dedicated GPC.

- Each SMX has 192 *CUDA Cores* whose structure is depicted in Figure 2.3. Each core contains a fully pipelined integer and floating point unit [28, p. 16] which enables it to execute one floating point or integer operation per clock cycle [32, p. 7].
- Beside the CUDA cores, there are 32 *Special Function Units* (*SFUs*) per SMX which are used to compute functions like sine, cosine or square root [32, p. 9]. While a SFU is busy, other work can be scheduled and performed in parallel.
- 32 *Load- / Store Units* are used to calculate memory addresses and grant read / write access to cache or DRAM [32, p. 8].
- All SMX feature 16 *Texture Units* to fetch filtered or unfiltered texel data from textures [28, p. 17 et seq.]. Performance is increased by using a dedicated *L1 Texture Cache* and global L2 cache.
- There is 64kB of on-chip memory which can be divided into *L1 Cache* and *Shared memory* by segments of 16kB + 48kB, 32kB + 32kB and 48kB + 16kB, respectively [33, p. 13]. Shared memory is primarily used for inter-thread communication or synchronisation.
- The *Register File* contains 2^{16} 32-bit registers with each executing thread having access to 63 of them [33, p. 7].

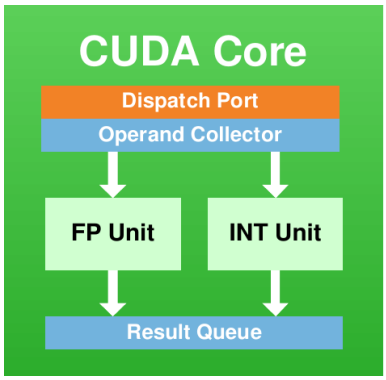


Figure 2.3.: NVIDIA CUDA core block diagram [32]

Beside the GK104 family, there are other Kepler based GPUs which feature some differences and enhancements. The *GK110* for example combines texture and uniform cache into one *read-only cache* [33, p. 13] and has a SMX ready for dynamic parallelism [33, p. 14 et seqq.].

Maxwell

The next generation of NVIDIA's architecture is called *Maxwell* and was introduced in 2014 [30].

As depicted in Figure B.3, the Maxwell streaming multiprocessor (*SMM*) architecture was redesigned and separated into four blocks with a dedicated warp scheduler, instruction buffer, 32 CUDA cores, eight load / store units and eight SFUs each. This organisation aligns to the warp size of 32 threads and reduces scheduling costs. Most notably, texture and L1 cache have been joined into one component with the result that each SMM now holds 94kB of shared memory.

2.1.2. AMD GCN Architecture

With *GCN* (*Graphics Core Next*), AMD introduced an architecture comparable to CUDA [27]. Rather than GPCs and CUDA cores, it features *CUs* (*Compute Units*) with *SIMD units* (*Single Instruction Multiple Data units*) to process *wavefronts*. A wavefront is equivalent to a warp with the difference that it contains 64 instead of 32 threads.

Figure B.5 shows the structure of a CU. It contains four SIMD units each featuring a 16 lane wide pipeline for single and double precision floating point calculations. In addition, all SIMDs possess 64kB of registers, a program counter and instruction buffers for ten wavefronts. The instruction buffers are backed up by a L1 instruction cache which is shared between four CUs. Similar to the previously mentioned SIMT model, each SIMD unit executes one instruction per cycle on 16 threads which not necessarily belong to the same wavefront. There are no special function units but microcodes are used to calculate transcendental functions.

Similar to CUDA's shared memory, a CU has 64kB of memory called *LDS* (*Local Data Share*) used for communication and synchronisation of wavefronts.

To manage control flow within wavefronts, the *Scalar Unit* examines conditional jumps or handles interrupts, function calls and returns. It is also used as an address generator to access memory for reads and writes through *L1 Cache*. This is not only used for general purpose data but also for texture sampling by using *TMUs* (*Texture Mapping Units*) which fetch and filter texel data.

In contrast to CUDA, fixed functionality of the graphics pipeline is not integrated in the CU but centralised in the GPU. To send data back to the pipeline e.g. after vertex shading, the *Export Unit* is used. It also grants access to the *GDS* (*Global Data Share*) which is similar to LDS but shared by the whole graphics processing unit and can be used for communication between several wavefronts dispatched to different CUs.

One example for a graphics card based on GCN is the *Radeon HD 7970*. As shown in Figure B.4 it contains 32 CUs. Connection to the host system is established by PCI-Express 3.0.

A *Command Processor* receives API calls and invokes the responsible components: *ACE*s (*Asynchronous Computation Engine*s) in case of general purpose computing or the *GCP* (*Graphics Command Processor*) in case of graphics processing.

Both hold a queue of tasks to keep track of execution, synchronisation and dependencies (e.g. fragment shading only after rasterisation). Tasks are executed by sending them to the *shader array* divided into work groups. If the necessary resources are available, wavefronts are formed and dispatched to CUs. Since work items and wavefronts are executed concurrently, ACEs and GCP need additional synchronisation to ensure completion and order of tasks.

The graphics pipeline for example works similar to the one implemented in CUDA. The *Geometry Engine* is used to assemble triangles following which vertex shading is executed [27, p. 12]. After a CU has finished, it sends back its result using the export unit and tessellation is performed by the geometry engine once all vertices have been returned. When the geometry stage has completed (i.e. after geometry shading), results are passed to the *Rasterizer* and once again to CUs for fragment shading. Now, these fragments are handed over to the *Render Backend* where blending, depth test and antialiasing is performed by ROP units before pixels are written to the frame buffer through memory controllers.

2.2. GPGPU Frameworks

The last section introduced different hardware architectures which enable general purpose computing on GPUs. To achieve an abstract programming level and to make application development more independent of underlying hardware, various frameworks for different use cases are available to program graphics processors.

After a section about basic concepts which all of them have in common, an introduction to the most important frameworks is provided. Since this thesis is based on OpenGL, this framework is covered by the dedicated chapter 3.

2.2.1. Basic Concepts

Code that is executed on the GPU (*device code*) is put into so called *kernels* or *compute shaders* which are written in a framework specific language, e.g. an extension of C [31, p. 80 et seqq.] or a shading language like GLSL. Since these are highly hardware dependent, they are mostly *jit*-compiled (*just in time*) when an application is started, so there is no need to provide object code for all available graphics cards. In case of OpenGL, compute shader source can be read from a file at runtime and is then compiled by the graphics driver by calling `glCompileShader(...)` [34, Chapter 19]. As this slows down program start-up, some frameworks also provide a method to use precompiled kernels with previously mentioned drawbacks [31, Chapter 3.1.1], [35, Chapter 5.8].

The *host code* which runs on the CPU dispatches kernels by API calls. A kernel is sent to GPU as a *global work group* consisting of *local work groups*. Each local work group is further subdivided into threads called *work items* which are basically *invocations* of the kernel [1, Chapter 12]. These are executed concurrently by the GPU and even local and global work groups can run in parallel [31, p. 32].

Each GPGPU framework has its own names for global / local work groups and work items (see Table 2.1). This section uses the naming convention of OpenGL compute shaders.

Table 2.1.: Naming conventions of GPGPU frameworks OpenGL [1], CUDA [31], OpenCL [36] and DirectCompute [37].

OpenGL	CUDA	OpenCL	DirectCompute
compute shader	kernel	kernel	compute shader
global work group	grid	NDRange	grid
local work group	block	work group	thread group
work item / invocation	thread	work item	thread

As can be seen in Figure 2.4, global and local work groups can be (but not necessarily have to be) defined with up to three dimensions resulting in a three dimensional ID for both, local groups and work items. This can be used to adapt a given use case or data structure which is addressed by the compute shader. A particle system for example could store particle data in a linear array whereas matrix and volume operations are natural 2D and 3D problems respectively [31, p. 10]. The number and size of local work groups are defined on dispatch [35, Chapter 5.10], [31, Chapter 2.2]. It is also possible to hard-code the number of work items per local group so the compiler can perform optimisation like utilisation of registers [36, Chapter 6.7.2]. Direct3D and OpenGL use the latter method [22, p. 291], [38, p. 58 et seq.].

Example: When filtering an image, a shader of local work group size 32 x 16 could be used so rectangles of 512 pixels are processed per group. The number of local work groups required to manipulate all pixels can be determined by dividing height and width of the input image by work group size:

$$\text{GlobalWorkGroupSize\_x} = \text{ImageWidth}/\text{LocalWorkGroupSize\_x}$$

$$\text{GlobalWorkGroupSize\_y} = \text{ImageHeight}/\text{LocalWorkGroupSize\_y}$$

For a 1024 x 768 sized image, 32 x 48 local work groups are required resulting in one invocation per pixel:

$$(32 \cdot 48) \cdot (32 \cdot 16) = 1024 \cdot 768$$

The concept of global work groups, local work groups and work items is tightly coupled to GPU architectures described in section 2.1 [32, p. 6 et seq.]. Table 2.2 shows this relationship as well as different layers of memory available. Invocations are scheduled

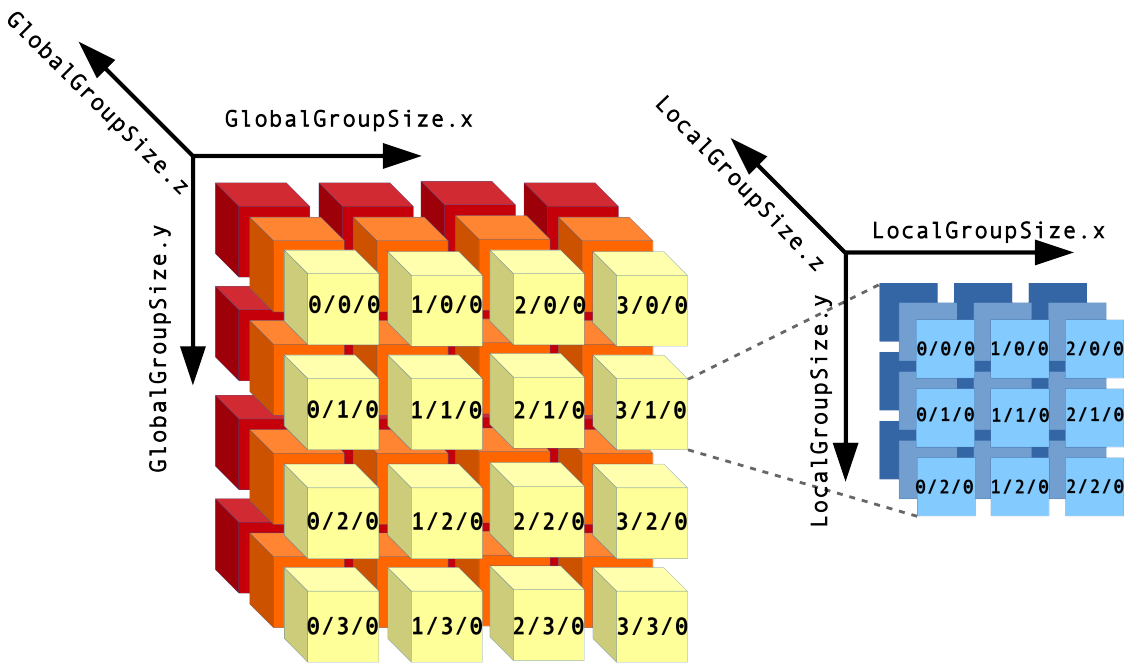


Figure 2.4.: Relationship of global work group, local work groups and work items. *Left:* Global work group subdivided to local work groups; *Right:* local work group subdivided to work items

on computation units like CUDA cores or AMD's SIMDs. Similar to functions, they have private local memory not accessible by others. Work items are dispatched to multiprocessors (streaming multiprocessors, compute units) in sets of local work groups. Local shared memory of a multiprocessor is used for communication and synchronisation between items within the same local work group. On the highest level, several global work groups are executed on the GPU. Global graphics memory is shared between all work items as well as between all active global work groups.

When dispatched to compute units, work items are grouped to form warps or wavefronts. To optimise resource usage, the local work group size should be a multiple of warp or wavefront size or a multiple of available compute cores and enough local work groups should be used to keep multiprocessors busy [31, Chapter 5.2], [39, p. 6]. The effect of a unsuitable number of work items per group is presented in chapter 6.

Unlike normal functions, no explicit parameters are passed to compute shader invocations but all share the same inputs [1, Chapter 12]. Data is accessed in form of *buffer* or *image* objects. Buffers can be seen as a special kind of array and images (also named textures in some cases) are optimised to read and write pixel storages.

To access the correct position in these data structures, each work item knows its own ID and the ID of its superior local work group [38, ch. 7], [31, p. 9 et seqq.]. Work item IDs are local which means they are only unique within a local group whereas local work group IDs are globally unique within their global work group.

Table 2.2.: Relationship of GPGPU framework concepts and GPU architecture

software view	hardware view	memory
work item	threads on compute core	per item private local memory
local work group	warps / wavefronts on multi-processor	per local group shared memory
global work group	kernels on GPU	global GPU memory

In case of image processing shown in Figure 2.5, the work item with ID (3/2) of its local work group with ID (1/1) can determine the position of the corresponding pixel as follows:

$$\text{Position\_x} = \text{GroupID\_x} \cdot \text{GroupSize\_x} + \text{ItemID\_x}$$

$$\text{Position\_y} = \text{GroupID\_y} \cdot \text{GroupSize\_y} + \text{ItemID\_y}$$

Given a local group size of 5x4, the appropriate pixel resides at coordinates (8/6) of the texture.

To populate buffers and images, API calls are used to copy data from host to device memory (`clEnqueueWriteBuffer(...)` [35, p. 101]) or to map buffers into host memory address space (`glMapBuffer(...)` [34, Chapter 6.3]).

As always, parallel computing needs mechanisms of synchronisation to avoid data hazards and race conditions or to retain order of execution. There are two main points in case of GPGPU [35, p. 32]:

- Host to device: Compute shaders are dispatched by an API call but control is returned to the host program immediately thus before executions have finished or even began [31, Chapter 3.2.5]. If the host relies on the results it has to wait until computation has completed. This can be done with API commands such as `glMemoryBarrier(...)` which waits until all writes to memory have been accomplished [34, ch. 7.12].

- Device to device: Since input and output buffers and images are shared between all work items, there are cases where access has to be synchronised. For example, race conditions can occur when applying a filter like blur to an image. As source and destination are the same and the output of one compute shader relies on pixels other instances may manipulate, no invocation must write to the image until all pixels have been recalculated.

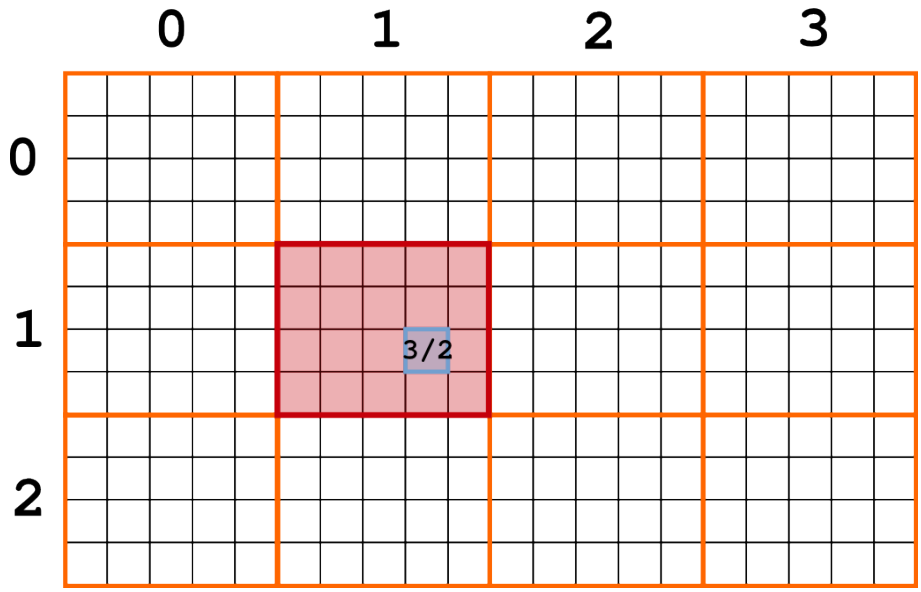


Figure 2.5.: Relationship between ID of local work groups (orange / red borders) and work items (black / blue borders). Local group IDs are shown on top (x) and on left side (y).

2.2.2. Overview of Frameworks

This section gives a short overview over some of the available GPGPU frameworks. For historical reasons, it also mentions how the rendering pipeline of OpenGL can be used for general purpose computations.

Frameworks covered by this section are:

- CUDA
- OpenCL
- DirectCompute

Using the rendering pipeline

Before GPU architectures and frameworks for general purpose computing had been released, GPGPU was possible by using the rendering pipeline. As an example, [40] describes mapping of concepts to the graphics processing scope of OpenGL:

- Floating point textures are used as input and output arrays.
- Compute logic is realised in fragment shaders rather than compute shaders.
- Computing is done by drawing to a texture.

At first, textures are defined which are used to map CPU side input data onto a rectangle rendered by the GPU. Texture size and especially their format have to be chosen depending on the data to process. This rectangle is transformed by a projection matrix that maps one fragment of the textured quad to one pixel of the frame buffer to avoid interpolation. To save computation results, another texture is bound to the frame buffer. Once again, size and format of the target texture have to match the use case.

When everything is set up, off-screen rendering is issued to OpenGL and fragment shaders perform the computations. They read their input data from the source texture and write their results to the OpenGL built-in variable `gl_FragColor`. These values are stored in the target texture bound to the frame buffer.

After rendering of the whole rectangle completed, this texture can be used to retrieve the computed data or for further calculations by another rendering pass.

This method of GPGPU has some major disadvantages because mapping problems to graphics scope is not always that easy. Also, neither ways of inter-process communication, nor synchronisation, nor atomic operations have been available until the introduction of compute shaders [41].

NVIDIA CUDA

CUDA is not only the name of NVIDIA's current hardware architecture but also their programming model which was introduced in 2006 and is now available in version 6.5 [31].

It uses an extension of C as high level language but other languages are supported, as well. Kernels are written in C and can be directly incorporated in the rest of program source code. Two interfaces are exposed to the developer: the low-level driver API and the runtime API based on the former. The latter is suitable for most cases but the driver interface provides more control and flexibility.

To build a CUDA program, the *nvcc* compiler is used. The first compilation step is the separation of host and device source resulting in pure C and *PTX* (*Parallel Thread Execution*) code respectively. Host code can now be compiled to binary by nvcc or any other compiler. PTX is the name of a virtual machine featured by CUDA GPUs which translates device code to the specific hardware instruction set when the kernel is loaded the first time [42].

Since host and device object code depend on the driver and hardware used, both are versioned. The software component has a CUDA version and GPUs have *compute capabilities* also called *SM versions*. Macros are provided to distinguish between available versions on compile time and API calls can be issued to query versions at runtime.

CUDA has a feature-rich API which also provides specialised functionality like textures, multiple devices, address space mapping and an inter-process communication API to transfer context sensitive data to other processes. A technique called *dynamic parallelism* enables kernels to launch other kernels and thus create new grids at runtime.

Graphics rendering is not covered, because CUDA is dedicated to GPGPU only. For this task, graphics libraries like Direct3D or OpenGL have to be used. CUDA provides an interface to register and map resources of both. While they are mapped, these must

not be manipulated by any graphics library operation because the resulting behaviour is undefined.

Developed by NVIDIA, CUDA is highly optimised but also only available for GPUs of their own brand [43].

Although being a feature-rich and optimised framework, CUDA was not used for this thesis for the following reasons:

- Runtime libraries: Additional libraries have to be installed by the user or statically linked to the program.
- Graphics interoperability: This requires additional work to be implemented in a safe way. Performance overhead is created when sharing graphics resources (see Figure 2.6). Also hiding explicit interoperability from the end user is more complex and could cause undefined behaviour when mapped resources are not accessed properly.
- Proprietary nature: Since CUDA is not available for non-NVIDIA graphics cards, it would break the engine's goal of vendor independence.

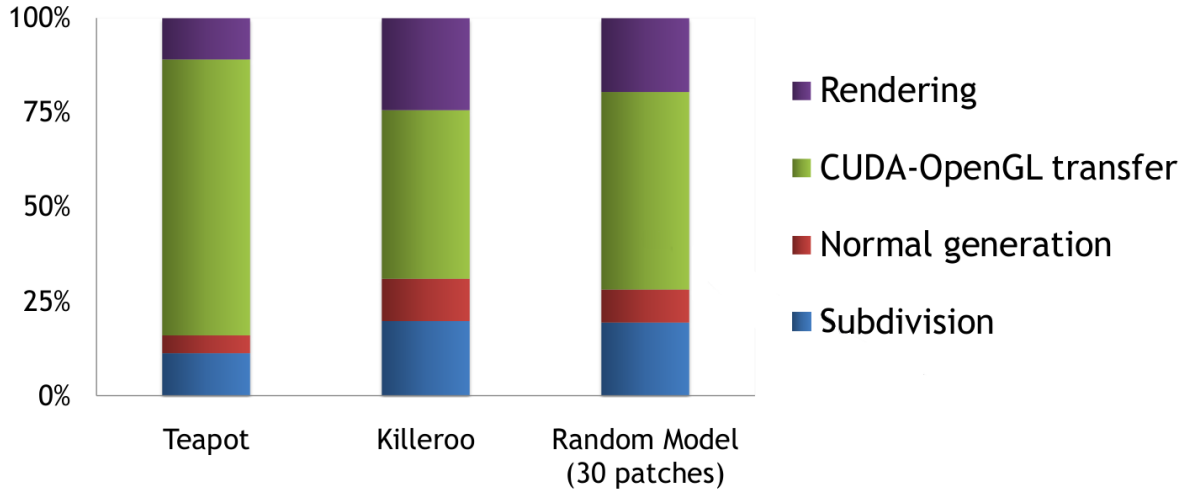


Figure 2.6.: Relative time consumption of different parts of CUDA programs with OpenGL interoperation. The programs are used for subdivision surface implemented with CUDA. [37, p. 29]

OpenCL

OpenCL (*Open Compute Library*) is an open standard first proposed by Apple and later on transferred to Khronos Group [44, Chapter 1]. Specification 1.0 was introduced in December 2008 [45] and latest version 2.0 released in November 2013 [46] which can be

found in [35]. The goal of OpenCL is to provide a uniform standard for heterogeneous platforms consisting of a set of different compute devices. Embedded systems like smart phones or tablets are also covered.

OpenCL is divided into platform layer, runtime environment and the OpenCL C programming language.

The platform layer provides an abstract view of heterogeneous systems: they contain a host and a multitude of compute devices. The capabilities of devices are queried by the host via API calls and contexts are created for all devices used by the program.

The runtime layer provides a low level API with methods to interact with these devices and to dispatch kernels for execution. The generic nature of OpenCL enables hardware to expose fixed functionality as so called *built-in* kernels. Due to the low level architecture, OpenCL is a powerful and flexible but also complex framework.

All hardware vendors who want to support OpenCL in their drivers have to provide an *ICD* (*Installable Client Driver*) which implements the OpenCL API [44, Chapter 2]. Function calls by the host code is passed on to the ICD which actually executes the command on the hardware.

Kernels are written in a variant of C99 [36]. The OpenCL C compiler builds them from source at runtime but it is also possible to provide precompiled binaries and even program libraries are supported [35].

Compared to CUDA, versioning is done in three different ways:

- Platform: The version of the OpenCL runtime used by the host program.
- Device: The OpenCL version supported by a specific device.
- Language: The OpenCL C language version supported by a specific device.

Device and language versions have to be queried at runtime but can be mixed within one OpenCL program.

Special functions are provided to support OpenGL interoperability but depend on the platform and hardware used [44, Chapter 10], [24]. At first, a OpenGL compatible context has to be created. This requires a special extension ("`cl_khr_gl_sharing`") which must be provided by the OpenCL device. Similar to CUDA, OpenGL resources can be registered and mapped. Once again, this needs careful synchronisation to prevent undefined behaviour.

Although OpenCL is a powerful framework supported by GPU vendors like AMD and NVIDIA, there are reasons why it was not chosen for this thesis:

- Additional installations: Similar to CUDA, the OpenCL runtime library is required to execute OpenCL programs. When computing on a graphics card, ICDs are not an issue since NVIDIA and AMD incorporate these into their graphics drivers.
- Graphics interoperability: An OpenGL sensitive context has to be created and resources have to be registered and managed explicitly by OpenCL. Similar problems arise as with CUDA. Also, availability of interoperability depends on the hardware and operating system specific set up code is required [44, Chapter 10].

- Code complexity: More set up code for devices, contexts and kernels is needed as well as capabilities of underlying hardware have to be queried. End users of the simulation engine also require deep knowledge of the complex OpenCL API if they want to extend the engine at some point.

DirectCompute

DirectCompute is the GPGPU component of Microsoft's Direct3D graphics library since version 11 [22, Chapter 5]. It is available for DirectX 10 and 11 compatible GPUs, but only for Windows Vista and above [37, p. 4].

Just as render shaders, compute shaders are written in *HLSL* (*Highg Level Shading Language*) and are set up the same way. Instead of being executed in the rendering pipeline, a decoupled compute stage is provided.

Being part of the same library, all graphics resources can be bound to and accessed by compute shaders similar to their graphics counterparts making communication to third party frameworks redundant. Also, resources do not have to be acquired explicitly.

DirectCompute meets the requirements for the simulation engine proposed by this thesis: Compute shaders are lightweight to set up, no additional libraries or drivers have to be installed and end users familiar to Direct3D do not have to learn an additional framework or language. Despite those advantages, DirectCompute is only available for Windows operating systems [37] which conflicts with the cross-platform philosophy of the engine.

3. OpenGL Compute Shaders

OpenGL is Khronos Group's open and cross-platform graphics standard [34]. GPGPU was only possible by using the rendering pipeline as mentioned in section 2.2.2 until compute shaders were introduced with the "ARB_compute_shader" extension [47] which was finally added to the *core profile* of OpenGL in version 4.3 [34, p. 675].

This chapter gives a short introduction on how to use OpenGL compute shaders for general purpose computing. At first, it is compared to frameworks described previously including reasons why OpenGL was selected for this thesis. Afterwards, a deeper look into the functioning of compute shaders is taken.

3.1. Comparison to other Frameworks

OpenGL compute shaders are part of OpenGL similar to DirectCompute being part of Direct3D [34, ch. 19]. This means, that no additional drivers besides the graphics cards driver have to be installed. OpenGL 4.3 is supported by NVIDIA since Fermi [48] and by AMD since their Radeon HD 5000 series [49].

Compute shaders are written in *GLSL* (*OpenGL Shading Language*) and handled equally to render shaders. Not being related to rendering, a separate compute stage is used to execute these apart from the rendering pipeline.

Since compute shaders are part of the graphics library, no context switches or resource registering / mapping has to be performed: graphics resources like textures can be bound to a compute shader the same way as to render shaders.

Although less functionality is supported compared to CUDA or OpenCL, compute shaders still have enough power for complex physical simulations [26]. Reduced complexity also facilitates integration into the simulation engine as well as its extensibility by end users. It is also not necessary to learn an additional language or entire framework if users are familiar with OpenGL and GLSL.

To summarise, OpenGL was selected because it provides the features needed for the engine proposed by this thesis:

- Cross-platform and vendor independence
- Features and computation power even for complex simulations
- No explicit resource sharing between graphics and compute library required
- Easy to learn and easy to extend by end users
- No additional libraries or drivers required

3.2. Using Compute Shaders

This section shows, how compute shaders are used. It covers the creation and implementation as well as input and output handling, shared memory and synchronisation techniques.

3.2.1. Creating and Dispatching a Compute Program

Compute shaders are created and used the same way as render shaders [34].

At first, a shader object has to be created by calling

```
GLuint glCreateShader(GLenum shaderType);
```

with `shaderType` equal to `GL_COMPUTE_SHADER`. The returned value is the ID of a new shader object. Source code can be attached to it via

```
void glShaderSource(GLuint shader,
                    GLsizei count,
                    const GLchar **string,
                    const GLint *length);
```

Parameter `shader` is the shader handle retrieved via `glCreateShader`. Source code is passed in an array of `count` strings. Lengths of strings are stored in `length`. The shader can be compiled by following command:

```
void glCompileShader(GLuint shader);
```

Now, a shader program has to be created with

```
GLuint glCreateProgram();
```

and shaders can be attached by calling

```
void glAttachShader(GLuint program, GLuint shader);
```

using the program handle returned by the previous function call. Either a compute shader or render shaders may be attached to a program. If they are mixed, an error is created when linking the program. At the end, the program has to be linked using

```
void glLinkProgram(GLuint program);
```

If this operation was successful, the program can be activated via

```
void glUseProgram(GLuint program);
```

Render shaders are implicitly invoked when a render call was issued through one of the `glDraw*(...)` commands. Compute shaders instead have to be issued by the

```
void glDispatchCompute(GLuint num_groups_x,
                       GLuint num_groups_y,
                       GLuint num_groups_z);
```

command. This enqueues the currently used shader program for execution. Parameters are the number of local work groups which form the global work group as described in subsection 2.2.1.

3.2.2. Writing a Compute Shader

Like render shaders, a compute shader is written in GLSL (see Listing 3.1). Since compute shaders were introduced in OpenGL 4.3, this version or higher has to be requested by the `#version` preprocessor command. A special `layout` qualifier defines the local work group size by setting `local_size_x`, `local_size_y` and `local_size_z`:

```
layout (local_size_x = 16,
        local_size_y = 16,
        local_size_z = 1) in;
```

A qualifier can be omitted if it is set to one, so this line is exactly the same as above:

```
layout (local_size_x = 16, local_size_y = 16) in;
```

Listing 3.1: Minimal working example for a compute shader [1, ch. 12].

```
1 #version 430 core
2
3 layout (local_size_x = 16, local_size_y = 16) in;
4
5 void main(void)
6 {
7     // Do nothing.
8 }
```

As described in subsection 2.2.1, each invocation of a compute shader has to determine the set of data to operate on. In GLSL, there are several builtin variables to address this problem:

- `gl_NumWorkGroups` is a three dimensional vector which holds the number of work groups. This is equivalent to the parameters passed to `glDispatchCompute(...)`.
- `gl_WorkGroupSize` is a three dimensional vector which stores the size of a work group specified by the layout qualifiers `local_size_x` and so on.
- `gl_WorkGroupID` is the three dimensional ID of the work group to which the shader invocation belongs as illustrated in Figure 2.4.
- `gl_LocalInvocationID` is the three dimensional ID of a work item relative to its work group as illustrated in Figure 2.4.
- `gl_GlobalInvocationID` uniquely identifies one work item from all invocations within a dispatch call by combining `gl_WorkGroupID` and `gl_LocalInvocationID` into one single 3D-vector.
- `gl_LocalInvocationIndex` is a flattened version of `gl_LocalInvocationID` which is computed as shown in equation 3.1. This can be used to index one-dimensional arrays [1, ch. 12].

$$\begin{aligned}&\text{gl\_LocalInvocationIndex} = \\ &\quad \text{gl\_LocalInvocationID.z} \cdot \text{gl\_WorkGroupSize.x} \cdot \text{gl\_WorkGroupSize.y} + \\ &\quad \text{gl\_LocalInvocationID.y} \cdot \text{gl\_WorkGroupSize.x} + \\ &\quad \text{gl\_LocalInvocationID.x};\end{aligned} \tag{3.1}$$

3.2.3. Input and Output

There are several ways to specify input and output variables:

- *Uniform* variables and buffers (write only)
- *Textures* (write only)
- *Images* (read and write)
- *Shader Buffer Objects* (read and write)

All of these methods are used the same way as in render shaders but since images and shader buffers are important to pass and retrieve data to and from compute shaders, they are explained below.

Image Objects

Image objects work similar to *texture samplers* but also allow write access to underlying textures [34, ch. 8.25]. To define an image, a new texture has to be created at first using the function

```
void glGenTextures(GLsizei n, GLuint* textures);
```

which creates `n` new textures and saves their IDs to `textures`. With

```
void glBindTexture(GLenum target, GLuint texture);
```

textures are bound to *texture units* which are activated by `glActiveTexture(...)`. Parameter `target` determines the dimensions of the texture, e.g. `GL_TEXTURE_2D` is used for two dimensions. To reserve space for the new texture, one of the `glTexStorage*(...)` methods is invoked. For example, `glTexStorage2D(...)` creates memory for two dimensional textures. All details can be found in [34, ch. 8].

To use a texture not only for read but also for write access, it has to be bound to one of the *image units* just as textures are bound to texture units:

```
void glBindImageTexture(GLuint unit,
                        GLuint texture,
                        GLint level,
                        GLboolean layered,
                        GLint layer,
                        GLenum access,
                        GLenum format);
```

Parameter `unit` is the image unit to which `texture` is bound. To select the mipmap `level` and texture `layer` (if `layered` is set to `true`), the eponymous variables are used. To tell OpenGL how the image is used, `access` is set to one of `GL_READ_ONLY`, `GL_WRITE_ONLY` or `GL_READ_WRITE` which are self-explaining. The `format` has to be compatible with the texture format specified with the `glTexStorage*(...)` call.

To access the image within a shader, it has to be defined as an input:

```
layout (binding = 0, rgba32f) uniform image2D data;
```

This makes the 2D texture bound to image unit at `binding` point zero accessible through the variable `data`. Other types are available for different texture types. The format specified in the layout qualifier has to be compatible with the format set by `glBindImageTexture(...)` but is only required if the image is written to. For details about image types and formats see [1, ch. 11].

The methods `imageLoad(...)` and `imageStore(...)` are used to read from and write to an image at given position `P`. They are overloaded functions to handle all types of textures. For example, the syntax for two dimensional textures is

```
vec4 imageLoad(image2D image, vec2 P);
vec4 imageStore(image2D image, vec2 P, vec4 data);
```

Shader Storage Buffer Objects

Another way to pass and retrieve data to and from shaders are *SSBOs* (*shader storage buffer objects*) which were introduced in OpenGL version 4.3 [34, p. 677].

They are created by the same command which is used to set up other memory buffers.

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

reserves `n` buffer handles and stores them in the `buffers` array. To actually use them, they have to be bound to one of the buffer targets by calling

```
void glBindBufferBase(GLenum target,
                      GLuint index,
                      GLuint buffer);
```

The `target` is one of the buffer binding targets whereas `index` is one of the available binding points for this target. The variable `buffer` is the OpenGL buffer handle generated by `glGenBuffers(...)`. A special target named `GL_SHADER_STORAGE_BUFFER` was introduced for SSBOs which consist of multiple binding points. All other targets like `GL_ARRAY_BUFFER` are valid as well so these buffers can be used for computation and as sources for e.g. vertex attributes at the same time.

As usual, API calls `glBufferData(...)` and `glBufferSubData(...)` can be used to write data into the buffer object whereas `glGetBufferSubData(..)` retrieves stored values. It is also possible to map the buffer via `glMapBuffer(...)`.

In GLSL, shader buffer objects are accessed via shader storage blocks which are declared with the keyword `buffer` as shown in Listing 3.2 [38, ch. 4.3.7]. The layout specifier `binding` equals the index of the buffer bound to the `GL_SHADER_STORAGE_BUFFER`

target which provides data for this block. Note that a "shader storage block declared without a `binding` identifier is initially assigned to block binding point zero" [38, ch. 4] making this qualifier mandatory if any other binding point should be used.

Listing 3.2: A basic shader storage block definition.

```
layout (binding = 0) buffer someDataBuffer
{
    int someVariable;
    vec4 colorData[];
};
```

The data of the buffer is accessed by the variables of the buffer block definitions and interpreted according to the type of the variable. In this example, the buffer contains one integer and an unsized array of four-dimensional float vectors. Only the last member of a buffer can be an unsized array [1, ch. 11]. It is also possible to define a buffer containing structures which is shown in Listing 3.3.

Listing 3.3: Shader storage block definition using structured data.

```
struct DataType
{
   int  id;
   vec3 position;
};

layout (binding = 0) buffer someDataBuffer
{
    DataType items[];
};
```

3.2.4. Shared Memory

In section 2.1, the concept of shared memory was introduced. To store a variable in shared memory, the keyword `shared` is used as can be seen in Listing 3.4.

Listing 3.4: Definition of shared variables.

```
// A shared integer
shared int sharedInt;

// A shared array of structured objects
shared struct baz_struct
{
    int id;
    vec3 currentPosition;
    vec3 oldPositions[8];
} moveableObjects[16];
```

Shared variables are used for reading and writing just as normal variables but since they are located in memory near to the compute core executing an invocation, operations

are performed more quickly. Access to memory used frequently by work items can be improved by copying the data into shared variables, manipulating them and writing them back to main memory at the end [1, ch. 12]. This optimisation technique is demonstrated in chapter 5 and evaluated in chapter 6.

As described in section 2.1, hardware limits the memory for shared variables. The size of available memory can be queried with `glGetIntergerv(...)` with parameter `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`.

3.2.5. Synchronisation

Since compute shaders are massively parallel threads, also running asynchronously to the host program executing on the CPU, synchronisation mechanisms have to be used to ensure order of execution and to prevent race conditions where necessary. As described in subsection 2.2.1, there are two types of synchronisation: device to device and host to device.

Device to Device

OpenGL does not guarantee the order in which work items or work groups are executed. This means, that reading data in one shader invocation can occur before or after a different work item has written to this location in memory even if both follow the same control flow. There are various techniques to synchronise control flow and memory access within a compute shader which are covered below.

Control flow synchronisation. To synchronise out of order execution, the `barrier()` command can be used. When a work item reaches a barrier, it waits until all other invocations within the same local work group reached this point [38, ch. 8.16]. For this reason a barrier must not be placed in diverging control flow paths, since a barrier stalls until all items reached the same barrier which is impossible unless all follow the same execution path. It is not possible to synchronise work items of different work groups with `barrier()`.

Memory barriers. The second way of synchronisation covers memory access. Reads from and writes to a shared variable, atomic counter (which are covered later on), image object or shader storage buffer are queued by the memory system [1, ch. 12]. A shader invocation can ensure that all writes it has issued are performed before continuing by calling one of the `memoryBarrier()` commands (see Table 3.1). This makes the work item wait until the memory system signals that all stores requested by this very item have completed.

Atomic operations. The GLSL has a variety of built-in atomic functions which operate on signed and unsigned integers but not on other data types like floats. These functions read a value from a specified memory location, modify and write it back to the same location. After this process completed, the original value is returned to the caller. It is

Table 3.1.: List of GLSL memory barrier commands [38, ch. 8.17].

GLSL command	Description
memoryBarrierBuffer()	Wait until all writes to shader storage buffers issued before the barrier have completed.
memoryBarrierImage()	Wait until all writes to image objects issued before the barrier have completed.
memoryBarrierShared()	Wait until all writes to `shared` variables issued before the barrier have completed.
memoryBarrierAtomicCounter()	Wait until all writes to atomic counters issued before the barrier have completed.
memoryBarrier()	A combination of all barriers listed above.
groupMemoryBarrier()	The same as memoryBarrier() but restricted to the work group of the invocation calling this function, i.e. writes are synchronised within the local work group but not for the global work group.

guaranteed that no other memory operation on the same part of memory interferes with an atomic operation [38, ch. 8.11].

For counting purpose, a special type for *atomic counters* is provided. These counters are 32-bit unsigned integer variables stored in buffer objects attached to the special `GL_ATOMIC_COUNTER_BUFFER` target at an index specified by the host application [34, ch. 7.7]. They are declared as `uniform` variables of type `atomic_uint`:

```
layout (binding = 0, offset = 0) uniform atomic_uint
   myCounter;
```

Once again, `binding` specifies the index of the buffer. Layout qualifier `offset` is an offset into the buffer, so multiple atomic counters can be stored within a single buffer. Since atomic counters are unsigned integers, the offset has to be a multiple of four.

To access these objects, the GLSL built-in methods `atomicCounterIncrement(...)`, `atomicCounterDecrement(...)` and `atomicCounter(...)` have to be used which increment, decrement or return the current value of the counter respectively [38, ch. 8.10].

Host to Device

OpenGL is based on the client-server model. The client (host) issues commands to the server (device) via API calls which are executed asynchronously to the calling application. For this reason, synchronisation between host and device must be possible. This

also includes the synchronisation of render and/or compute shaders executed by subsequent draw and dispatch calls. Similar to device to device synchronisation, the OpenGL API provides several methods for this purpose.

Control flow synchronisation. The easiest way of control flow synchronisation is to call `glFinish()`. This method blocks until all OpenGL commands issued previously have been executed [34, p. 18].

A more flexible way is provided by *sync objects* [34, ch. 4.1]. A sync object has two states: unsignalled and signalled. The call of `glFenceSync(...)` creates a new unsignalled sync object and injects a *fence* into the OpenGL command queue. When this fence is reached, the status of the related sync object is changed to signalled. The application can check the status with the `glGetSynciv(...)` command and, if the status is signalled, can safely assume that all commands prior to the fence have completed. To actively wait for a fence, `glClientWaitSync(...)` can be used which blocks until the fence has been reached.

Memory barriers. Similar to GLSL, the OpenGL API includes a method for memory synchronisation [1, ch. 11], [34, ch. 7.12]:

```
void glMemoryBarrier(GLbitfield barriers);
```

This synchronises memory transactions issued by a shader. All shaders executed afterwards see the latest data written by any other shader before the memory barrier.

Parameter `barriers` is a bitwise OR concatenated list of barriers to wait for. For example, `GL_SHADER_STORAGE_BARRIER_BIT` ensures that any writes initiated by shaders prior to the barrier have completed before the respective memory locations are used as source for shader storage objects. This means, that the barrier has to be set according to the intended usage of the modified memory: If a compute shader updates a buffer that serves vertex attributes for the next draw call, the barrier bitmask must include the `GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT` to ensure that the new data is used.

A detailed discussion of all barriers would be out of scope of this thesis. For a full list of all options available see [34, ch. 7.12].

4. Simulation Engine

The main focus of this bachelor thesis was the creation of a generic simulation engine capable of rendering and updating n-body systems in real time. This chapter presents the created solution.

At first, an architectural overview of the engine is given and the programming language and third-party libraries which have been used are stated. Afterwards, basic classes and functionality are described followed by an explanation of the creation, definition, rendering and updating of particle systems. The last two sections cover techniques of debugging and profiling and the engine built-in shader library.

4.1. Introduction

This section gives a short introduction of the created simulation engine by presenting an overview of the software architecture and explaining which third-party libraries are used.

4.1.1. Architectural Overview

The basic architecture of the engine is derived from Irrlicht3D and Orge3D. The most important components that make up a full simulation are listed below and discussed in detail in the referenced sections. Figure 4.1 on page 41 presents a diagram of all classes of the engine.

Engine: This is the main component of the simulation engine. It initialises the simulation, manages particle systems and is responsible for input processing for keyboard and mouse (subsection 4.2.1).

Buffers: Buffers are abstractions of the OpenGL buffer objects which wrap all required API calls. Special classes for uniform and atomic counter buffers are provided (subsection 4.2.2).

GPUProgramService: This object creates and manages shader programs as well as shader source code which can be loaded from files or directories to provide a possibility of creating a shader library (subsection 4.2.3).

ShaderProgram: This is a wrapper for OpenGL shader programs. The `RenderProgram` and `ComputeProgram` classes are derived from this parent which are used for rendering and computing respectively. All shader programs are managed by a `GPUProgramService` (subsection 4.2.3).

Mesh: This class holds geometrical information about a particle and is created through the `MeshManager` (subsection 4.2.5).

Material: Materials describe how a mesh is rendered. They are created through the `MaterialManager` (subsection 4.2.5).

RenderSystem: A `RenderSystem` is used to initialise OpenGL and to abstract the rendering related parts of the graphics API. It is used to draw particle systems by setting up meshes, materials and issuing the render call (subsection 4.2.4).

ComputeSystem: Similar to drawing particles with the render system, a class called `ComputeSystem` is used to update particle attributes which controls the set up and dispatch of compute shaders (subsection 4.2.4).

ParticleSystem: A `ParticleSystem` holds the data (buffers) and behaviour (actions) and thus the definition of a particle or n-body system. A simulation can contain multiple systems at the same time (section 4.3).

Action: Actions are used to model the behaviour of a particle system. Multiple actions can be attached to one system to achieve complex results (section 4.3).

4.1.2. Programming Language and third-party Frameworks

C++ was chosen as programming language because it is object-oriented, cross-platform, fast [50] and used by many 3D engines and video games [15].

Some third party frameworks to handle OpenGL, mathematical operations and file system access are used all of which are cross-platform.

OpenGL can be considered the main framework that was chosen for the reasons stated in chapter 3. It is used to render and update particle systems. The GLEW (OpenGL Extension Wrangler Library) takes care of loading available OpenGL extensions [51].

To create an OpenGL context and a render window, the GLFW3 library is used due to its flexibility and its input handling system [52].

The GLM library introduces vectors and matrices to C++ which are used similar to vector and matrix operations in GLSL [53]. This is a header only, thus lightweight, library.

The Boost filesystem library is used to read shader source code from files and directories since it provides a convenient abstraction of platform dependant file system operations [54].

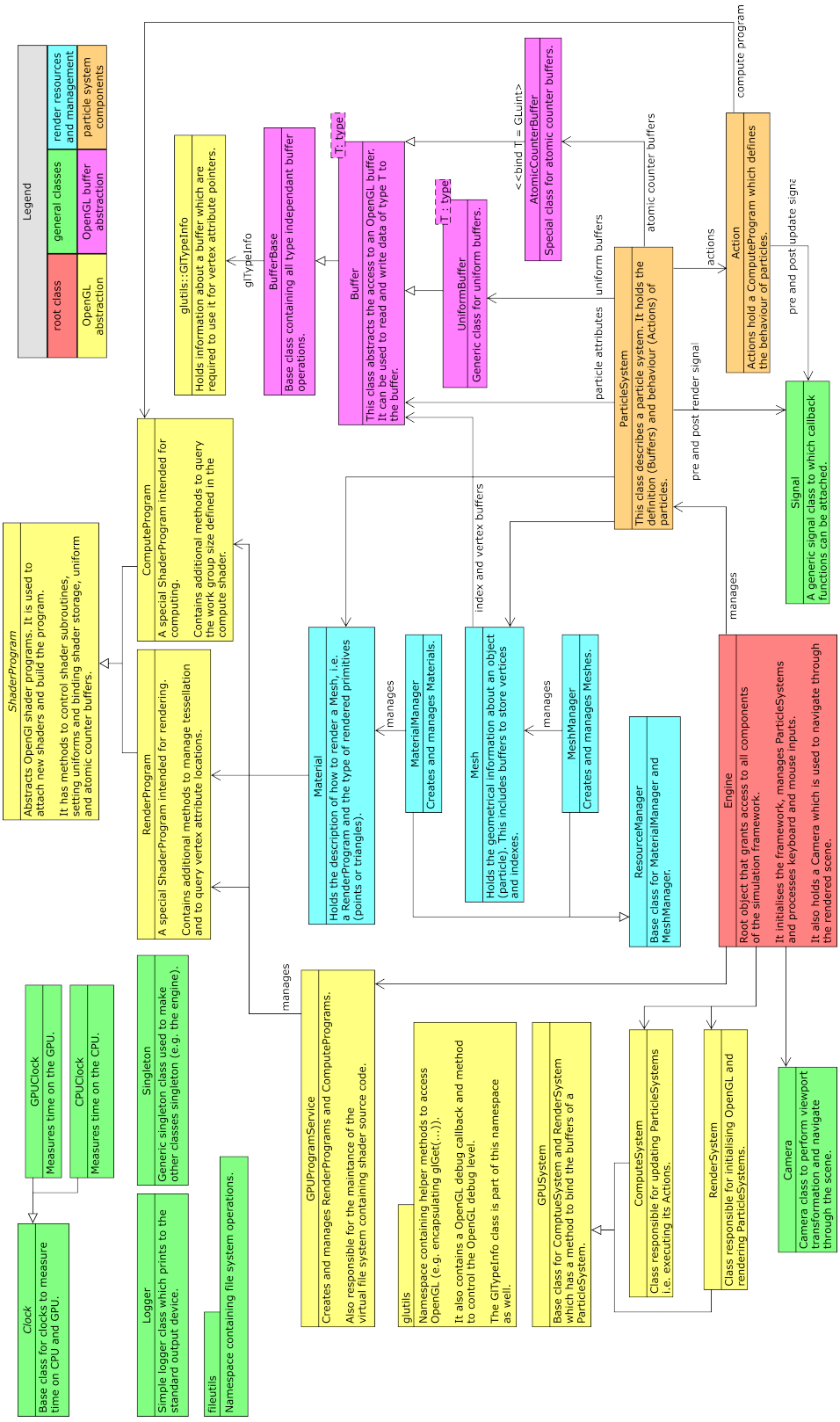


Figure 4.1.: Class diagram of the architecture of the engine.

4.2. Basics

This section covers the basic concepts and functionality of the simulation engine by explaining the classes mentioned above. Since the engine is centred around particle systems, these are discussed in section 4.3.

4.2.1. The Engine

Similar to Irrlicht3D and Ogre3D, there is a root object which is called `Engine`. It is the central interface to access all components of the framework and declared as singleton since only one engine may be active at a time. Listing 4.1 demonstrates the most basic set up to begin with a new simulation.

Listing 4.1: Basic set up of a simulation.

```
1  int main()
2  {
3      Engine* engine = Engine::getInstance();
4      engine->init(1024, 768, false);
5
6      // Set up code for shaders, materials, meshes and
7      // particle systems goes here...
8
9      while(!engine->windowClosed())
10     {
11         engine->processEvents();
12         engine->updateAllParticleSystems();
13         engine->drawAllParticleSystems();
14     }
15
16     engine->terminate();
17     return 0;
18 }
```

The first step is to initialise OpenGL as render and compute backend by calling `Engine::init(...)`. Parameters are the screen width, screen height and a flag to toggle full screen mode. A fourth parameter enables OpenGL debug capabilities which are covered in subsection 4.4.1. If initialisation was successful, a new window is created which is used as render target.

New particle systems are created by calling `Engine::createParticleSystem(...)`. Their definition, rendering and update process are covered in detail later on in section 4.3.

The `while`-loop in line 9 executes until the user requests the window to close. This main loop (in game engines also referred to as *game loop*) successively processes mouse and keyboard input events (line 11), updates all particle systems (line 12) and renders them to the window (line 13).

After the main loop was quit, `Engine::terminate()` has to be called to destroy the render window, the OpenGL context and all resources created by the simulation framework.

The engine also includes a first person camera class [23, ch. 2] which enables the user to navigate through the scene via mouse and keyboard. The standard controls are listed in Table C.2. When rendering, the view projection matrix of this camera is passed to the render system for transforming the view port accordingly.

Besides this predefined inputs, the engine exposes a `keyEventSignal` to which custom input handlers can be attached. Whenever a key was pressed, all registered callbacks are notified about this event. This allows the user to define custom controls, e.g. switching render modes. Callback functions for this signal have the interface

```
void keyCallback(int keycode, int action, int modifiers);
```

The parameters match the GLFW parameters for key callback functions [52]. The function is connected to the signal via

```
engine->keyEventSignal.connect(keyEventCallback);
```

From now on, when `Engine::processEvents()` is executed, all registered handlers get notified about key events and are able to perform customised actions.

Signals for mouse movement and button presses can be implemented the same way but are currently not supported.

4.2.2. Memory Objects

OpenGL has a set of memory objects to represent buffers, images or textures (see chapter 3). The simulation engine provides an abstraction API for these objects to simplify their usage. The developed wrapper classes are managed by abstract objects like meshes or particle systems and binding is controlled by the render and compute systems (see subsection 4.2.4 and section 4.3 respectively). The end user usually accesses the memory objects to write or read data but he is also permitted to call low-level functions if necessary, e.g. apply buffer bindings manually.

Buffers

OpenGL buffers are general data stores which are used in various ways. Most notably, vertex positions and indexes are read from buffers and passed to the vertex shader. With the introduction of shader storage buffers, uniform buffers and atomic counters, even more use cases arose. SSBOs are the main method of supplying input to compute shaders and reading their output.

Ogre3D and Irrlicht3D provide a wrapper class for buffer objects. Whereas Irrlicht3D's video driver is responsible to update their data, Ogre3D's buffer class encapsulates all necessary OpenGL calls. The abstraction used by the simulation engine follows the basic concepts of the latter as explained in the following.

Buffer objects are created immutable, i.e. with a fixed size. This means that the maximum number of items cannot be modified afterwards which is not necessary since n-body systems rarely change the number of particles and uniform and atomic counter buffers retain the number of stored items. On creation, the actual size of a buffer

of a generic type `T` is calculated by the constructor as `bufferSize = sizeof(T) * itemCount`.

OpenGL implementations can optimise buffer handling according to some hints. The most important one is passed as `usage` parameter to `glBufferData(...)` on buffer creation and describes how and at which frequency the buffer is used by the application [34, ch. 6.2]. As second hint, the graphics library "may make different choices about storage location and layout based on the initial binding" [34, p. 54]. Both is handled by the constructor of a buffer by choosing a suitable `usage` and by binding it to an appropriate target. If necessary, they can be specified manually, as well.

Another level of abstraction regarding type safety is included. On client side, functions like `glBufferData(...)` and `glMapBuffer(...)` receive and return pointers of type `void*`. Casts from and to `void*` are error-prone since the C++ compiler cannot perform any type checks which may lead to undefined behaviour. To resolve this issue, the buffer wrapper is a template class which ensures that only data that matches the template type is associated with the underlying OpenGL buffer. This also makes data handling more convenient, because no manual casting is necessary.

Since some operations on buffers are type independent, a `BufferBase` class is provided which is specialised by the templated buffers. This makes it possible to store buffers of different types in a single STL container, as well.

The data of a buffer can be modified by either `setData(...)` or by buffer mapping. The first one sets the values of the whole buffer which is why a pointer to a memory location that matches its size has to be passed. The latter works similar to the mapping of OpenGL itself. When `map()` is called, a pointer is returned that can be used to read and / or write to the buffer. After all operations completed, `unmap()` has to be called to safely dissolve the mapping.

To correctly address the data, the different buffer layouts have to be kept in mind [36, p. 130 et seqq.]. If a shader storage block is defined with neither `std140` nor `std430` layout qualifier, its variables can be reorganised or omitted by the OpenGL implementation which results in several drawbacks. Querying and calculating the location of variables creates additional overhead and the user has to know these locations when accessing mapped buffers. Additionally, no structured data can be used since the buffer class cannot associate locations of the structure with variables of the storage block.

As mentioned above, OpenGL buffers can be used as the source of vertex data. Vertex attributes are set up by calls to `glVertexAttribPointer(...)` with mandatory parameters being the type (e.g. `GL_FLOAT`) and size (e.g. 3 for a three dimensional vector). Buffers have a `GLTypeInfo` object to provide this information. A built-in mechanism is used to guess these parameters from known types (see Table C.1) but they have to be initialised manually if unsupported or structured ones are used.

Another issue that is encapsulated by the buffer implementation is the bind-to-modify principle of OpenGL. This means that buffers have to be bound to a target before data can be assigned or before they can be mapped. To make this task more convenient, buffers are bound implicitly when necessary. By using the target `GL_COPY_READ_BUFFER`, conflicts with other bindings is minimised. Furthermore, the current state is saved and restored afterwards. Listing 4.2 shows this procedure in pseudo code.

Listing 4.2: State preservation for buffer bindings.

```
// An operation that needs the buffer to be  bound to a target
// because of OpenGL's bind-to-modify design.
Buffer::setData()
{
    // Save current buffer binding
    currentlyBoundBuffer = getBoundBuffer(GL_COPY_READ_BUFFER);

    // Bind this buffer and modify it
    bind(GL_COPY_READ_BUFFER, this);
    someOpenGLOperation();

    // Restore previously binding state
    bind(GL_COPY_READ_BUFFER, currentlyBoundBuffer);
}
```

In addition, to avoid bindings that fell out of use, the current implementation limits the number of bindings established manually by calling `Buffer::bind(...)` to one. This is done by checking the current state of a buffer. If the buffer was bound by this method previously, the binding is dissolved before a new one is applied. Although this limitation does not affect the sample applications provided by this thesis it can be easily removed if it becomes too restrictive for future applications.

Uniform Buffers

A separate class derived from the buffer class is provided for wrapping uniform buffers. This is mainly used to deliver a proper binding target (`GL_UNIFORM_BUFFER`) and usage hint which defaults to `GL_DYNAMIC_DRAW` because uniform buffers are most likely modified by the engine but never by shader programs.

Similar to the generic class, this is a template class to be able to store data of arbitrary types.

Although these objects are intended for uniform buffers, they can still be used for other purposes e.g. SSBOs or even vertex attribute buffers by binding them manually before rendering or computing is performed.

Atomic Counter Buffers

Besides uniform buffers, there is also a wrapper class for atomic counter buffers derived from the general buffer class. This basically implies that it is used with the target `GL_ATOMIC_COUNTER_BUFFER`. Usage is hinted as `GL_DYNAMIC_COPY` because the application may specify initial counter states and read new values back from the graphics library. Shader programs will use atomic counter operations to access these buffers (see chapter 3). These built-in functions read the old value and write the modified one back to the buffer hence the `COPY` hint. Once again, these default properties do not limit the buffer to being used as source for atomic counters.

In OpenGL, atomic counters are always unsigned integers [34, ch. 7.7]. For this reason, the corresponding wrapper class is not a template but it is limited to this type.

Image Objects

As explained in chapter 3, image objects can be used to read from and write to textures. This is very useful for image processing as well as for storing general data. Image objects are not supported at the moment because more powerful and flexible shader storage buffers can be used which are more suitable for the use case of n-body and particle simulations.

Textures

Other important OpenGL objects are textures. Since the main focus of this thesis has been the implementation of a generic and GPGPU based engine, texturing and advanced visual effects have been neglected.

4.2.3. Shader Program Handling

Besides memory objects, shader programs are the most important concepts of OpenGL which are necessary for both, rendering and computing. The engine contains classes to abstract shader programs and to simplify their handling. To create and manage these, a class called `GPUProgramService` is used.

Shader Programs

The abstract `ShaderProgram` class encapsulates an OpenGL program and its attached shader objects and exposes corresponding API functions via its methods. It is a common class for graphics engines so it can be found in Ogre3D and is also discussed in [23, ch. 1], as well. Irrlicht3D does not have a dedicated shader program class.

When instantiated, an OpenGL program handle is generated. New shaders can be created and attached via `addNewShader(...)`. The `build()` routine is used to compile all attached shaders and link the program. The build status is observed and error messages are logged if building fails.

Part of the exposed API are methods to modify uniform variables, i.e. querying their locations and setting their values. For the latter case, function overloading is used to provide a unified method name for all types of uniform variables. Parameters of these functions are the name of a uniform variable to query its location implicitly and its new value.

To bind uniform, shader storage or atomic counter buffers to the program three methods are available: `bindUniformBuffer(...)`, `bindShaderStorageBuffer(...)` and `bindAtomicCounterBuffer(...)`. Arguments passed to this methods are the name of a buffer block or atomic counter uniform variable and a buffer object. The latter is bound to the appropriate target at the index that is queried from the layout `binding` qualifier of the buffer block or uniform which corresponds to the given name. This procedure makes attaching buffers more convenient since no layout bindings have to be kept in mind but rather unique block or variable names are used to bind buffers.

Previous to this implementation, a solution using `glUniformBlockBinding(...)` and `glShaderStorageBlockBinding(...)` was examined. These two functions allow to overwrite the layout `binding` qualifier defined in the shader program. To set up buffers, they are not bound to the hard coded binding but to an explicitly specified index. Most of the time, this is done by the render and compute systems covered shortly. Afterwards, the mentioned API calls adjust the `binding` of the buffer block to reference the correct index.

The main drawback of this becomes visible as soon as buffers have to be bound manually which requires an unused index. A system to keep track of currently active bindings has to be introduced to make it possible to determine an empty index. Since this is just a hint, any index can be used which could eventually break the bindings applied by the render and compute system and result in unintended behaviour.

Furthermore, binding points can change with every render and dispatch call (e.g. if two separate particle systems are drawn or updated) which makes caching the binding points and avoiding round trips to the driver impossible.

Eventually, this method cannot be used for atomic counters since they are not managed via buffer blocks and consequently no `glAtomicCounterBlockBinding(...)` function is available. The final implementation chosen for the engine shares the same code for shader storage, uniform and atomic counter buffers.

OpenGL shader programs can have subroutines that are selected by the application on runtime [34, ch. 7.9]. To wrap the more complex functions for subroutine management, an API similar to the binding of buffers is used:

```
bool ShaderProgram::selectSubroutine(
                            GLenum shaderStrage,
                            std::string selectorName,
                            std::string subroutineName);
```

The first parameter is the shader stage for which the subroutine should be selected. The `selectorName` is the name of the subroutine uniform variable which stores the selected routine. The last parameter is the name of the subroutine which should be used.

The selected routines are cached inside the shader program class because OpenGL requires all subroutines to be set at once with a single function call. When a program was successfully built, fail-safe defaults are queried to initialise the cache which pre-empts OpenGL errors.

To activate subroutines, `ShaderProgram::activateSubroutines()` has to be executed. When calling `glUseProgram(...)`, all selected subroutines are reset to default values [34, ch. 7] and have to be re-activated. This is done by the render and compute systems which invokes `activateSubroutines()` automatically.

As mentioned at the beginning, the shader program is an abstract class and thus cannot be instantiated. Instead, objects of its specialised sub-classes `RenderProgram` and `ComputeProgram` are used.

The `RenderProgram` is used for rendering and thus able to query vertex attribute locations. Shaders of all programmable pipeline stages can be attached with only vertex and fragment shader being mandatory. When a tessellation control or evalua-

tion shader is linked into the program, a flag is set to true which can be queried by `usesTessellation()`. The size of the patches that are used for tessellation can be controlled by `setTessellationPatchSize(int patchSize)`.

The `ComputeProgram` on the other hand is used for compute shaders. Only one compute shader can be attached per program. An additional function that is called `ComputeProgram::getNumWorkItemsPerGroup()` is provided to query the number of work items per local work group.

GPUProgramService

Two of the main design goals lead to the introduction of the `GPUProgramService`: implicit resource management and reusable shader code. It corresponds to Irrlicht3D's `GPUProgrammingServices`. Ogre3D on the other hand contains a set of different managers to handle different kinds of shader programs.

To achieve the first goal, render and compute programs are only allowed to be created through this service. When destroyed, it releases and deletes all of its shader programs. In case of the simulation engine, only one `GPUProgramService` is available which is part of the `Engine`.

There is one API call to create render and compute programs respectively:

```
RenderProgram* createRenderProgram(
        const std::string& id,
        const std::string& vertSrcFile,
        const std::string& fragSrcFile,
        const std::string& tcsSrcFile = "",
        const std::string& tesSrcFile = "",
        const std::string& geoSrcFile = "");

ComputeProgram* createComputeProgram(
        const std::string& id,
        const std::string& computeSrcFile);
```

Both of them take an ID which uniquely identifies the shader program. This can be used in combination with `getComputeProgram(...)` and `getRenderProgram(...)` to access one of these programs without the need of storing and passing pointers. The remaining parameters are paths to files of the *virtual file system* (see below) containing the source code for a certain shader stage, e.g. `tesSrcFile` is a file containing a tessellation evaluation shader. As mentioned above, only vertex and fragment shaders are mandatory for render programs.

The second goal that is achieved by the `GPUProgramService` is the reusability of shader source code. A dedicated OpenGL extension `GL_ARB_shading_language_include` is provided for this purpose which introduces an `#include` directive [55]. This uses *named strings* to create some sort of Unix-like virtual file system which resides within the graphics driver.

The `GPUProgramService` has two methods to add single files or whole directories to the virtual file system:

```
bool GPUProgramService::addSourceFile(
                const std::string& sourceFilePath,
                const std::string& destination);

bool GPUProgramService::addSourceDirectory(
                const std::string& sourceDirectoryPath,
                const std::string& destination,
                bool recursive);
```

Parameters `sourceFilePath` and `sourceDirectoryPath` have to be a valid file or directory on the hard drive. The `destination` is the full path of the virtual file or directory to which the content is stored. The `recursive` flag controls whether sub-directories should be added as well. In both cases the names of added files and sub-directories are preserved as can be seen in Listing 4.3.

Listing 4.3: Creation of shader programs using the `GPUProgramService`.

```
1 // Get instance of the gpu program service.
2 GPUProgramService* gpuService =
3           Engine::getInstance()->getGPUProgramService();
4
5 // Load a directory of shader source files:
6 // ./res/shader/vertex.glsl
7 // ./res/shader/fragment.glsl
8 // ./res/shader/include/uniforms.glsl
9 gpuService->addSourceDirectory("./res/shader", "/myshaders");
10
11 // Now create a new render program using both files.
12 gpuService->createRenderProgram("renderer", "/myshaders/vertex.glsl",
       "/myshaders/fragment.glsl");
```

Shaders can include files residing in the virtual file system. Listing 4.4 demonstrates how the vertex shader includes a header containing uniform definitions (line 5).

To be able to use `#include`, the shader has to enable the OpenGL extension which is shown in line 3.

This process can be roughly compared to the *gcc* option `-I` which specifies a directory that is searched for include files. The only differences are the possibilities of renaming file paths and of adding single files to the virtual file system.

Similar to C++, inclusion guards should be used in GLSL header files to prevent any errors caused by multiply defined symbols.

There is no need to care about source code that is included unnecessarily. The GLSL compiler provided by the graphics driver uses highly efficient optimisations to purge unused code or variables. Since shader programs are only compiled once on application start, the additional overhead for the compiler can be ignored in most cases especially for simulations with small shaders.

Listing 4.4: Sample of a shader including a header file.

```
#version 430

#extension GL_ARB_shading_language_include : require

#include </myshaders/include/uniforms.glsl>

in vec4 vertexPosition;
void main()
{
    // Perform transformation using the model view projection matrix
    // defined in uniforms.glsl.
    gl_Position = u_ModelViewProjectionMatrix * vertexPosition;
}
```

4.2.4. GPU Abstraction

Ogre3D and Irrlicht3D use abstractions of graphics APIs like Direct3D and OpenGL to provide a common and reusable interface which is called `RenderSystem` and `VideoDriver` respectively. Implementations of this interface provide functions needed for rendering objects and modifying render states. This concept allows the render systems to be exchanged as needed.

These approaches inspired the abstraction of the GPU layer, i.e. rendering and computing. To do so, a `RenderSystem` is introduced which takes care of the rendering related OpenGL features and a `ComputeSystem` responsible for dispatching compute shaders. Since some methods are shared by both, a parent class `GPUSystem` was added.

In contrast to the examined game engines, the render and compute systems are tied to OpenGL. An additional abstraction layer is required if the underlying frameworks should be interchangeable. Since using a different library would withdraw the benefits gained by OpenGL (see chapter 3), this feature was not included.

The render and compute system classes are described in the following. The full rendering and update process is covered in section 4.3.

RenderSystem

On one hand, the render system initialises the OpenGL context and creates a render window. On the other hand it is used to render particle systems (see section 4.3) and to control render states like v-sync or depth testing.

Initialisation is issued by the engine when `Engine::init(...)` was called:

```
GLFWwindow* RenderSystem::init(int width, int height,
                bool fullscreen, bool debug);
```

The parameters correspond to those passed to the method of the engine. They set the width and height of the window and toggle full screen mode. The `debug` option controls

whether OpenGL should create a debug context or not. This also registers an error callback function to the graphics library (see subsection 4.4.1).

Since the GLFW-library is used, the created `GLFWwindow` is returned to the engine so it can handle mouse and keyboard inputs.

Similar to Irrlicht3D, a begin-end block is used to perform rendering. A call to `beginFrame()` tells the render system to start a new frame. Technically, this clears colour and depth buffers. After all draw calls have been issued, `endFrame()` is used to bring the render result to the screen, i.e. to swap back and front buffers.

At the moment, only a method to draw particle systems is supported by the render system. This process is described in detail in subsection 4.3.3.

In previous implementations, the render system was not directly accessible but only through the `drawAllParticleSystems()` command of the engine. This method calls the render system's function `beginFrame()`, sets the current view projection matrix retrieved by the camera, iterates over all particle systems, draws them using the render system and calls `endFrame()` eventually. To increase flexibility and to allow advanced visualisation (e.g. trajectories of particles), the API of the render system was exposed to the end user.

ComputeSystem

Whereas the render system is responsible for drawing, the compute system updates particle systems. The detailed process is covered in subsection 4.3.4. As described above, the render system creates the OpenGL context which is done as soon as the engine is initialised. For this reason, a separate initialisation of the compute system is not required.

Similar to rendering, the engine provides a method `updateAllParticleSystems()` which invokes the compute system for all particle systems.

4.2.5. Render Resources

When rendering an *entity* (i.e. a certain object in the scene or, in case of this thesis, a particle), there are two mandatory information required: the geometrical structure (*mesh*) and a description of how to render it (*material*). These *resources* are a common concept of 3D engines and thus can be found in Irrlicht3D and Ogre3D. This allows objects of the same geometry to be rendered in different ways or, vice versa, the same rendering technique to be applied to objects with different structure.

In Ogre3D, both of these are separate classes that are joined by an entity. On the other hand, Irrlicht3D uses the mesh class to store materials. When attaching a new entity to the scene, a new `MeshSceneNode` is instantiated which holds a reference to the mesh and makes a copy of the stored materials to avoid one scene node manipulating the materials of others. Since it provides a cleaner interface and does not perform implicit copying, the approach proposed by Ogre3D is used for the simulation engine.

The concepts of meshes and materials were added to the engine and are explained in the following.

Meshes

Meshes hold the geometrical information of one or more entities. The most important ones are vertex positions, vertex normals and uv-coordinates for texturing. If indexed drawing methods like `glDrawElements(...)` are used, the order of vertices have to be stored as well [34, ch. 10.5].

At the moment, only vertex positions (three-dimensional floating point vectors) and indexes (unsigned integers) are supported. They are stored in separate buffers that are initialised when the mesh is created. Since buffers are immutable, the full list of vertices and indexes have to be passed to the constructor:

```
Mesh(size_t vertexCount, glm::vec3* vertices,
     size_t indexCount, GLuint* indexes);
```

where `vertexCount` and `indexCount` store the number of vertices and indexes.

OpenGL uses *VAOs* (*vertex array objects*) which "represent a collection of sets of vertex attributes" [34, p. 28] which are passed to vertex shaders on rendering. At the moment, these objects are stored in the mesh class but should be moved to the particle system in a future version (see chapter 6).

Materials

Unlike meshes which store geometrical data, materials describe how to render these objects. Usually, they store a render program and textures that are applied to the surface of the entity. Ogre3D goes even further and also stores depth buffer settings and culling modes in materials.

In case of the simulation engine, only the render program and a render type is stored since textures are currently not supported and other states are managed by the render system itself.

The render type controls the primitive that is used to render an object. Currently supported types are `NP_RT_POINTS` and `NP_RT_TRIANGLES` to render points and triangles respectively. If tessellation is enabled, this type is ignored and rendering is done via patches as described in subsection 4.3.3.

A new material is constructed via

```
Material(RenderProgram& renderProgram,
         render_types renderType);
```

with `renderType` defaulting to `NP_RT_TRIAGNLES`.

4.2.6. Resource Management

One of the requirements of the engine was the implicit management of resources, e.g. meshes, materials, buffers and particle systems. This is achieved by constructing resource through the interface of a *resource managers* rather than instantiating objects directly. This makes it possible to share meshes and materials between entities, as well.

Irrlicht3D has only a manager for meshes which is called `MeshCache` since materials are managed by the `MeshSceneNode` classes as explained in the previous section. Ogre3D uses dedicated `MeshManager` and `MaterialManager` classes that are responsible for this task. These concepts have been adapted to the simulation engine and are explained below.

The `MeshManager` provides an interface similar to the constructor of the mesh class:

```
const Mesh* createMesh(std::string id,
        size_t vertexCount, glm::vec3* vertices,
        size_t indexCount, GLuint* indices);
```

Additional to parameters of the constructor, the ID of the mesh is passed which is used to retrieve its pointer through the manager. No new mesh is created if the ID is already assigned.

The `MaterialManager` on the other hand supports two ways of creating new materials:

```
const Material* createMaterial(const std::string& id,
          RenderProgram* renderProgram,
          render_types renderType);

const Material* createMaterial(const std::string& id,
          const std::string& renderProgramId,
          render_types renderType);
```

The first version takes a reference to a render program, whereas the second version only takes an ID by which the render program is retrieved via the `GPUProgramService` which can be more convenient in some cases. The `renderType` is directly passed to the material constructor and defaults to `NP_RT_TRIANGLES`. Once again, the `id` is used to identify the material and query its pointer through the material manager.

To provide visual feedback as soon as possible, both managers provide hard-coded default resources so no custom meshes and materials have to be created or loaded when beginning with a new simulation. Two default meshes are available: a single point and an icosahedron. The default material performs basic vertex and fragment shading including view port transformation but no model transformation, i.e. all particles are rendered at the origin of the simulation. The engine built-in shader library covered in section 4.5 also contains a basic vertex and fragment shader.

Ogre3D also uses resource managers for memory objects and shader programs. In the simulation engine, buffers are intended to be created only by meshes or particle systems which makes a separate manager redundant. A major drawback of this implementation is that uniform and atomic counter buffers cannot be shared between two particle systems since when destroyed, both of them would attempt to delete the shared buffers. Shader programs on the other hand are managed by the `GPUProgramService` as described in subsection 4.2.3.

Another solution for providing implicitly managed resources would be the usage of shared pointers. But since meshes, materials and shader programs are constructed before they are referenced by any other class, this technique cannot be used for these

kind of resources. A hybrid solution which uses shared pointers for buffers and actions but managers for all other objects could solve this issue. Implementing and profiling possible performance impacts of this idea is not covered by this thesis but regarded as future work.

4.3. Particle System

Contrary to other solutions presented in section 1.2, the simulation engine uses a generic interface to create particle systems. This section covers the creation of particle systems, the definition of particle attributes and behaviour as well as the render and update processes which visualise and run the simulation.

4.3.1. Creation

A new, plain particle system is created by calling

```
ParticleSystem* Engine::createParticleSystem(
                    int particleCount,
                    const Mesh& mesh,
                    const Material& material);
```

The `particleCount` limits the maximum number of particles within a system. This is necessary because buffers for attributes are created with a fixed size that cannot be changed afterwards (see subsection 4.2.2). On the other hand, n-body systems rarely change the number of simulated bodies. Mesh and material are mandatory arguments as well, since they are required for rendering.

The engine can be seen as a resource manager for particle systems. When it is terminated via `terminate()`, all systems are automatically deleted. To manually destroy a particle system, it has to be passed to `Engine::deleteParticleSystem(...)` rather than calling `delete`.

4.3.2. Definition

A particle system is defined by the attributes and behaviour of its particles. Both can be configured to satisfy the needs of a certain simulation. Besides per particle data, uniform and atomic counter buffers can be added to the system, as well.

Particle Attributes

A new particle attribute can be added by the template method

```
Buffer<T>* addParticleAttribute(
                const std::string& name,
                GLenum glType,
                int glBaseSize);
```

A new shader storage buffer of type `T` is created to store values for all particles, i.e. it has an item count that equals the number of particles in the system. Parameter `name` must be unique among all attributes to make them identifiable. When rendering or computing, the buffer is bound to the shader storage block of the same name (see subsection 4.3.3 and 4.3.4).

The last two parameters, `glType` and `glBaseSize`, are only required if the buffer is used as a source for vertex attributes as explained in subsection 4.2.2.

By returning the pointer of the newly created buffer, its initial data can be specified.

Another idea to define particle attributes could be the use of a templated particle system class. Attributes would be defined in a structure that is used as template argument. Only one buffer would be required to store the structured data. However, this approach has several drawbacks. For example, always the full buffer has to be bound and no distinction between render and compute relevant ones can be made by future versions to improve performance. Besides that, defining vertex attribute pointers into the buffer requires complex offset and stride rules. Eventually, future versions of the engine could provide a scriptable interface which reads particle definitions from a configuration file. By using a structure, this would not be possible since the compiler has to know this definition to create the particle system class. All of these drawbacks led to the dismissal of the idea of a templated particle system.

Uniform Buffers

A new uniform buffer of type `T` can be created and attached to the particle system via

```
UniformBuffer<T>* addUniformBuffer(
    const std::string& name,
    int itemCount);
```

Parameter `name` must be uniquely among all uniform buffers and is used to bind the buffer to a uniform buffer block with corresponding name. In contrast to particle attributes, the `itemCount` has to be specified explicitly because the size cannot be guessed from the number of particles in the system.

Similar to the creation of attributes, the returned pointer can be used to initialise the buffer.

Atomic Counters

Buffers for atomic counters can be added with

```
AtomicCounterBuffer* addAtomicCounterBuffer(
    const std::string& name,
    size_t itemCount);
```

The `itemCount` tells the number of counters to be stored in the buffer. Just as attribute and uniform buffers, parameter `name` is used to identify atomic counter buffers among themselves and to bind them to shader programs.

Once again, the pointer returned by this method can be used to initialise the buffer.

Since atomic buffers are not organised in blocks (unlike shader storage and uniform buffers) they are bound to uniform variables. To access all counters inside a buffer with `itemCount` greater than one, either the offset rules described in [38, ch. 4.4] or arrays of atomic counters have to be used inside the shader:

```
layout (binding = 0) uniform atomic_uint myCounters[4];
```

Particle Behaviour

The behaviour of particles of a simulation is defined by `Action` objects which hold compute programs containing the logic of how attributes are updated [20]. A system can have several actions that are stored in a list and executed sequentially when it is updated. They can be seen as building blocks which can be chained to model complex effects. According to [20], the overhead of dispatching several actions in a row is reduced by GPU-side L2 caches. Since L2 and L1 caches of a GTX 660 are limited to 512kB and a maximum of 48kB respectively (see subsection 2.1.1), this may not be true for large particle systems but examining the caching is out of the scope of this thesis.

The concept of actions makes it possible to reuse compute shaders for various different simulations. For example, a motion integration can be implemented and attached to different systems to allow its particles to travel through the scene. On the other hand, this does not prohibit using only a single action with one optimised shader whereby the overhead created by the action list is minimal. This allows to balance reusability and performance according to the use case of the simulation.

To create a new action and append it to the action list of the particle system,

```
Action* appendAction(ComputeProgram& computeProgram);
```

is called. The action is returned so it is possible to hook into the pre and post update signals which are covered in subsection 4.3.5.

The global and local work group sizes do not have to be passed to the action manually because they are respectively queried from the compute program and calculated dynamically as explained in subsection 4.3.4.

4.3.3. Render Process

One main goal of the simulation engine is the visualisation of n-body simulations. As described in subsection 4.2.4, the RenderSystem is responsible for drawing and provides a `drawParticleSystem(...)` method.

In this section, the detailed render process of the `drawParticleSystem(..)` method of the render system is described which is comprised of following steps:

1. Bind and set up the material
2. Bind the mesh and set up vertex attributes
3. Bind particle attribute, uniform and atomic counter buffers

4. Emit pre render signal
5. Activate shader subroutines
6. Set up tessellation
7. Issue the OpenGL draw call
8. Emit post render signal
9. Restore OpenGL state, unbind buffers, mesh and material

In the first step, the material of the particle system is bound, i.e. the corresponding render program is activated and values for built-in uniforms are assigned. At the moment, the only built-in uniform variable is the view projection matrix which is set by the engine when `drawAllParticleSystems()` is executed. If particle systems are rendered manually, this matrix has to be set explicitly to the render system by calling `setViewProjectionMatrix(...)`.

The next step activates the vertex array object of the mesh. Its index buffer is bound to the `GL_ELEMENT_ARRAY_BUFFER` target to serve as indexes for the draw call. The vertex buffer is set up to provide data for the built-in `np_in_position` vertex shader input variable (see section 4.5). This is accomplished by the method

```
bool RenderSystem::setVertexAttribute(
        const std::string& attributeName,
        Buffer<T>* buffer,
        bool instanced);
```

which can be used to set up custom vertex attributes as well. Its parameters are the name of a vertex shader input variable, a buffer object and an optional flag that controls the instance divisor. At first, the location of the variable is queried and the type of the buffer is checked. If the location could not be found or if the buffer does not provide a proper `GlTypeInfo` (see subsection 4.2.2), the attribute cannot be set. If both have valid values, the buffer is bound to `GL_ARRAY_BUFFER`, the vertex attribute pointer is set up and activated. If the `instanced` flag is true, an attribute divisor of value one is initialised. After completion, the buffer is unbound again since it is no longer needed.

In step 3 of the render process, particle attributes, uniforms and atomic counter buffers are bound by using the methods provided by the shader program class.

An OpenGL implementation does not have to support shader storage buffer blocks for vertex shaders [34, p. 561]. If unavailable, the particle attributes cannot be bound automatically. The only workaround to pass per particle attributes is to use manually created instanced vertex attributes as described above which can be done inside the pre render signal covered in subsection 4.3.5.

After the set up of vertex attributes, the initialisation is now almost complete and so the pre render signal is emitted which allows to inject custom set up code (see subsection 4.3.5).

Since these event callbacks are able to change subroutines, these are not activated until step 5 which is performed by calling `ShaderProgram::activateSubroutines()`.

The next step is only required if tessellation is used, i.e. a tessellation control or evaluation shader was linked into the render program and therefore its `usesTessellation()` method evaluates to true. The drawing mode passed to the draw call is changed to `GL_PATCHES` and the size of the patch is set up according to the value provided by the material.

Now, as the initialisation has completed, the actual drawing call is dispatched via `glDrawElementsInstanced(...)`. The render mode is determined by the material or defaults to patches if tessellation is used as explained in the last paragraph. The number of vertices and the type of indexes correspond to the size and type of the index buffer of the mesh. The number of instances equals the number of particles in the system.

Each shader invocation must be able to determine the data of the particle it renders. For example, each vertex shader has to know the position of the particle (not to be confused with the vertex position) to transform vertices to the correct location. This can be achieved by the GLSL built-in variable `gl_InstanceID`. Similar to the IDs of compute shader invocations, this variable is unique per rendered instance [34, ch. 11.1] and can serve as an index into the particle attribute buffers.

After the rendering was issued (but before it completed since it is executed asynchronous to the CPU), the post rendering signal is emitted and its registered callbacks are invoked (see subsection 4.3.5).

Eventually, the render system has to clean up the OpenGL state. This is required, since material and mesh can be shared by many particle systems which not necessarily have the same definition. For example, buffer bindings which are not overwritten could result in wrong data passed to shaders which could lead to unintended behaviour. To ensure a consistent state, the set up steps have to be reversed. On one hand, all buffers and the render program are unbound. On the other hand, since the vertex array object is part of the mesh, all vertex attribute pointers have to be disabled as well as the VAO has to be unbound. This could be improved by moving the VAO to the particle system which is regarded as future work.

4.3.4. Update Process

Whereas the render system is used to draw particle systems, the compute system is responsible for updating its particles. For this purpose, a `updateParticleSystem(...)` method is provided which is covered in this section.

A particle system can contain multiple actions that define how the particles behave. When updated, the action list is processed sequentially and for each action the following steps are performed by the compute system:

1. Bind the compute program
2. Bind particle attribute, uniform and atomic counter buffers
3. Emit pre update signal

4. Activate shader subroutines
5. Dispatch the compute program
6. Apply synchronisation
7. Emit post update signal

At first, the compute program of the currently processed action is activated and particle attribute, uniform and atomic counter buffers are bound the same way as for rendering. Afterwards, the pre update signal is emitted and shader subroutines are activated.

Now that initialisation completed, the compute program is dispatched in step 5 by calling `glDispatchCompute(...)`. This requires the local work group size to be passed as parameters which is calculated by the simple equation:

$$\text{localWorkGroupSize} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \left\lceil \frac{\text{particleCount}}{\text{workItemsPerGroup}} \right\rceil \\ 1 \\ 1 \end{pmatrix}$$

This assumes that the global work group is only one-dimensional. It is still allowed to define compute shaders with three-dimensional local work group sizes but the global work group size y and z are never greater than one.

This simplification avoids setting the number of work items and local work groups manually and removes a source of errors when both, application and shader program, use hard coded values that diverge. If this limitation becomes too restrictive for some applications, a method to overwrite this default behaviour can be added easily in a future release.

The mayor drawback of this method is that maybe more shader invocations are dispatched than necessary which is caused by rounding up the number of work groups. If the IDs of work items are used to determine an offset to access a buffer object, the shader has to perform manual bounds checks since "out-of-bounds access produce[s] undefined behavior"[38, ch. 5.11].

As described in chapter 3, synchronisation is needed to avoid race conditions. For this reason, a memory barrier for shader storage buffers and atomic counters is issued implicitly after the dispatch of the compute program. If required, additional synchronisation can be manually applied by hooking into the post update signal (see subsection 4.3.5) which is emitted in the next step.

Similar to the render system, the compute program and all particle attribute, uniform and atomic counter buffers have to be unbound to preserve a consistent OpenGL state and to avoid consequential errors. This only needs to be done after all actions of a particle system have been processed, i.e. at the end of the `updateParticleSystem(...)` method, since for each action all buffers are rebound and the compute program binding is overwritten as well.

4.3.5. Particle System Events

At some points earlier, signals and events were mentioned which allow to hook into the render and update process of a particle system. The user can provide multiple callback functions for each signal to perform additional work that is not covered by the engine, for example setting up custom buffer bindings and uniform variables or applying additional synchronisation.

Irrlicht3D has a similar technique provided by the `IShaderConstantSetCallBack` interface which is implemented by the user and attached to a material [16]. Every time the material is used for rendering, the callback is invoked to set up uniforms of the shader program.

There are four different signals available in the simulation engine:

- `preRenderSignal`
- `postRenderSignal`
- `preUpdateSignal`
- `postUpdateSignal`

Render signals are part of a particle system and are triggered when a particle system is rendered. On the other hand, update signals belong to an action and are emitted when the particles are updated.

Listing 4.5 shows a small example of how callbacks can be connected to these signals.

Listing 4.5: Registering callbacks to particle system events.

```
1 //...
2
3 void preRenderCallback(ParticleSystem* pSys, RenderSystem*
      renderSystem)
4 {
5     std::cout << "Pre render callback\n";
6 }
7
8 void postRenderCallback(ParticleSystem* pSys, RenderSystem*
      renderSystem)
9 {
10    std::cout << "Post render callback\n";
11 }
12
13 void preUpdateCallback(ParticleSystem* psys, const ComputeSystem*
      computeSystem)
14 {
15    std::cout << "Pre update callback\n";
16 }
17
18 void postUpdateCallback(ParticleSystem* psys, const ComputeSystem*
      computeSystem)
```

```
19 {
20     std::cout << "Post update callback\n" << std::endl;
21 }
22
23 void main()
24 {
25     // ...
26
27     ParticleSystem* pSys = engine->createParticleSystem(1024, mesh,
       material);
28     Action* action = pSys->appendAction(*computeProgram);
29
30     pSys->preRenderSignal.connect(preRenderCallback);
31     pSys->postRenderSignal.connect(postRenderCallback);
32
33     action->preUpdateSignal.connect(preUpdateCallback);
34     action->postUpdateSignal.connect(postUpdateCallback);
35
36     // ...
37 }
```

The parameters passed to these callbacks are the currently processed particle system and the render or compute system. This gives the user all relevant objects without storing them in global variables or needing long-winded object traversals beginning at the level of the engine.

4.4. Debugging and Profiling

Other important topics when developing a simulation are debugging and profiling which are covered in this section.

4.4.1. Debugging

Debugging is a very important task especially for graphical applications. Most of the time, when an error occurs, the screen stays black and shows no sign of what went wrong. To overcome this issue and to provide a better debug environment, the simulation engine logs errors that arise in the engine itself and in OpenGL.

This is done by a lightweight `Logger` class which prints information and errors to the standard output and error devices.

Internal errors are prefixed with the class name where the failure was detected and a detailed error description. For example, when the build of a shader program failed, the error log produced by the driver is queried and printed to provide a hint about the source of the error.

Another way of error reporting is achieved by the debug reporting system of OpenGL [34, ch. 20]. This is enabled by initialising the engine in debug mode by passing true as the fourth parameter to `Engine::init(...)`. An OpenGL debug context is created and

a callback to log debug messages is passed to `glDebugMessageCallback(...)`. Once again, the logger is used to print these messages.

To control the log level, `Engine::setGlDebugLevel(...)` is called with one of the available OpenGL severity levels (e.g. `GL_DEBUG_SEVERITY_HIGH`) which disables all messages with a lower one. This internally modifies the debug behaviour of the graphics library which can be adjusted by the `glDebugMessageControl(...)` API function.

Some of the messages generated by OpenGL can be too generic and their source cannot be found easily. Since synchronous error reporting is used, these messages are created as soon as the problem occurs. By executing the simulation with a debugger and setting a break point inside the debug message callback, the call stack and consequently the exact method invocation that caused the report can be detected.

4.4.2. Profiling

Another important point is the measurement of performance to find bottlenecks within the engine and any simulation. For this reason, two *clocks* are provided to measure time on CPU as well as on GPU.

Both work in a start-stop manner. The clock is started before the task that is profiled and stopped after its completion. `Clock::getElapsedTime()` is called to get the time that passed by. In some cases, the result is not available as soon as the clock was stopped. For this reason, the method `timeAvailable()` is used to determine if the result can be read safely.

For time measurements on the CPU, the `CPUClock` is available which uses GLFW methods to query the system time on `start()` and `stop()` [52]. The elapsed time is calculated by subtracting start time from stop time. Because it executes on the CPU, the time is available as soon as the clock was stopped, so there is no need to check `timeAvailable()`.

The `GPUClock` is not as straightforward because OpenGL executes asynchronously to the CPU. For example, draw or dispatch calls immediately return to the caller before the operation completed. To get the correct result, OpenGL timer queries are used to measure the time of methods that execute on the graphics card [34, ch. 4.3].

When `GPUClock::start()` is called, a new `glBeginQuery(...)` command is injected into the OpenGL pipeline to start the time query. To stop the clock, `glEndQuery(...)` is enqueued which means that it is not actually executed until the GPU control flow reached this command. To ensure that the query already terminated, `timeAvailable()` must be called and eventually evaluate to true.

One limitation of this implementation is, that only one query can be active at a time. If multiple `GPUClocks` are started simultaneously, an OpenGL error will be created.

4.5. Engine built-in Shader Library

The simulation engine ships with a couple of GLSL files that can be used by simulations. This demonstrates how a library for render and compute shaders can be created and

distributed. The library is placed in a directory that contains the source code files. As usual, they can be loaded with `GPUProgramService::addSourceDirectory(...)`.

All headers provide C++ style inclusion guard to be save of errors related to multiply defined symbols.

Most of the samples included in this thesis use some of the provided utilities to reduce code duplication.

The following two sections provide a short overview of the standard functionality provided by the shader library of the simulation engine.

4.5.1. Render Utilities

A vertex and a fragment shader are provided in files "standard-vertex-shader.glsl" and "standard-fragment-shader.glsl" respectively. They can be used to quickly achieve visual output for new simulations by performing standard model and view port transformation for vertices with a fragment colour defaulting to orange.

As mentioned previously, the engine automatically provides some input to the vertex shader. These are vertex attributes that are declared in "vertex-inputs.glsl". At the moment, the only variable available (`np_in_position`) is used to pass vertex positions of the mesh to the vertex shader in form of three-dimensional floating point vectors. New ones (e.g. normals and texture coordinates) can be added easily by appending their definitions to the header.

Another render related header is "uniforms.glsl". The current file only provides the uniform matrix `np_viewProjectionMatrix` which stores the view projection matrix of the camera.

Both, "vertex-inputs.glsl" and "uniforms.gls" are used to implement the previously mentioned standard vertex shader.

The header "point-sprite-utils.glsl" provides a function that allows point sprites to be rendered as circles.

4.5.2. Compute Utilities

Beyond rendering, there are headers that provide functions used to update particle attributes via compute shaders.

A function to compute a *global invocation index* is put into "globalinvocationindex.glsl" which can be compared to the `gl_LocalInvocationIndex` of OpenGL. It can be used to index into buffers and arrays if both, global and local work group, are one-dimensional.

The header "gravity-utils.glsl" contains the gravitational constant [56] and methods to calculate the acceleration that is applied on a body based on Newton's law of universal gravitation. All gravitation based example simulations created for this thesis rely on this header.

4.6. Review of Design Goals

In section 1.1, several requirements have been introduced which influenced the architecture of the engine. In this section these are reviewed and evaluated if they have been accomplished.

Real time rendering and GPU computing. This is accomplished by using OpenGL 4.3 as render as well as compute backend. This allows to access data manipulated by the update process within the render shaders without the need of copying any buffers.

Generic design. The engine provides a particle system that can be customised in means of attributes as well as behaviour. Actions, meshes and materials allow to create building blocks which can be reused in many scenarios.

Vendor independency and cross-platform. By choosing OpenGL as render and compute framework, simulations can be executed on all devices supporting version 4.3 of this library. Being an open standard, OpenGL is not limited to a certain hardware manufacturer or operating system.

To keep hardware and driver requirements as low as possible, only features of OpenGL 4.3 are used since versions 4.4 and 4.5 are not as widely supported.

Also, excessive use of OpenGL extensions was avoided. Not being a mandatory part of the specifications, drivers do not have to implement all of them which could restrict the number of supported systems. The only one used, `GL_ARB_shading_language_include`, can be easily removed by providing a client side implementation incorporated to the engine.

All third-party libraries that have been used are cross-platform and vendor independent and do not limit the portability of the software.

Reusable GPU code. By using the `GL_ARB_shading_language_include` extension, an `#include` directive is available that enables the inclusion of source code that is used frequently. This is demonstrated by the sample applications of this thesis which use engine built-in libraries for common tasks.

Flexibility and extensibility. Although the engine provides a high-level API, all of its low-level components like the render and compute systems can be accessed as well. It is also possible to directly issue OpenGL calls if some functionality is not encapsulated as desired.

By providing render and update events, the extensibility is increased. The user is able to hook into the render and update processes to perform special tasks and to manipulate the behaviour of the engine.

Easy to learn and to use. This goal cannot be evaluated as easily as the other ones and will only show when future users provide feedback and suggestions on how to improve usability.

Object oriented OpenGL abstraction. OpenGL is based on C but its view on resources like buffers is object oriented since the specification speaks of "objects" which simplified the design of an object oriented API that is provided by the engine.

Implicit resource management. All resources created by the engine are managed implicitly. Shader programs are handled by the `GPUProgramService` whereas particle systems are managed by the engine itself. Buffers are created through the mesh or particle system classes which are responsible for their destruction. Resource managers are used to create meshes and materials.

When the engine is terminated, all memory consumed by its objects is released so the user does not have to care about object lifecycles.

5. Gravitational N-Body Systems

Whereas the last chapter covered the developed simulation engine, this chapter presents two sample applications which have been created to verify the implementation and to demonstrate the capabilities of the engine. Both are gravitational n-body systems in which particles influence each other according to Newton's law of universal gravitation. The first one uses randomly distributed particles to demonstrate different techniques of integrating the equation of motion over time as well as optimisation techniques involving shared memory. The second one simulates our solar system as a specific example of a scientific use case producing quantitative results.

5.1. Physical and numerical Basics

Gravitational simulations consist of two fundamental concepts: calculation of the acceleration applied on a particle by gravitational forces and update of its position according to the equation of motion.

Calculation of Attractions

Gravitational n-body systems are based on Newton's law of universal gravity which states that each body within the system exerts an attracting force on all other bodies which is proportional to the masses of the particles and inverse proportional to the square of their distance.

The vector of the force exerted by body j on body i is calculated by the following formula:

$$\vec{f}_{ij} = G\frac{m_i m_j}{\|\vec{r}_{ij}\|^2} \cdot \frac{\vec{r}_{ij}}{\|\vec{r}_{ij}\|} \tag{5.1}$$

where the left multiplicand is its magnitude and the right one defines its direction. G is the *gravitational constant*, m_i and m_j are the masses of the bodies and $\vec{r}_{ij}$ is the vector from i to j calculated as $\vec{r}_{ij} = \vec{x}_j - \vec{x}_i$.

To obtain the total force acting on body i, all forces exerted by all $n-1$ interaction partners have to be summed up:

$$\vec{f}_i = \sum_{\substack{j=1 \\ j\neq i}}^{n} G\frac{m_i m_j \vec{r}_{ij}}{\|\vec{r}_{ij}\|^3} = Gm_i \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{m_j \vec{r}_{ij}}{\|\vec{r}_{ij}\|^3} \tag{5.2}$$

For the simulation, the effect of this force, i.e. the acceleration influencing the velocity of particle i, is of main interest. Due to the definition of $F = ma$, equation 5.2 can be rewritten to calculate the acceleration a_i:

$$\vec{a}_i = \frac{\vec{f}_i}{m_i} = G \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j \vec{r}_{ij}}{\|\vec{r}_{ij}\|^3} \tag{5.3}$$

Sometimes, this term is also referred to as "the force per unit mass, $\mathbf{F}_i$" [57].

When the distance between two bodies becomes very small, the acceleration grows infinitely. Since this is not desired for the numerical integration of gravitational simulations, a *softening factor* ϵ is introduced to equation 5.3 which must be greater than zero [13]:

$$\vec{a}_i \approx G \sum_{j=1}^{n} \frac{m_j \vec{r}_{ij}}{\left(\|\vec{r}_{ij}\|^2 + \epsilon^2\right)^{3/2}} \tag{5.4}$$

This does not only prevent the acceleration growing out of bounds but also removes the constraint $j \neq i$ from the sum in equation 5.3 which was needed to avoid a division by zero. The theoretical interaction of a body with itself is negated since the distance $\vec{r}_{ii}$ and thus the resulting force equals zero.

Integration of the Equation of Motion

To update the position of a particle, its equation of motion has to be integrated numerically which can be denoted as an *ODE* (*ordinary differential equation*) of second order [58]. There are different methods to integrate these equations two of which are covered in the following paragraphs.

The first one evaluated is the second order explicit *Runge-Kutta method* also known as *improved Euler* or *Heun's method* [58] which is defined as

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{2}(k_1 + k_2)\Delta t \\ k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \Delta t, y_n + \Delta t k_1) \end{aligned} \tag{5.5}$$

Let $f(t, y)$ be the function of the particles velocity and y the particles position. Using $\vec{p}$ for the position and $\vec{v}$ for the velocity one can simplify the equation to

$$\vec{p}_{n+1} = \vec{p}_n + \frac{1}{2}(\vec{v}_n + \vec{v}_{n+1})\Delta t \tag{5.6}$$

with an initial velocity of $\vec{v}_0$ and a time step of Δt. The new velocity $\vec{v}_{n+1}$ can be calculated using the acceleration obtained by equation 5.4:

$$\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n \Delta t \tag{5.7}$$

To avoid the recalculation of all integration steps on update, $\vec{v}_{n+1}$ has to be saved so it can serve as $\vec{v}_n$ for equation 5.6 in the next iteration.

An alternative approach for the integration of the equation of motion is provided by the *Verlet method* [59]. Instead of calculating the velocity, it is implicitly retrieved by storing the positions at t and $t - h$:

$$\vec{r}_i(t+h) = -\vec{r}_i(t-h) + 2\vec{r}_i(t) + \sum_{j \neq i} \vec{f}(r_{ij}(t))h^2 \tag{5.8}$$

where t equals the current time and h is the time step Δt. Once again, the vector $\vec{f}$ is the force per unit mass [57] which is equal to $\vec{a}_i$ in equation 5.4.

Adapted to the integration of the equation of motion, this results in

$$\vec{p}_{n+1} = -\vec{p}_{n-1} + 2\vec{p}_n + \vec{a}_n \Delta t^2 \tag{5.9}$$

to update a particles position. The acceleration $\vec{a}_n$ is calculated according to equation 5.4.

5.2. The Common Gravity Sample

The common gravity example application demonstrates how to simulate a gravitational, collisionless n-body system using Euler and Verlet integration. A method of optimising memory access is implemented as well. It also includes a benchmark utility which is explained in section 6.1.

To visualise the particles, circular point sprites are rendered. Their colour is used to encode different properties, e.g. the velocity vector and speed of the particle or its distance to the origin of the scene. These render modes are selected as shader subroutines and can be switched at runtime. Transformation is accomplished by adding the current particle position to the vertex coordinate followed by multiplication with the view projection matrix. The size of the point sprites is calculated relative to the distance of the particle to the camera position.

The goal of this application is to evaluate the computational performance of the engine but not the physically correctness of the computations. For this reason, arbitrary values for positions, masses and velocities of the particles have been chosen rather than real quantities. A validation of the correctness of the gravitational functions is provided by the solar system sample discussed in section 5.3.

Figure 5.1 shows the simulation of 65k particles. The colour represents the speed of each particle with slow ones painted blue and faster ones painted red.

Implementation

The particles are defined with two four-dimensional floating point vectors. The first one stores current positions in its first three components and masses in its fourth element. The second attribute saves either the velocities (Euler) or positions (Verlet) of the last

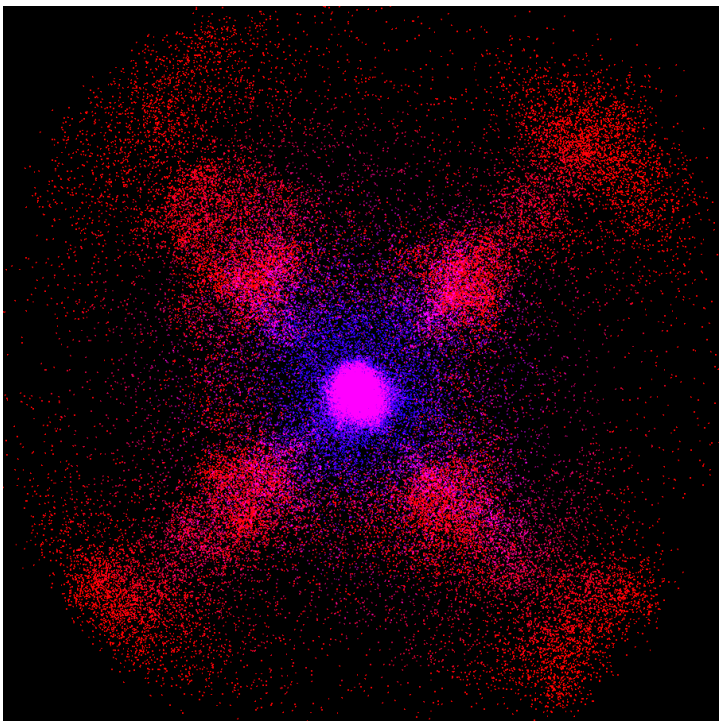


Figure 5.1.: Gravitational simulation of 65k particles. Slow particles are blue whereas fast ones are red.

integration step with the fourth component being unused. To update the particles, a single action is executed. Its compute shader determines the acceleration and recalculates the position for each particle.

A separate compute shader was implemented per integration method and optimisation technique. This avoids time that otherwise has to be spend on control flow statements but introduces a lot of code duplication. Since the application is also used for benchmarking, this disadvantage was accepted.

Since the acceleration is always calculated using equation 5.4, the engine built-in shader library provides a header containing gravitational utilities (as mentioned in section 4.5) including the function

```
vec3 npCalcAcceleration(
    in vec3 attractedPosition,
    in vec3 attractorPosition,
    in float attractorMass,
    in float softeningFactor);
```

which returns a three-dimensional acceleration vector. Parameter `attractedPosition` is the position of the attracted particle, i.e. the one that is updated by a compute shader invocation. Arguments `attractorPosition` and `attractorMass` are the position and mass of the attracting body whereas the `softeningFactor` is equivalent to ϵ in equation 5.4.

Since only one particle-particle interaction is calculated, this method has to be invoked with all partner particles and results have to be summed up to obtain the vector $\vec{a}_i$.

As can be seen in equation 5.4, the gravitational constant can be factorised which optimises the runtime. For this reason, `npCalcAcceleration(...)` does *not* take this constant into account which saves $3n^2$ *FLOPs* (*floating point operations*) per particle system update. For this specific simulation, the gravitational constant is ignored since, in contrast to the solar system simulation, no physically correct scope was chosen for the values of position, initial velocities and masses.

After the acceleration has been computed, the attributes of the particle are updated. The value of Δt is passed to the compute shaders as a uniform variable `timeStep` that can be dynamically increased or decreased at runtime.

In case of Euler integration, position and velocity are recalculated via equations 5.6 and 5.7 and written to the appropriate buffer.

On the other hand, the Verlet method calculates the new position using equation 5.9 and moves the old position into the second buffer. Two different ways of performing the copy operation have been implemented. The first one is a shader based swap operation that copies the current position into the second attribute and then stores the new position into the first buffer. The second method is implemented on the client side by calling

```
bool ParticleSystem::swapParticleAttributes(
    const std::string& firstAttributeName,
    const std::string& secondAttributeName);
```

in a post update callback of the action. This method swaps the attribute buffers identified by the names passed as arguments. Internally, only two pointers are interchanged which keeps this operation cheap.

In Listing 5.1, a simplified compute shader implementation is presented to exemplarily demonstrate the update process using Euler integration.

No synchronisation is required when storing the updated attributes (line 28 et seq.) because accessing shader storage buffers is incoherent which means changes are not visible to other shader invocations [38].

As explained in subsection 4.3.4, the compute system may dispatch more work items than necessary. To avoid out-of-bounds access of buffers, each shader invocation checks if its index is less or equal to the particle count (line 8) that is provided through a uniform variable. All work items that are used to fill up the last local work group immediately return.

Listing 5.1: Sample implementation of a compute shader to update gravitational n-body systems using Euler integration.

```
// Include headers...
// Definition of position and velocity buffers...

layout (local_size_x = 1024) in;
void main()
{
  // Return if this invocation is only used to fill up the last local
    work group.
  if(gl_GlobalInvocationID.x >= particleCount)
    return;

  vec3 position     = positions[gl_GlobalInvocationID.x].xyz;
  vec3 oldVelocity  = velocities[gl_GlobalInvocationID.x].xyz;
  vec3 acceleration = vec3(0.0, 0.0, 0.0);

  // Iterate over all particles
  for(uint i = 0; i < particleCount; ++i)
    acceleration += npCalcAcceleration(
      position,          // position of particle being updated
      positions[i].xyz, // position of interaction partner
      positions[i].w,   // mass of interaction partner
      0.1);

  // Update velocity and position
  vec3 newVelocity = oldVelocity + acceleration * timeStep;
  position += 0.5 * (oldVelocity + newVelocity) * timeStep;

  // Store updates attributes
  positions[gl_GlobalInvocationID.x].xyz = position;
  velocities[gl_GlobalInvocationID.x].xyz = newVelocity;
}
```

Optimisation

There are many techniques to optimise code that is executed on the GPU. For example [31] and [60] introduce optimisations for CUDA and OpenCL which can be mapped partially to the scope of OpenGL compute shaders.

Most important and easiest to accomplish, an appropriate local work group size has to be selected. This parameter should be set to keep as many multiprocessors and compute cores as possible busy to maximize GPU occupancy [31]. Chapter 6 evaluates the impact on compute performance.

Another common method to improve memory throughput uses shared memory for memory locations that are read by multiple invocations of a local work group [13]. Buffers are stored in main memory and thus reading is slow whereas shared memory resides on the chip of a multi processor which can be accessed faster. To reduce the number of main memory transactions, variables are copied to shared memory. For all

subsequent accesses, invocations now use the shared memory to read their values.

As can be seen in Listing 5.1, each shader invocation has to read the position and mass of all interaction partners. Listing 5.2 demonstrates how this access can be improved by shared memory. The performance improvement gained by this optimisation is examined in chapter 6.

Listing 5.2: Optimised version of Listing 5.1 using shared memory.

```
#define tileSize 1024
shared vec4 shPositions[tileSize];

layout (local_size_x = 1024) in;
void main()
{
  // Usual set up code...

  // Iterate over all tiles.
  for(uint i = 0, currentTile = 0;
    i < particleCount;
    i += tileSize, ++currentTile)
  {
    // Index into the buffer where the tile begins.
    uint tileBegin = currentTile * tileSize;

    // Bounds check required if last tile is not fully populated.
    if((tileBegin + gl_LocalInvocationIndex) < particleCount)
    {
      // Populate shared variable
      shPositions[gl_LocalInvocationIndex] =
        positions[tileBegin + gl_LocalInvocationIndex];
    }
    // Only continue if shared array fully populated.
    barrier();

    // For all particles in the tile, update acceleration.
    for(uint p = 0;
      p < tileSize && (tileBegin + p) < particleCount;
      ++p)
    {
      acceleration += npCalcAcceleration(particlePosition,
        shPositions[p].xyz, shPositions[p].w,
        softeningFactor);
    }

    // Synchronise for next iteration.
    barrier();
  }

  // Use calculated acceleration to integrate particle motion...
}
```

The particles are partitioned into *tiles* of fixed size `tileSize` which, in this example,

is equal to the `gl_WorkGroupSize.x`. An array of shared memory of this size is used to temporarily store the positions of all particles residing in the tile currently processed. The OpenGL specification requests the available shared memory to be at least 32768 bytes [34, p. 566] which is enough to store 8192 4-dimensional, single precision floating point numbers.

To calculate all particle-particle interactions, the loop in line 10 iterates over all tiles. At first, each shader invocation writes one value in this array to initialise it. To ensure that the array was fully populated, a control flow barrier is executed. No shared memory barrier is required, since "[s]hared variables are implicitly coherent" [38, ch. 4.3.8]. Afterwards, all work items iterate over the shared memory block and calculate the force for the first tile, i.e. the first `tileSize` particles. After the iteration completed, the local work group is synchronised again to make sure that the content of the shared array is not needed any more.

This loop is repeated until all tiles and thus all particle-particle interactions have been evaluated. Now, each work item continues to integrate the equation of motion just as usual.

The boundary checks in line 18 and 29 have to be performed to ensure proper loading and calculation if the last tile is not fully populated.

5.3. The Solar System Sample

The application introduced in the last section targets the computational performance of the engine. To verify the physical correctness of the engine built-in gravity shader utilities and to demonstrate a scientific use case, the solar system example was implemented. It simulates the nine planets of our solar system and the dwarf planet Pluto orbiting around the sun which is fixed at the origin of the coordinate system.

The planets and the sun are initialised with the data listed in Table C.3. They are aligned along the x-axis of the coordinate system using the distance from the sun as offset. The orbital velocity serves as their initial velocity directed along the y-axis.

To ensure accuracy, double precision floating point buffers are used to store positions and velocities.

The planets are rendered as icosahedra which are tessellated to create a spherical shape [61]. Whereas the size of the meshes are chosen arbitrarily to make them visible, the distance from the sun is true to scale of $1 : 10^8$. To scale the planet, vertex coordinates are multiplied by a constant factor.

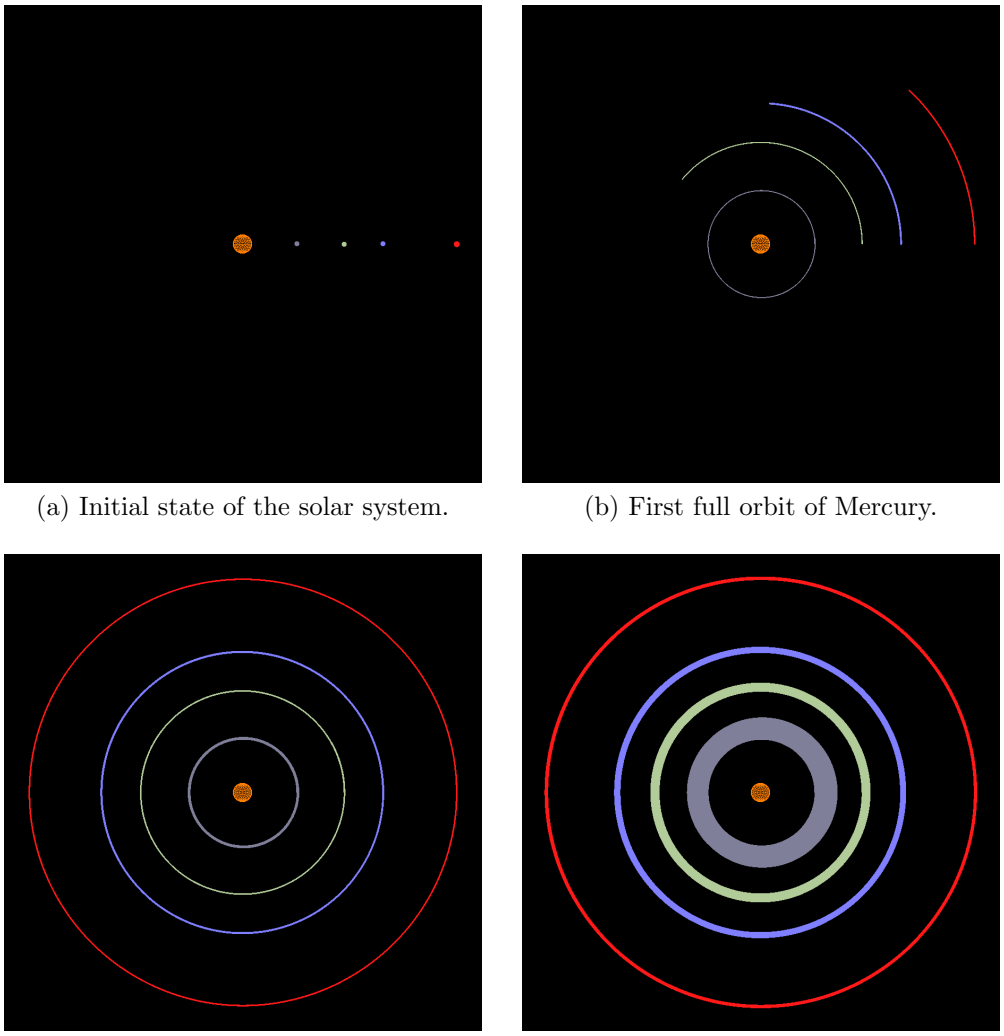
Transforming the vertices of the mesh is performed similar to the common gravity sample by adding the particle position to the scaled vertex coordinate. Since the tessellation process generates a high number of new vertices, performing this operation on all of them could limit performance. An additional action that calculates the combined model view projection matrix on a per particle basis could improve runtime.

The update process is similar to the gravity application using Euler integration. No shared memory optimisation has been implemented since the number of particles is too small to achieve improvements. The time step was initially set to 900 seconds.

The simulation is updated in each frame. To avoid the refresh rate of the screen limiting the speed of the application, vertical synchronisation can be disabled at runtime.

A simple method of drawing the trajectories of the planets was implemented by preventing the frame buffer from being cleared, i.e. `RenderSystem::beginFrame()` is not called. To avoid graphical artefacts, this mode can be toggled on runtime after the camera was moved to the desired location.

To evaluate the accuracy of the simulation, Figure 5.2 shows the accumulated trajectories of the inner planets Mercury, Venus, Earth and Mars at different times which gives a qualitative visualisation of the system's stability. As one can see, the planets are slowly drifting away from the sun. This is clearly visible when the orbits of Mercury in Figure 5.2(b) (one full revolution) and Figure 5.2(d) (approximately 117 revolutions) are compared.



(a) Initial state of the solar system.

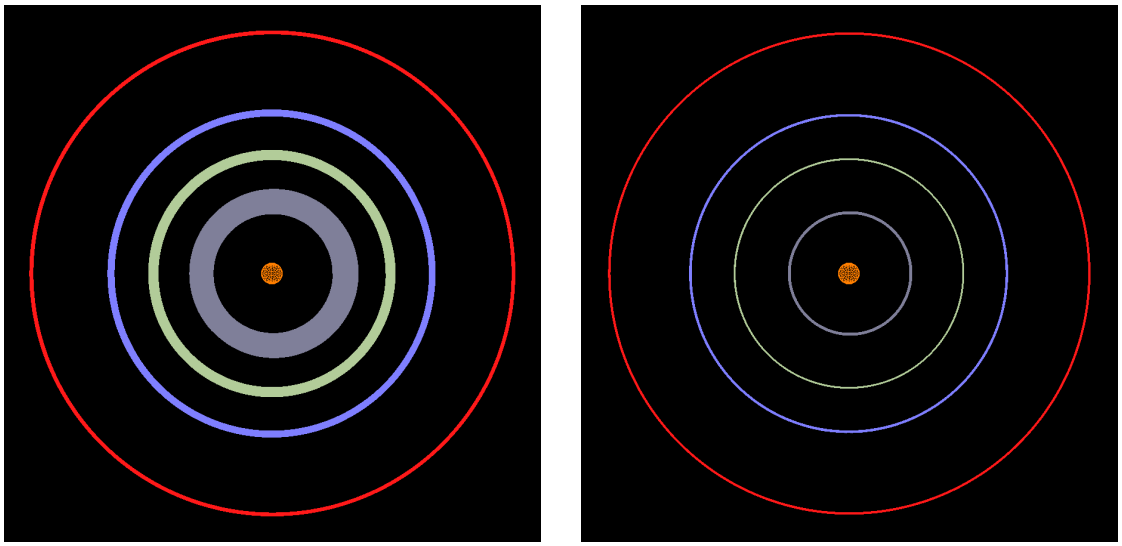
(b) First full orbit of Mercury.

(c) All planets completed at least one orbit.

(d) Solar system after one million updates.

Figure 5.2.: The inner planets of the solar system simulation at different time steps.

The main source of this error is the step size which determines the granularity of the integration of the equation of motion. Inaccuracies of each step are accumulated which results in the observed effect. Figure 5.3(a) shows the solar system after one million updates using the initial time step of 900 seconds. In contrast Figure 5.3(b) presents the system after 2×10^7 updates using a time step of 45 seconds. In both cases, the totally elapsed time is the same: $900s \cdot 1 \times 10^6 = 45s \cdot 2 \times 10^7 = 9 \times 10^8 s$.



(a) Step size = 900s, 1×10^6 updates

(b) Step size = 45s, 2×10^7 updates

Figure 5.3.: Comparison of updating the solar system sample with different time step sizes.

Additionally, Figure 5.4 presents a distant view of the simulation so the outer planets are visible as well. This proves that the system is still stable (i.e. no planet left its orbit) after 2×10^9 updates with a step size of 45 seconds which results in a simulation interval of approximately 2853 years.

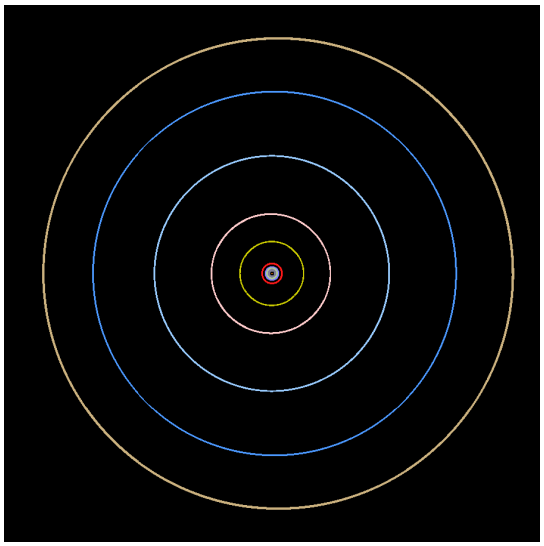


Figure 5.4.: Distant view of the solar system simulation including outer planets after 2×10^9 updates with step size of 45 seconds. (Lines thickened for better visibility.)

6. Performance

In this chapter, the performance of the engine is evaluated on a basic level. For this reason, benchmarks have been introduced which are explained at first. Afterwards, the general performance of the engine is examined on one hand and details of the render and update processes are covered on the other hand.

The benchmarks were executed on three different graphics cards: NVIDIA GeForce GTX660, an NVIDIA Quadro 4000 and an NVIDIA GeForce GTX780 Ti.

All performance data used for the charts of this chapter can be found as tables in Appendix D. Note that the horizontal axes of these charts are not linear due to the number of particles used for benchmarking (see below).

6.1. Benchmarks

Two benchmarks have been implemented to measure the performance of the render and compute process respectively. Times for set up, rendering or dispatching and clean up are evaluated separately by hooking into pre and post render and update signals. For each stage the time consumed by CPU and GPU are determined since both work asynchronously. For all measurements, the clocks introduced in subsection 4.4.2 have been used. Particle systems with different particle count are benchmarked to examine how the performance of the engine alters with increasing number of bodies.

Render Benchmark

To measure the time consumed by the render process, an OpenGL window of resolution 1440x900 is created in full-screen mode. Vertical synchronisation is disabled so rendering is performed at highest possible speed. Systems with 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072 and 262144 particles are rendered successively with 1000 iterations per particle count. The mesh used for the particles is an icosahedron which is not tessellated. For vertex and fragment shading, simple shader programs are used.

Compute Benchmark

To examine the performance of the compute process, the compute benchmark included in the gravity example can be used. It can be launched with different integration and optimisation techniques to further investigate different implementations. The tested particle counts are 2, 512, 1024, 4096, 8192, 16384, 32768 and 65536. For each configuration, 100 updates are performed.

In contrast to rendering, compute shaders for n-body simulations tend to be GPU-intensive. By using the shaders provided by the gravity simulation one can observe if the bottleneck of a simulation is caused by the CPU side update process or by the computations performed on the graphics hardware.

Additional to tracking times, an approximation of the performed *GFLOPS* (*giga floating point operations per second*) is calculated to evaluate the efficiency of the implemented update methods. To calculate the total FLOPs per update, the equations 6.1 and 6.2 are used for Euler and Verlet integration respectively where n refers to the number of particles. The constants are determined by counting the floating point operations of the shader code (cf. [13]).

$$\text{Euler-FLOPs} = (18 + 22n)n \tag{6.1}$$

$$\text{Verlet-FLOPs} = (21 + 22n)n \tag{6.2}$$

To calculate the floating point operations per second, these FLOP counts are divided by the time t spent by the GPU to perform one update of the particle system: FLOPS = FLOPs/t.

6.2. General Performance Observations

Figure 6.1 compares render and update processes to each other. This proves the expected runtime complexities which is linear for the render and quadratic for the update process. As a consequence, both have a notable impact on the overall performance of the engine. For a small number of particles, complex render shaders could exceed the time needed to update the system. On the other hand, recalculation of large amounts of particles is more likely to be more time-consuming than rendering. For these reasons, both of them are examined in detail in subsequent sections.

In Figure 6.2, the maximum *FPS* (*frames per second*) achieved when rendering are shown. As expected, they decline as particle count increases. With 262144 particles each rendered as an icosahedron which results in approximately 3.15 million vertices, the engine still performs with 122 FPS (GTX660) and 144 FPS (GTX780 Ti). Only the Quadro 4000 drops down to 45 FPS.

Figure 6.3 compares the performance of the update process of the gravity sample application using Verlet integration, shared memory optimisation and a local work group size of 512. These can be compared to the theoretical peak GFLOPS of 1881.5 (GTX660), 486.4 (Quadro4000) and 3977 (GTX780 Ti) [48]. All test systems reach approximately 36% of their maximum performance when updating 65k particles with this configuration.

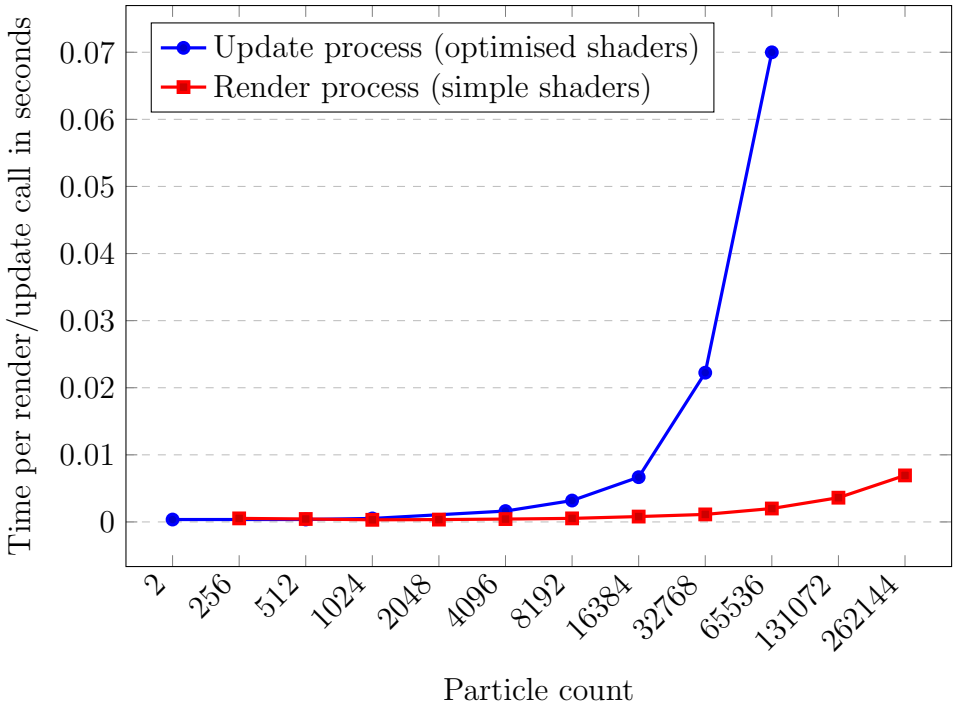


Figure 6.1.: Comparison of performance of render and compute process on an NVIDIA GeForce GTX780 Ti. Compute shader uses Verlet integration, shared memory, local work group size of 512.

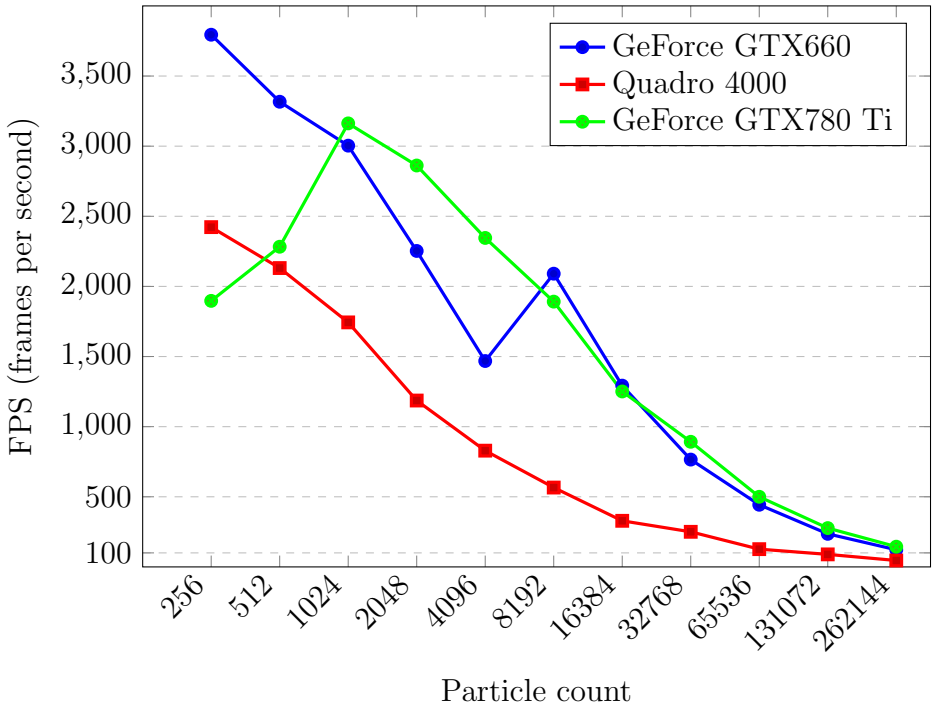


Figure 6.2.: Peak performance of the engine when rendering different number of particles.

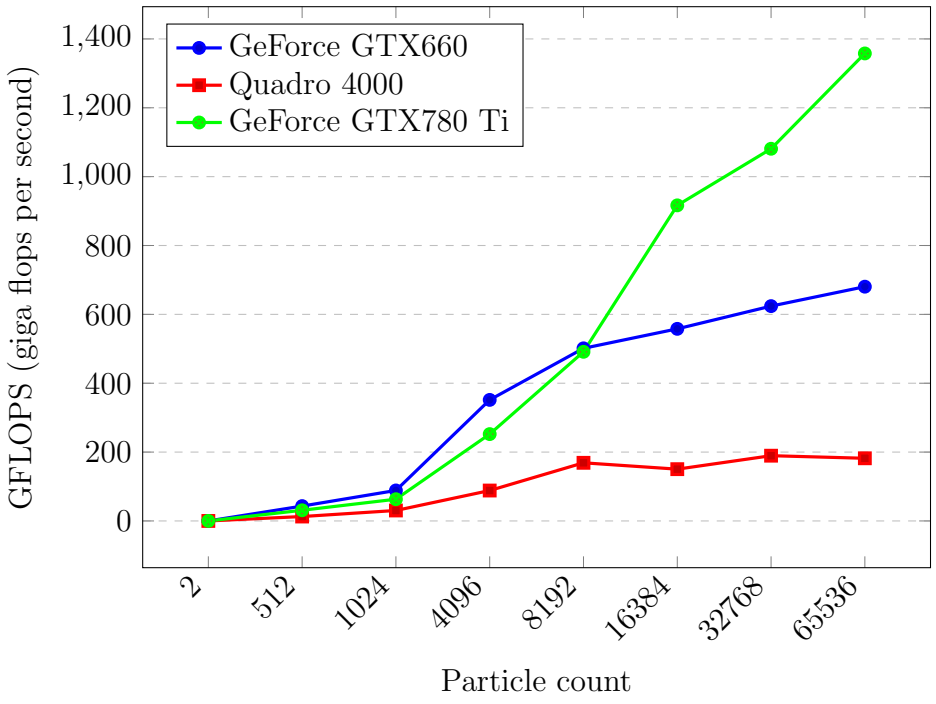


Figure 6.3.: Peak performance of the engine when updating the gravity sample application (Verlet integration, shared memory, local work group size of 512).

6.3. Render Process

Since the previous section only provides a rough idea, this section evaluates the performance of the render process in more detail. The results are presented in Figure 6.4.

As expected, set up and clean up times for both CPU and GPU are nearly constant whereas the time consumed by the GPU executing the actual render call increases as more and more particles are drawn.

On the other hand, also the rendering time on CPU increases which was not expected because no data copying or the like is performed. One source could be state changes which force the graphics driver to validate the context state when `glDraw(...)` is called [62]. Since the update process is mainly GPU bound (see next section), this could be caused by vertex attribute pointers which are rebound every frame. Moving vertex array objects from `Mesh` to `ParticleSystem` class could possibly avoid this overhead. Adjusting vertex attribute pointers would only be necessary after the particle system's material or mesh had been exchanged. Besides that, all redundant state changes should be eliminated to improve overall performance [63].

Nonetheless, further investigations are needed to properly identify the source for the observed performance decline.

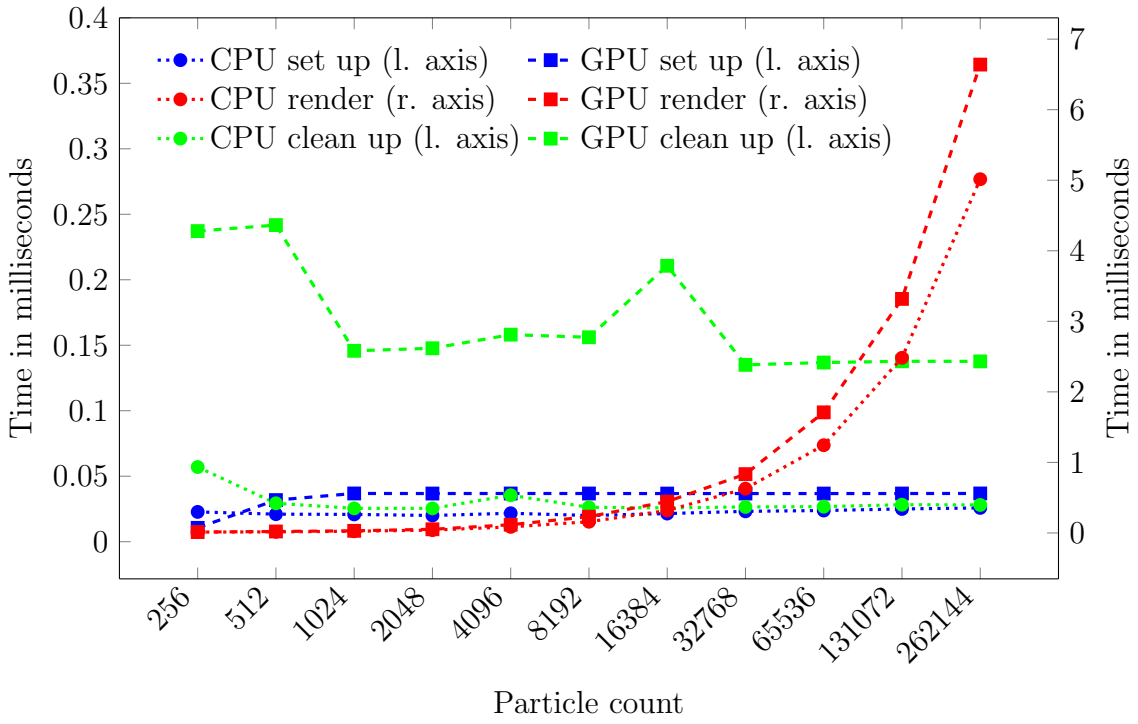


Figure 6.4.: Results of the render benchmark on an NVIDIA GeForce GTX780 Ti.

6.4. Update Process

Similar to the previous section, this one takes a detailed look at the performance of the update process. Figure 6.5 shows the time consumed by its substeps.

Times for set up and clean up for both CPU and GPU as well as the time spent on the CPU for updating are constant. This means that the update process is mainly GPU-bound which is desirable since the best performance improvements can be made by optimising compute shaders of a simulation.

As with rendering, further and more detailed profiling of the update process is required to minimise engine overhead like redundant state changes.

For compute shaders, however, one of the most important optimisations is the use of a proper local work group size to maximise occupancy of compute cores [31], [60]. The gravity benchmark was executed with different group sizes to demonstrate this effect in Figure 6.6. Best performance was achieved by using a local work group size of 192 for both, GeForce GTX660 and GeForce GTX780 Ti, and 1024 for the Quadro 4000. In case of the former two, this equals the number of CUDA cores of one of their multiprocessors [48] which means that all cores are utilised and no communication between processors limits performance [31]. When shared memory is used, the work group sizes with best performance are 512 (GTX660), 256 (GTX780 Ti) and 1024 (Quadro 4000). This means, that the configuration cannot be optimised in a generic way but has to be adjusted manually for different compute shaders, implementations and hardware.

As explained in section 5.2, shared memory optimisation was added to the compute

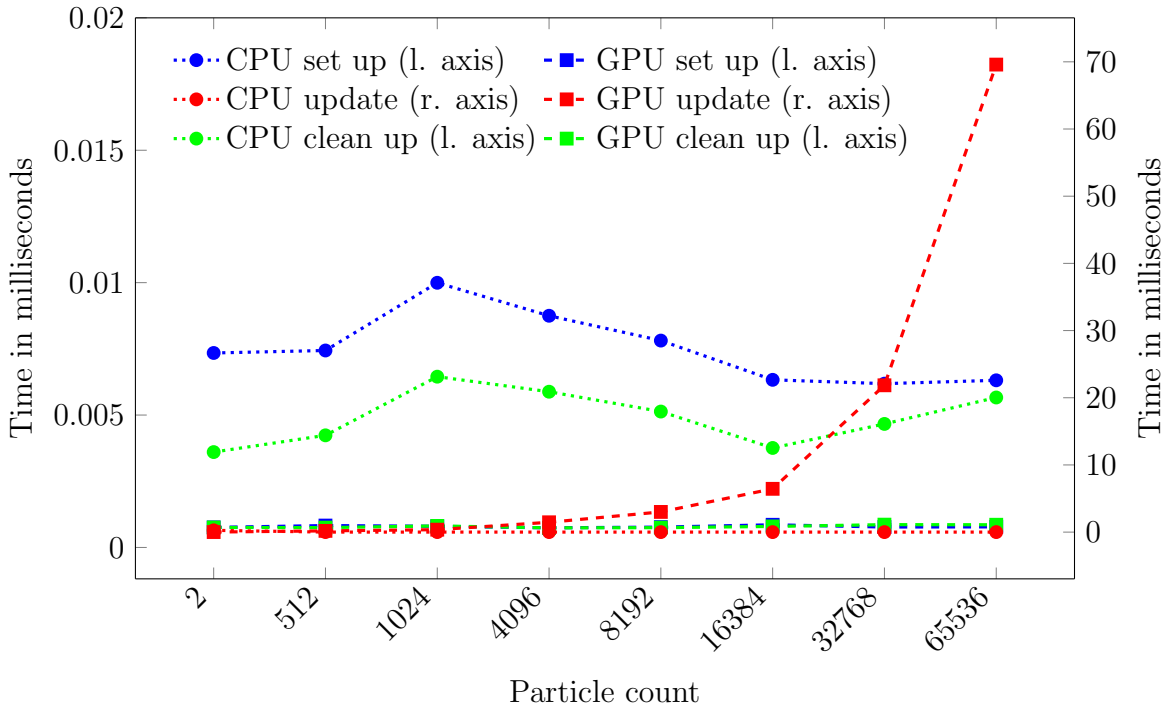


Figure 6.5.: Results of the update benchmark (Verlet integration, shared memory, local work group size of 512) on an NVIDIA GeForce GTX780 Ti.

shaders used by the gravity application. The improvements achieved on a GTX780 Ti using Verlet integration with a local work group size of 512 are depicted in Figure 6.7. As one can see, the performance gain increases proportional to the number of particles. In this specific test case, the compute shader using shared memory exceeds the GFLOPS of the unoptimised shader by approximately 30% when processing 65k particles.

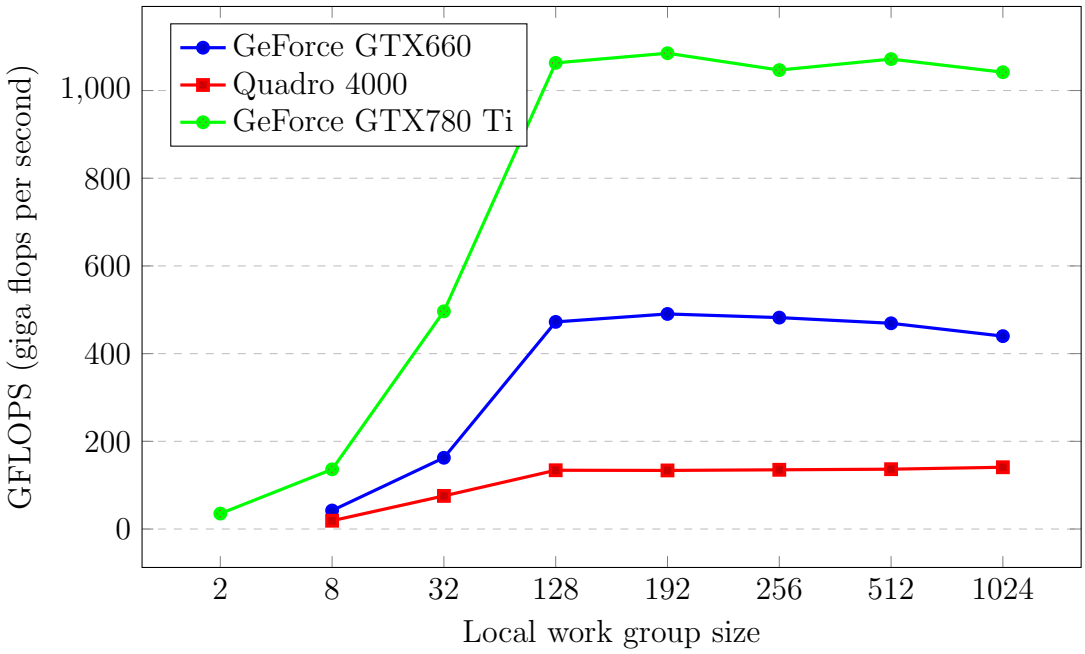


Figure 6.6.: Performance of different local work group sizes (Verlet integration, no shared memory, 65k particles).

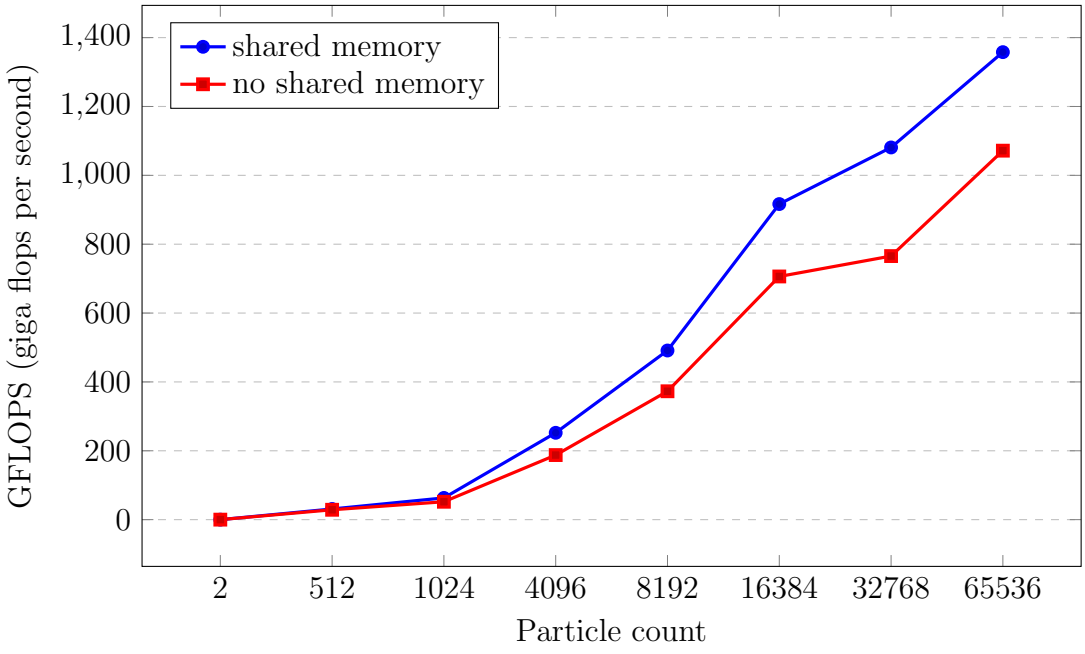


Figure 6.7.: Performance of shared memory optimisation on an NVIDIA GeForce GTX780 Ti (Verlet integration, local work group size of 512).

7. Conclusions

This bachelor thesis proposed a simulation engine for n-body and particle systems which uses the powerful parallel processing capabilities of the GPU. A wide variety of use cases can be covered ranging from visual effects to scientific applications like the solar system simulation provided as example.

Different GPU architectures and basics of GPGPU have been introduced. By comparing different frameworks, the most appropriate one (OpenGL) was determined. The basics of its compute shaders were examined to be able to integrate it into the simulation engine.

Except the learning curve and usability, which cannot be assessed objectively at this time, all initial design goals have been accomplished. The vendor independent and cross-platform engine is able to render and update simulations interactively. An object oriented API is provided which is simplified by not requiring manual memory management. Its generic and flexible design allows all kinds of simulations to be defined and missing features can be added easily. Source code can be shared between shader programs to improve reusability and to reduce code duplication.

The sample applications presented in this thesis give an outline of how the proposed software is used and what it is capable of. The results of render and compute benchmarks provide starting points to optimise the implementation of engine and application which are not remedied by this thesis.

Being a complex topic, many tasks could not be covered. The most platform limiting part of the engine is the OpenGL extension used to provide an include directive to GLSL which should be removed since it is not widely supported. Newer versions of OpenGL introduce features to create immutable buffers and methods making the bind-to-modify principle obsolete. The handling of vertex array objects should be improved as well as redundant state changes should be eliminated to increase overall performance.

One of the most promising improvements would be the introduction of a scripting interface. Materials and particle systems could be defined in configuration files whereas a scripting language (e.g. Lua) could be used to implement particle event callbacks. On one hand, this would remove unnecessary recompilations of the simulation. On the other hand, setting up shader programs and materials could be simplified a lot.

Mentioning all possible enhancements would be beyond the scope of this thesis and has to be covered by future applications and engine extensions.

Bibliography

[1] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3, Eighth Edition*, 8th ed. Addison-Wesley Professional, Mar. 2013.

[2] W. T. Reeves, "Particle systems - a technique for modeling a class of fuzzy objects", *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, Apr. 1983, ISSN: 0730-0301.

[3] S. Drone, "Real-time particle systems on the gpu in dynamic environments", in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07, San Diego, California: ACM, 2007, pp. 80–96, ISBN: 978-1-4503-1823-5.

[4] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications", pp. 154–159, 2003.

[5] H. H. Hu, D. D. Joseph, and M. J. Crochet, "Direct simulation of fluid particle motions", *Theoretical and Computational Fluid Dynamics*, vol. 3, no. 5, pp. 285–306, 1992.

[6] B. Xu and A. Yu, "Numerical simulation of the gas-solid flow in a fluidized bed by combining discrete particle method with computational fluid dynamics", *Chemical Engineering Science*, vol. 52, no. 16, pp. 2785–2809, 1997.

[7] J. Dawson, "Particle simulation of plasmas", *Rev. Mod. Phys.*, vol. 55, pp. 403–447, 2 Apr. 1983.

[8] W. W. Lee, "Gyrokinetic approach in particle simulation", *Physics of Fluids (1958-1988)*, vol. 26, no. 2, pp. 556–562, 1983.

[9] R. Overzier, G. Lemson, R. E. Angulo, E. Bertin, J. Blaizot, B. M. B. Henriques, G.-D. Marleau, and S. D. M. White, "The millennium run observatory: first light", *MNRAS*, no. 428, pp. 778–803, 2013.

[10] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars", *Nature*, vol. 435, pp. 629–636, 2005.

[11] S. Roettger, M. Schulz, W. Bartelheimer, and T. Ertlt, "Automotive soiling simulation based on massive particle tracing", in *Data Visualization 2001*, ser. Eurographics, D. Ebert, J. Favre, and R. Peikert, Eds., Springer Vienna, 2001, pp. 309–317.

[12] D. Helbing, I. Farkas, and T. Vicsek, "Simulating dynamical features of escape panic", *Nature*, vol. 407, pp. 487–490, 2000.

[13] L. Nyalnd, M. Harris, and J. Prins, "Fast n-body simulation with cuda", in *GPU Gems 3*, Addison-Wesley Professional, 2007, pp. 667–695.

[14] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm", *Nature*, vol. 324, pp. 446–449, 1986.

[15] Wikipedia, *List of game engines – wikipedia, the free encyclopedia*, 2014. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=List_of_game_engines&oldid=638548186` (visited on 12/21/2014).

[16] G. Nikolaus et al. (2014). Irrlicht3d source code. Version 1.8.1, [Online]. Available: `http://sourceforge.net/projects/irrlicht/` (visited on 09/30/2014).

[17] The OGRE Team. (2014). Ogre3d source code. Version 1.8, [Online]. Available: `https://bitbucket.org/sinbad/ogre/` (visited on 10/14/2014).

[18] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a gpu-based particle engine", in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '04, Grenoble, France: ACM, 2004, pp. 115–122, ISBN: 3-905673-15-0.

[19] L. Latta, "Building a million particle system", in *Game Developers Conf.*, San Francisco, CA, Mar. 2004.

[20] D. K. McAllister, "The design of an api for particle systems", Department of Computer Science, University of North Carolina at Chapel Hill, Tech. Rep., 2000.

[21] P. Krajcevski and J. Reppy, "A declarative api for particle systems", in *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, ser. PADL'11, Austin, TX, USA: Springer-Verlag, 2011, pp. 130–144, ISBN: 978-3-642-18377-5.

[22] J. Zink, M. Pettineo, and J. Hoxley, *Practical Rendering and Computation with Direct3D 11*. A K Peters/CRC Press, 2011.

[23] M. M. Movania, *OpenGL Development Cookbook*. Packt Publishing, Jun. 2013.

[24] M. Bailey, "Combining gpu data-parallel computing with opengl", in *ACM SIGGRAPH 2013 Courses*, ser. SIGGRAPH '13, Anaheim, California: ACM, 2013, 14:1–14:65, ISBN: 978-1-4503-2339-0.

[25] NVIDIA Corporation. (2014). Nvidia gameworks™ opengl samples, [Online]. Available: `https://developer.nvidia.com/gameworks-opengl-samples` (visited on 11/22/2014).

[26] J. Hunz, "The possibilities of compute shaders - an analysis", B.S. thesis, University of Koblenz-Landau, Campus Koblenz, 2013.

[27] Advanced Micro Devices, Inc., "Amd's graphics core next (gcn) architecture", Tech. Rep., Jun. 2012.

[28] NVIDIA Corporation, "Nvidia gf100", Whitepaper, 2010.

[29] ——, "Nvidia geforce gtx 680", Whitepaper, 2012.

[30] ——, "Nvidia geforce gtx 980", Whitepaper, 2014.

[31] ——, *Nvidia cuda c programming guide*, Aug. 2014.

[32] ——, "Nvidia's next generation cuda™ compute architecture: fermi™", Whitepaper, 2009.

[33] ——, "Nvidia's next generation cuda™ compute architecture: kepler™ gk110", Whitepaper, 2012.

[34] M. Segal and K. Akeley, "The opengl® graphics system: a specification. version 4.3 (core profile)", The Khronos Group Inc., Tech. Rep., Aug. 2012.

[35] Khronos OpenCL Working Group, "The opencl specification", Tech. Rep., Mar. 2014.

[36] ——, "The opencl c specification", Tech. Rep., Mar. 2014.

[37] T. Ni, "Direct compute – bring gpu computing to the mainstream", in *GPU Technology Conf.*, San Jose, CA, Oct. 2009.

[38] Khronos Group, "The opengl® shading language 4.3", Tech. Rep., Aug. 2012.

[39] T. Cohen, *Gpu utilization on gcn*, Dec. 2013.

[40] D. Göddeke, "GPGPU–Basic math tutorial", Fachbereich Mathematik, Universität Dortmund, Tech. Rep., Nov. 2005, Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300.

[41] Khronos Group. (2014). Opengl 4.5 reference pages, [Online]. Available: `https://www.opengl.org/sdk/docs/man/` (visited on 10/02/2014).

[42] NVIDIA Corporation, "Parallel thread execution isa", Tech. Rep., Aug. 2014.

[43] ——, (2014). Cuda gpus, [Online]. Available: `https://developer.nvidia.com/cuda-gpus` (visited on 10/03/2014).

[44] R. Banger and K. Bhattacharyya, *OpenCL Programming by Example*. Packet Publishing, 2013.

[45] J. Hirshon, *The khronos group releases opencl 1.0 specification*, online, Dec. 2008. [Online]. Available: `https://www.khronos.org/news/press/2008/12` (visited on 10/03/2014).

[46] ——, *Khronos finalizes opencl 2.0 specification for heterogeneous computing*, online, Nov. 2013. [Online]. Available: `https://www.khronos.org/news/press/khronos-finalizes-opencl-2.0-specification-for-heterogeneous-computing` (visited on 10/03/2013).

[47] Khronos Group. (2014). Arb_compute_shader, [Online]. Available: `https://www.opengl.org/registry/specs/ARB/compute_shader.txt` (visited on 11/21/2014).

[48] Wikipedia, *List of nvidia graphics processing units – wikipedia, the free encyclopedia*, 2012. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=632014753` (visited on 11/05/2014).

[49] ——, *List of amd graphics processing units – wikipedia, the free encyclopedia*, 2012. [Online]. Available: `http://en.wikipedia.org/w/index.php?title=List_of_AMD_graphics_processing_units&oldid=632378301` (visited on 11/05/2014).

[50] B. Fulgham and I. Gouy. (2014). The computer language benchmarks game, [Online]. Available: `http://benchmarksgame.alioth.debian.org/` (visited on 12/21/2014).

[51] M. Ikits and M. Magallon. (2015). The opengl extension wrangler library, [Online]. Available: `http://glew.sourceforge.net/` (visited on 01/03/2015).

[52] The GLFW Development Team. (2014). Glfw3 documentation, [Online]. Available: `http://www.glfw.org/docs/latest/` (visited on 11/27/2014).

[53] G-Truc Creation. (2015). Opengl mathematics, [Online]. Available: `http://glm.g-truc.net/0.9.6/index.html` (visited on 01/03/2015).

[54] B. Dawes. (2015). Boost filesystem library, [Online]. Available: `http://www.boost.org/doc/libs/1_57_0/libs/filesystem/doc/index.htm` (visited on 01/03/2015).

[55] Khronos Group. (2013). Arb_shading_language_include, [Online]. Available: `https://www.opengl.org/registry/specs/ARB/shading_language_include.txt` (visited on 11/21/2014).

[56] P. J. Mohr, B. N. Taylor, and D. B. Newell, "Codata recommended values of the fundamental physical constants: 2010", *Rev. Mod. Phys.*, vol. 84, pp. 1527–1605, 4 Nov. 2012.

[57] S. J. Aarseth, *Gravitational N-Body Simulations*. Cambridge University Press, 2003, ch. 1.

[58] J. G. de Jalón and E. Bayo, *Kinematic and Dynamic Simulation of Multibody Systems*, ser. Mechanical Engineering Series. Springer New York, 1994, ch. 7.

[59] L. Verlet, "Computer 'Experiments' on classical fluids. i. thermodynamical properties of lennard-jones molecules", *Phys. Rev.*, vol. 159, no. 1, pp. 98–103, Jul. 1967.

[60] Advanced Micro Devices, Inc. (2013). Opencl optimization guide, [Online]. Available: `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/` (visited on 12/12/2014).

[61] P. Rideout. (2010). Triangle tessellation with opengl 4.0, [Online]. Available: `http://prideout.net/blog/?p=48` (visited on 01/05/2015).

[62] OpenGL Discussion and Help Forum. (2014). Gldrawelements() blocking in cpu - takes 5ms to return, [Online]. Available: `https://www.opengl.org/discussion_boards/showthread.php/178871-glDrawElements()-blocking-in-CPU-takes-5ms-to-return` (visited on 12/30/2014).

[63] P. Cozzi and C. Riccio, *OpenGL Insights*. A K Peters / CRC Press, Jul. 2012.

[64] D. R. Williams. (2014). Planetary fact sheet - metric, [Online]. Available: `http://nssdc.gsfc.nasa.gov/planetary/factsheet/` (visited on 12/13/2014).

[65] ——, (2014). Sun fact sheet, [Online]. Available: `http://nssdc.gsfc.nasa.gov/planetary/factsheet/sunfact.html` (visited on 12/13/2014).

A. CD Content

The CD that can be found on the last page contains all work created in the course of this thesis. Figure A.1 outlines its file hierarchy. The main `README.md` file contains general information about the structure of the CD similar to this chapter.

The `engine` directory contains a readme file (outlining its structure; compilation and run instructions), the source code of the simulation engine and sample applications as well as documentation (API, diagrams).

The full set of benchmark results used by chapter 6 and Appendix D are stored in the `performance-data` directory.

Finally, directory `thesis` contains a PDF version of this thesis and its LaTeX source code.

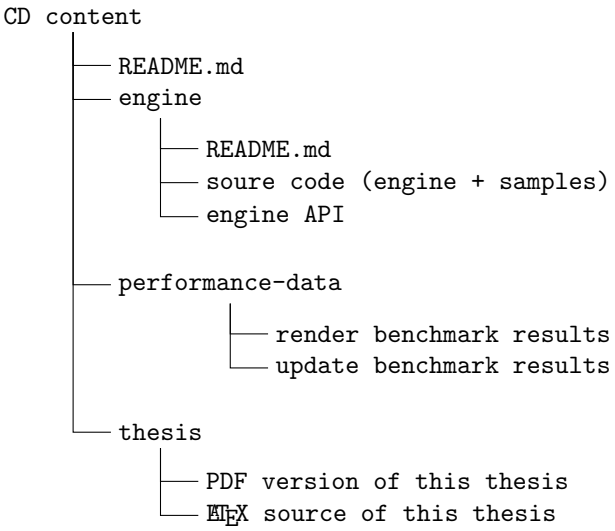


Figure A.1.: File hierarchy of the CD contents.

B. GPU Architecture Diagrams

This chapter contains diagrams of GPU architectures.

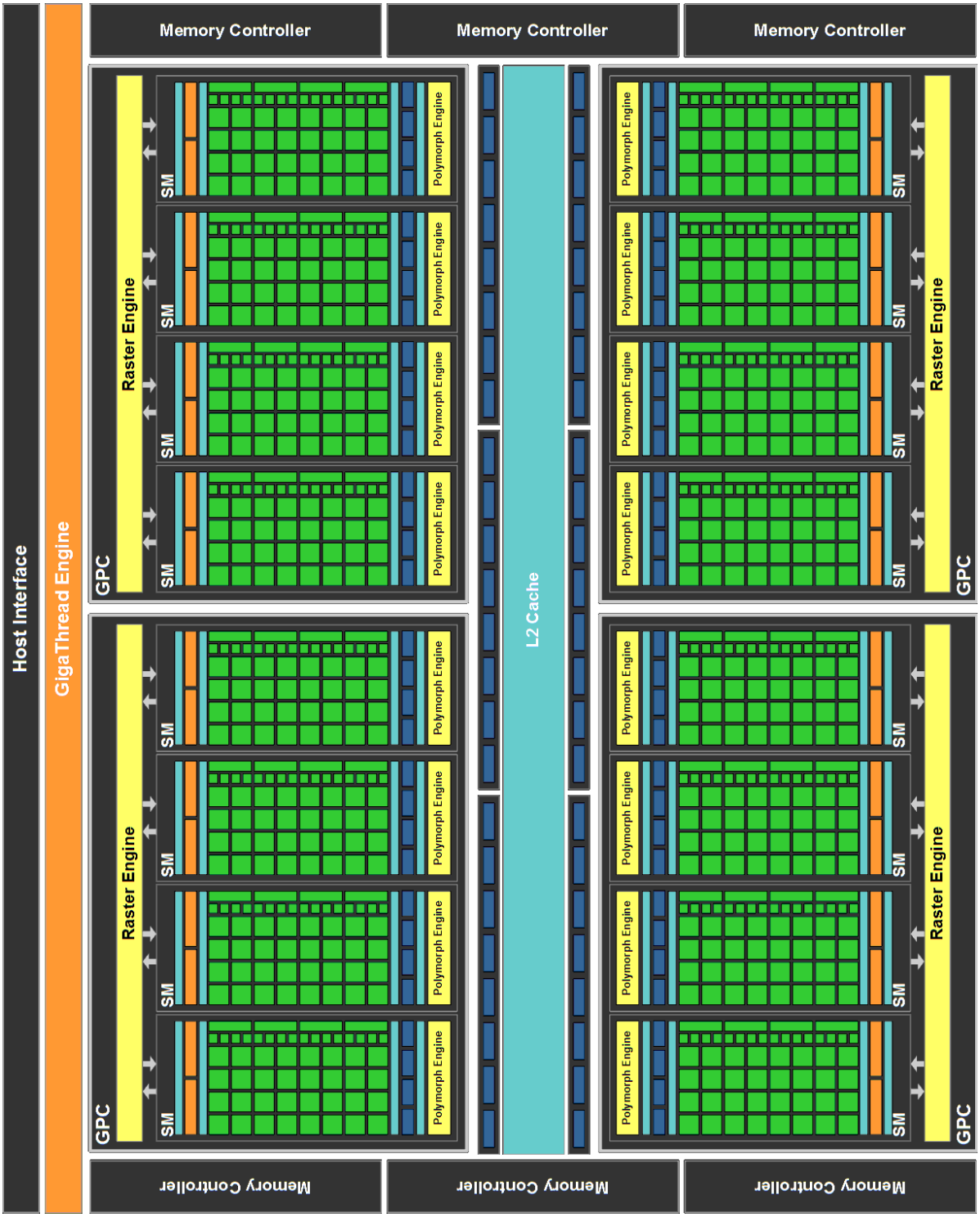


Figure B.1.: Block diagram of the NVIDA GF100 [28]

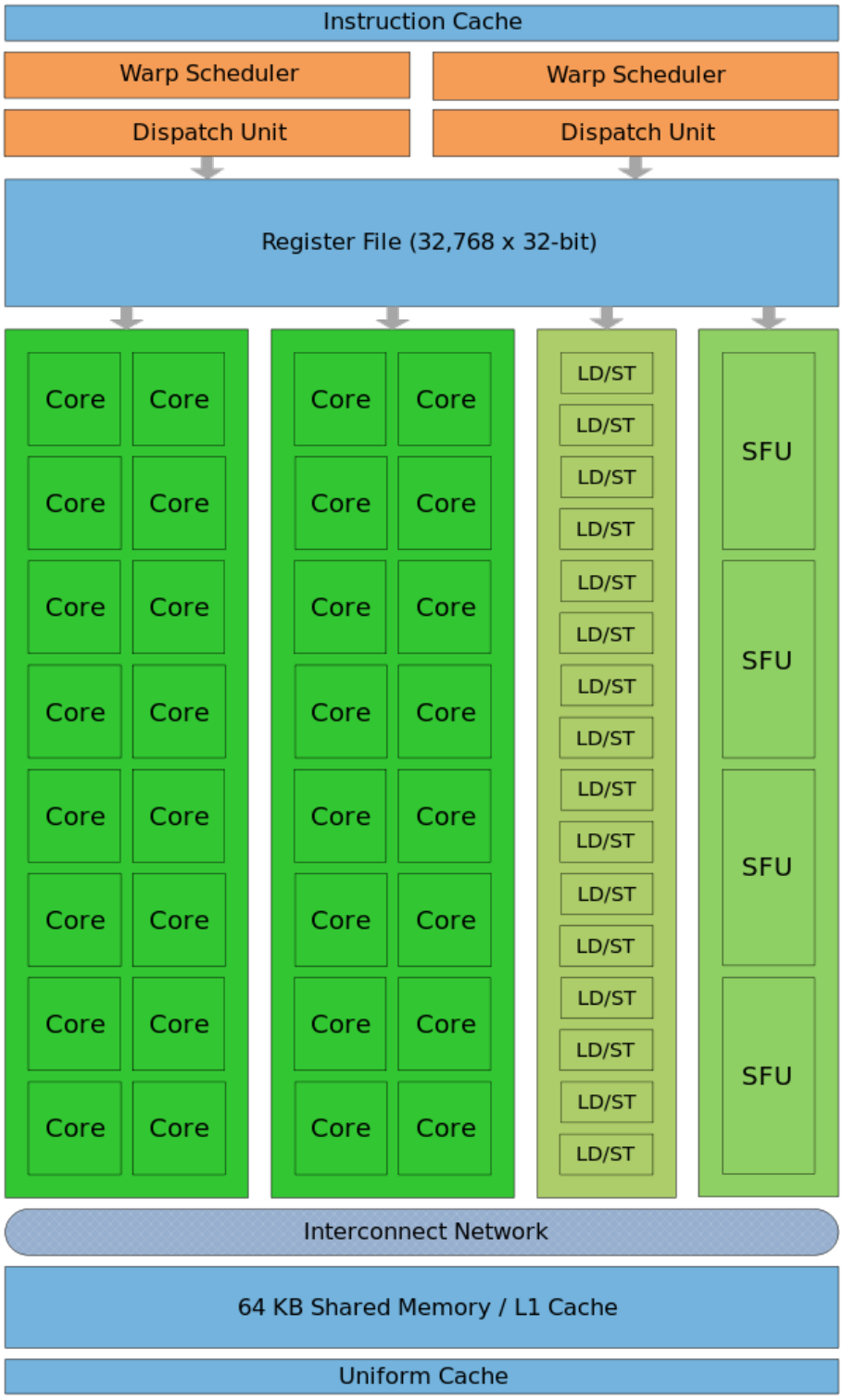


Figure B.2.: Block diagram of the NVIDIA Fermi streaming multiprocessor [32]

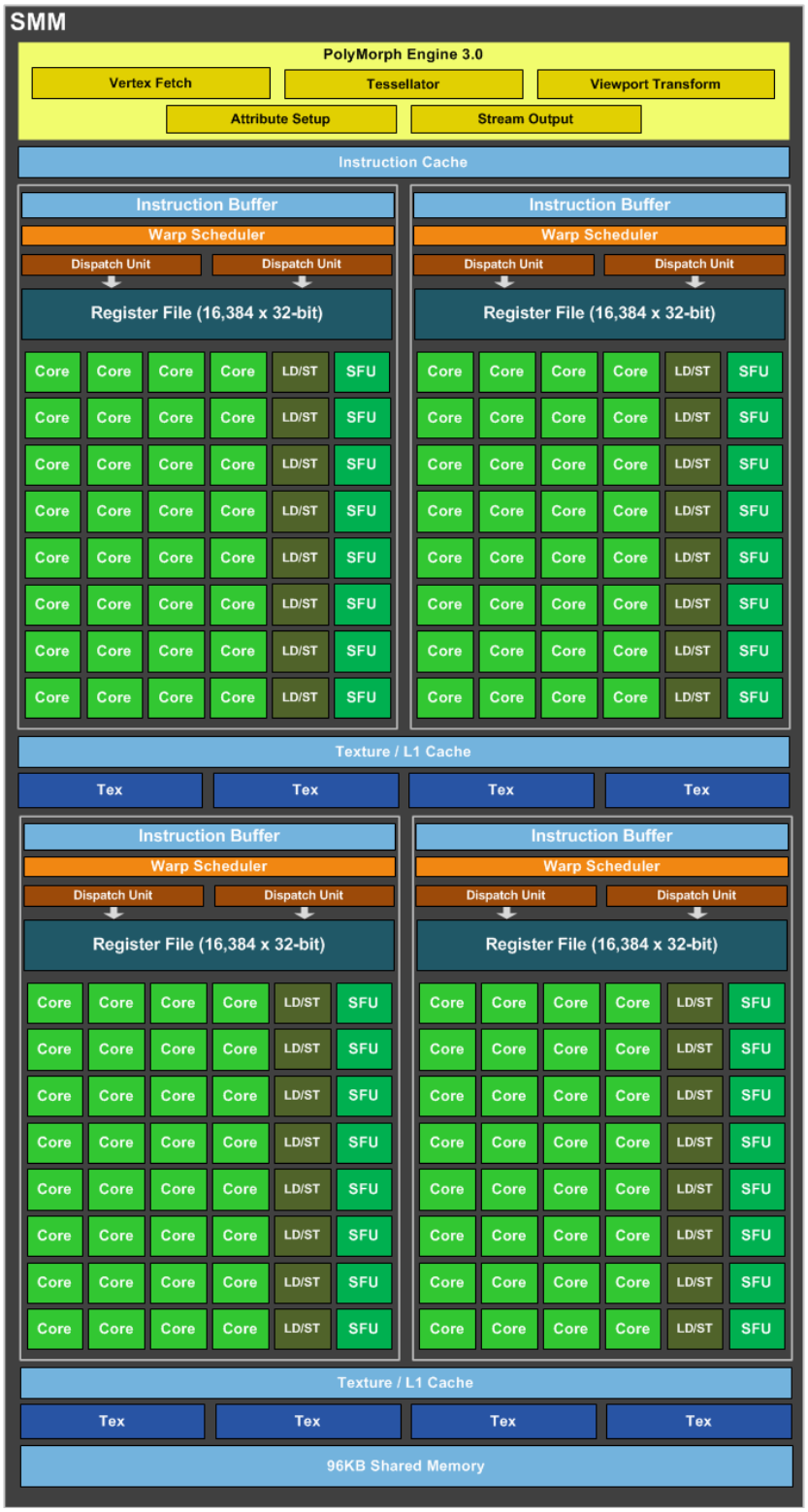


Figure B.3.: Block diagram of the NVIDIA Maxwell streaming multiprocessor [30]

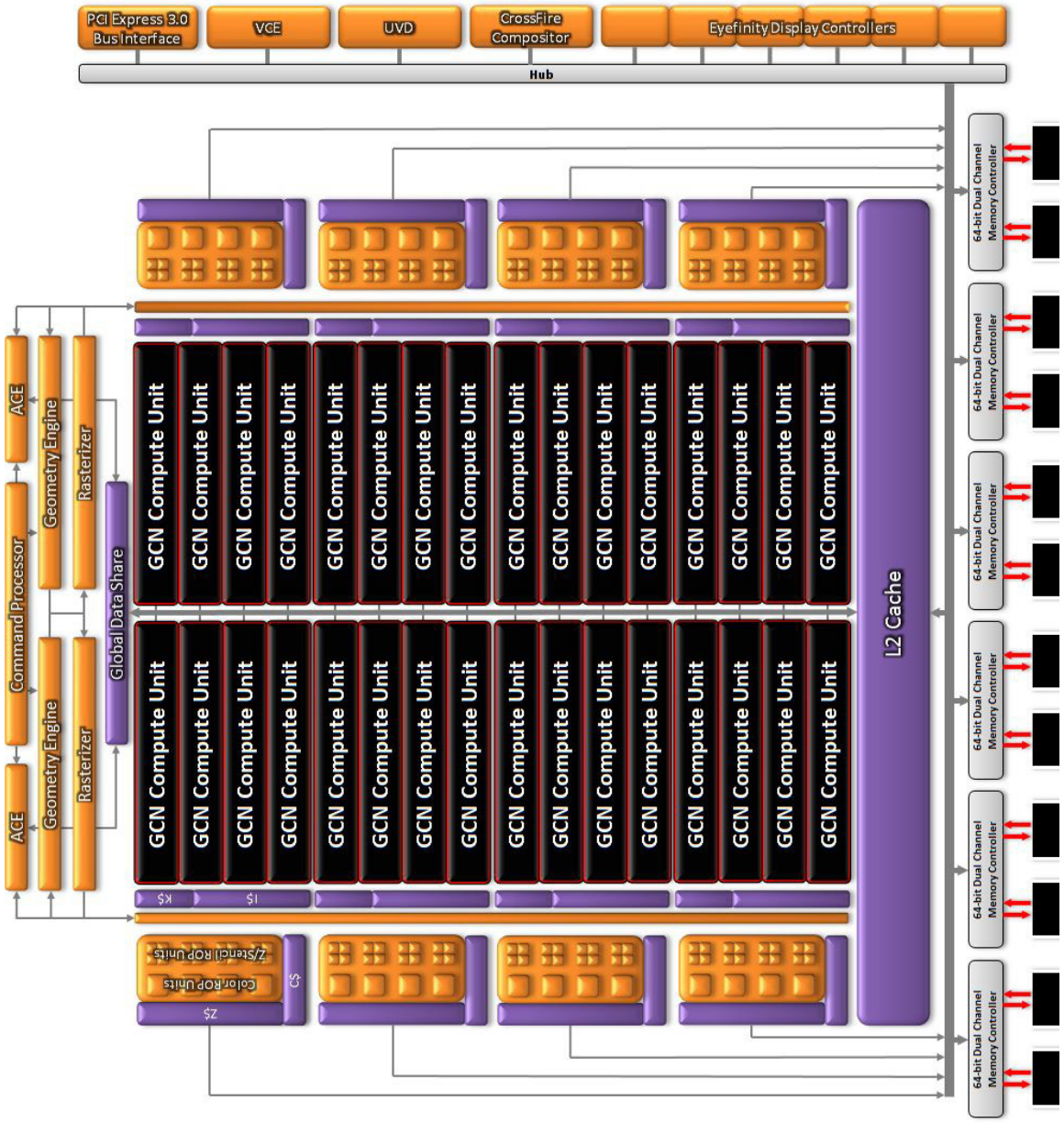


Figure B.4.: Block diagram of the AMD Radeon HD 7970 GPU [27]. *Note:* the symbol "$" is short for "cache".

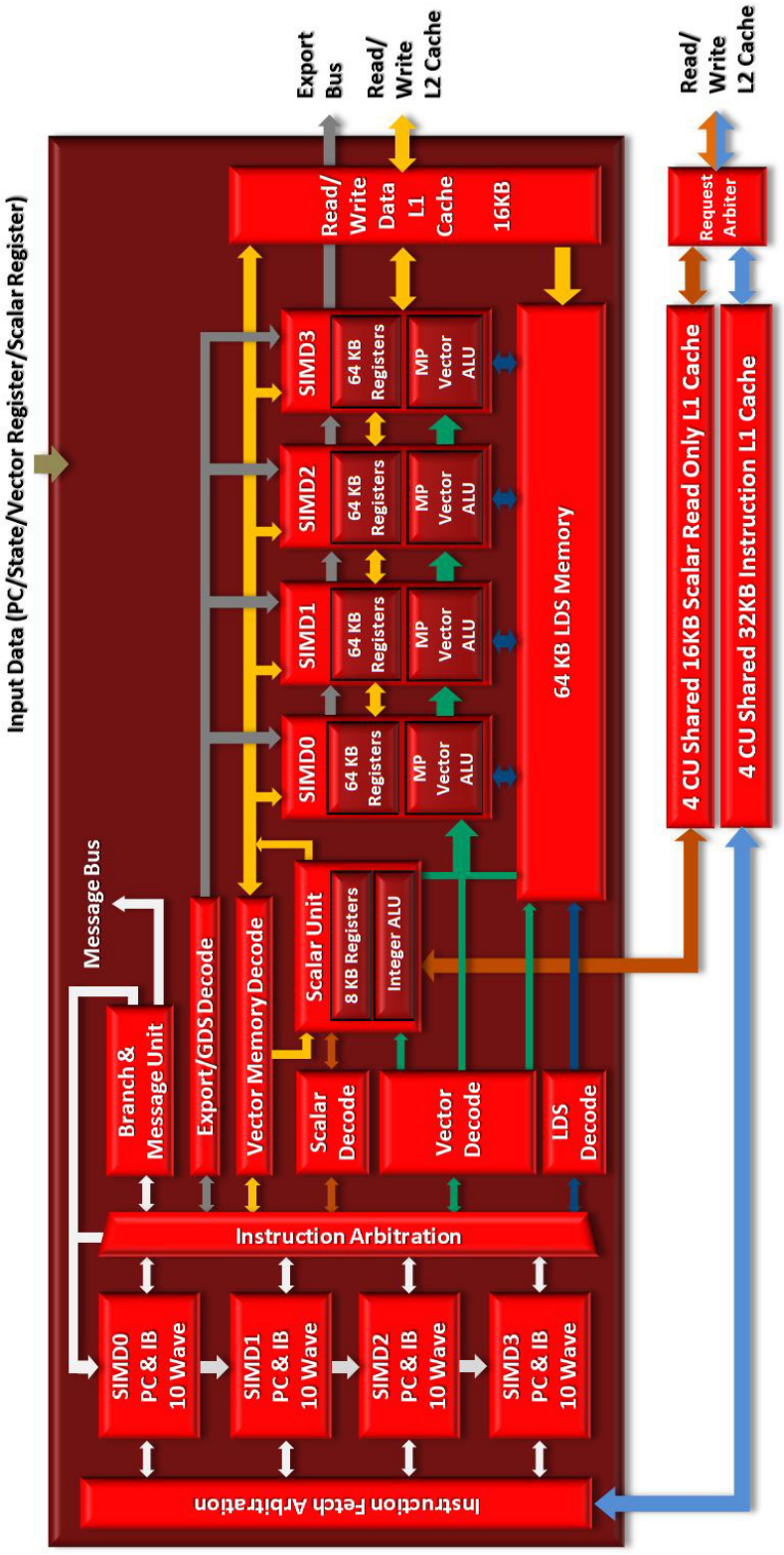


Figure B.5.: Block diagram of the AMD GCN compute unit [27]

C. Additional Tables

Table C.1.: List of buffer types supported as vertex attributes.

C++ type	related OpenGL type	related type size
bool	GL_BOOL	1
glm::bvec2	GL_BOOL	2
glm::bvec3	GL_BOOL	3
glm::bvec4	GL_BOOL	4
unsigned int	GL_UNSIGNED_INT	1
int	GL_INT	1
glm::ivec2	GL_INT	2
glm::ivec3	GL_INT	3
glm::ivec4	GL_INT	4
float	GL_FLOAT	1
glm::vec2	GL_FLOAT	2
glm::vec3	GL_FLOAT	3
glm::vec4	GL_FLOAT	4
double	GL_DOUBLE	1
glm::dvec2	GL_DOUBLE	2
glm::dvec3	GL_DOUBLE	3
glm::dvec4	GL_DOUBLE	4

Table C.2.: Standard camera controls.

Key	Action
w	move forward
s	move backwards
a	strafe left
d	strafe right
q	move down
e	move up
y	rotate clockwise
x	rotate counter clockwise
mouse	left click + drag to rotate camera

Table C.3.: Masses, initial positions and initial velocities of the solar system simulation [64] [65].

	Mass (10^{24}kg)	Distance from sun (10^9m)	Orbital velocity (10^3m/s)
Sun	1988500	0	0
Mercury	0.330	57.9	47.4
Venus	4.87	108.2	35.0
Earth	5.97	149.6	29.8
Mars	0.642	227.9	24.1
Jupiter	1898	778.6	13.1
Saturn	568	1433.5	9.7
Uranus	86.6	2872.5	6.8
Neptune	102	4495.1	5.4
Pluto	0.0131	5870	4.7

D. Performance Data

Table D.1.: Comparison of performance (time in seconds) of render and compute process (NVIDIA GeForce GTX780 Ti, Verlet integration, shared memory, local work group size of 512)[Figure 6.1].

particle counts	render process	update process
2	n/a	0.000357142
256	0.000527175	n/a
512	0.000438037	0.000362094
1024	0.000316217	0.000520204
2048	0.000349398	n/a
4096	0.00042633	0.00162046
8192	0.000528678	0.00318739
16384	0.000799377	0.00666956
32768	0.00112085	0.0222422
65536	0.00199991	0.0699921
131072	0.0036126	n/a
262144	0.00693321	n/a

Table D.2.: Peak performance (FPS) of the engine when rendering different number of particles [Figure 6.2].

particle count	GeForce GTX660	Quadro 4000	GeForce GTX780 Ti
256	3794.54	2422.67	1896.9
512	3316.68	2130.94	2282.91
1024	3003.61	1743.49	3162.39
2048	2252.86	1186.32	2862.07
4096	1468.03	828.896	2345.6
8192	2090.76	565.85	1891.51
16384	1292.55	328.712	1250.97
32768	765.295	250.265	892.181
65536	443.162	126.667	500.022
131072	235.813	89.3715	276.809
262144	122.193	45.7918	144.233

Table D.3.: Peak performance (GFLOPS) of the engine when updating the gravity sample application (Verlet integration, shared memory, local work group size of 512).[Figure 6.3]

particle count	GeForce GTX660	Quadro 4000	GeForce GTX780 Ti
2	0.0143404	0.000495125	0.0138472
512	43.1452	12.8037	31.0076
1024	88.5587	30.3707	63.1063
4096	351.6	88.3382	252.234
8192	501.355	168.836	491.257
16384	557.943	150.402	916.602
32768	623.952	189.502	1080.99
65536	680.404	181.987	1357.84

Table D.4.: Results (time in seconds) of the render benchmark on an NVIDIA GeForce GTX780 Ti [Figure 6.4].

particle count	CPU set up	GPU set up	CPU render	GPU render	CPU clean up	GPU clean up
256	2.2744e-05	1.05728e-05	1.22014e-05	1.28663e-05	5.70137e-05	0.00023717
512	2.10255e-05	3.17753e-05	1.55877e-05	2.1054e-05	2.93154e-05	0.000241764
1024	2.0837e-05	3.6823e-05	2.41971e-05	3.03479e-05	2.54631e-05	0.000145759
2048	2.00104e-05	3.67752e-05	4.07359e-05	5.21714e-05	2.54265e-05	0.000147703
4096	2.17083e-05	3.68011e-05	8.6355e-05	0.000116849	3.54764e-05	0.000158091
8192	2.00151e-05	3.67973e-05	0.000158988	0.000230882	2.62601e-05	0.000156108
16384	2.14847e-05	3.67552e-05	0.000318329	0.000444741	2.58302e-05	0.000210667
32768	2.3186e-05	3.67739e-05	0.000627838	0.000834032	2.65498e-05	0.000135018
65536	2.38045e-05	3.676e-05	0.00124383	0.00170941	2.67605e-05	0.000136807
131072	2.49971e-05	3.67865e-05	0.00248241	0.00331766	2.82425e-05	0.000137739
262144	2.58131e-05	3.68319e-05	0.00501427	0.0066392	2.80736e-05	0.000137677

Table D.5.: Results (time in seconds) of the update benchmark (Verlet integration, shared memory, local work group size of 512) on an NVIDIA GeForce GTX780 Ti [Figure 6.5].

particle count	CPU set up	GPU set up	CPU update	GPU update	CPU clean up	GPU clean up
2	7.34438e-06	7.5776e-07	0.000265925	9.38816e-06	3.60178e-06	7.5456e-07
512	7.43802e-06	8.2688e-07	3.81389e-06	0.000186339	4.23406e-06	7.4944e-07
1024	9.99478e-06	7.9904e-07	5.1249e-06	0.000365893	6.44917e-06	8.128e-07
4096	8.74969e-06	7.4144e-07	4.45345e-06	0.00146366	5.8811e-06	7.4048e-07
8192	7.81203e-06	7.6832e-07	4.10251e-06	0.00300569	5.13215e-06	7.4752e-07
16384	6.33029e-06	8.56e-07	3.26716e-06	0.00644328	3.75738e-06	7.9328e-07
32768	6.18585e-06	7.7152e-07	3.52643e-06	0.0218532	4.66486e-06	8.6176e-07
65536	6.31206e-06	7.7696e-07	3.41469e-06	0.0695887	5.6632e-06	8.576e-07

Table D.6.: Performance (GFLOPS) of different local work group sizes (Verlet integration, no shared memory, 65k particles)[Figure 6.6].

local group size	GTX660	Quadro4000	GTX780 Ti
2	n/a	n/a	35.1874
8	42.2068	18.9302	136.086
32	162.493	75.5195	496.516
128	472.352	134.028	1063.03
192	490.461	133.617	1085.07
256	482.143	135.067	1046.64
512	469.166	136.444	1071.68
1024	439.887	140.862	1041.74

Table D.7.: Performance (GFLOPS) of shared memory optimisation on an NVIDIA GeForce GTX780 Ti (Verlet integration, local work group size of 512)[Figure 6.7].

particle count	with shared memory	without shared memory
2	0.0138472	0.0151422
512	31.0076	28.6627
1024	63.1063	51.9419
4096	252.234	187.572
8192	491.257	372.847
16384	916.602	706.281
32768	1080.99	765.535
65536	1357.84	1071.68

E. Licence

This document and all of its images, tables and listings (except those taken from third-party sources) are published under the CC BY 4.0 licence[1].

The source code of the simulation engine is published under the zlib licence[2].

[1] `https://creativecommons.org/licenses/by/4.0/`
[2] `http://opensource.org/licenses/zlib-license.php`