Real-Time Rigid Body Interactions

Fredrik Fossum

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

This project focuses on developing parallel codes for rigid body interactions. The focus will be on parallelization in Open CL for GPUs using a fairly simple graphics interface. Two or more rigid bodies will be interacting in this real-time simulation.

Abstract

Rigid body simulations are useful in many areas, most notably video games and computer animation. However, the requirements for accuracy and performance vary greatly between applications.

In this project we combine methods and techniques from different sources to implement a rigid body simulation. The simulation uses a particle representation to approximate objects with the intent of reaching better performance at the cost of accuracy. We simulate cubes in order to showcase the behavior of our simulation, and also to highlight its flaws.

We also write a graphical interface for our simulation using OpenGL which allows us to move and zoom around our simulation, and choose whether to render cube geometry or the particle representations. We show how our simulation behaves in a realistic way, and when running our simulation on a CPU we are able to simulate several hundred cubes in real-time.

We use OpenCL to accelerate our simulation on a GPU, and take advantage of OpenCL/OpenGL interoperability to increase performance. Our OpenCL implementation achieves speedups up to 12 compared to the CPU version, and is able to simulate thousands of cubes in real-time.

Acknowledgments

I would like to thank my adviser Dr. Anne C. Elster for her advice. I would also like to thank NVIDIA for their hardware donations to our HPC-lab.

Trondheim, Jun 2011

Fredrik Fossum

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we briefly explain what a rigid body simulation is, and our motivations for creating one. We also explain what contributions our work brings to the field.

In simple terms, a rigid body simulation is a simulation of interacting objects that do not deform when they collide. In real life such objects do not exist, as all objects will deform when they collide with something. However, this deformation can be computationally expensive to calculate.

In some cases the deformation of an object is an important factor in the physical behavior of the object. In these cases rigid body simulation is not appropriate. Examples of an object like this is a piece of cloth, or jelly.

However, for many objects the deformation is very insignificant. When simulating such objects it can be a good idea to simplify the simulation by assuming that the objects cannot be deformed at all, in other words to do a rigid body simulation.

1.1 Motivation

One area where rigid body simulations is particularly useful is computer animation. Even with sophisticated software, animation is often a long and work intensive process. While manually animating realistic physics for a couple of objects is no problem, animating thousands of them *is.* For such a task, simulation is the only feasible option. Computer animation does not require the rigid bodies to be simulated in real time. However, good performance is still important. The simulation time will directly influence the productivity of the animator. A fast simulation will allow him or her to discover problems faster, and allows for more experimenting for better results.

A second important area for rigid body simulations is video games. A common trait for all rigid body simulation in video games is that they have to be real-time. This effectively limits the complexity of the simulation to something that looks and feels right, rather than being absolutely physically correct. However, between different games, the requirement for complexity in the rigid body simulation can vary.

Sometimes the rigid body simulation can be important to the gameplay itself. An example of this could be an adventure game where you have to move objects around and jump on them to reach your goal. In other cases the rigid body simulation is only for visual effect. An example of this could be the falling debris from a destroyed unit in a strategy game. Clearly, the first example requires a more accurate rigid body simulation than the latter.

In this thesis we want to especially focus on the last example; simulations which do not require very fine accuracy. Instead we aim for these simulations to be very fast, while at the same time look and feel correct.

1.2 Contributions

We provide a full open source implementation of a rigid body simulation, using a particle method. The source code can be found at `http://code.google.com/p/particle-rigid-body/`. We combine techniques from different sources to create a fast and stable simulation. We accelerate our simulation with OpenCL, and show how OpenCL is well suited to this task.

Chapter 2

Rigid Body simulation

In this chapter we describe what a rigid body simulation entails, and present some of the existing techniques and methods available.

A rigid body simulation can be roughly divided into two parts, collision detection and collision response [3]. The purpose of collision detection is to check whether the rigid bodies are in contact. If they are in contact, the simulation should also at this point extract any information about the collision necessary for the next step, collision response. This information can vary depending on the techniques used, but examples might be information such as intersection points and penetration depths.

Collision response is about creating appropriate forces and impulses between the colliding rigid bodies, to make the objects behave in accordance with the laws of physics.

The computation time spent for each of these two parts is roughly equal, making it important to be able to efficiently perform both.

2.1 Techniques and methods

Most collision detection algorithms use polyhedron representations for the rigid bodies. This means the bodies are made of vertices, edges and faces, just like the representation used by 3D graphics libraries such as OpenGL. Many algorithms also require that the bodies are convex. Fortunately, non-convex models can always be split into a group of convex ones. One technique to do this is using binary space partitioning [9]. However, finding a minimal partitioning is NP-hard, and many partitioning methods often produce a large number of convex smaller bodies, which is inefficient.

It is important for collision detection to limit the search somehow. Testing every single pair of objects for a possible collision is very computationally intensive. Considerable time can be saved by using a space partition. There are various different ways to do this, and some of the most notable are voxel grids, octrees, k-d trees and BSP trees [3].

Another technique to improve the performance of collision detection is to use a bounding volume of the body, such as a sphere or box. Detecting collisions with these shapes is a lot simpler and faster, however, they often do not provide a good approximation of the original shape.

The most accurate methods for collision response are the Finite Element Methods. They work by decomposing the body into a large number of smaller elements and then calculate the stress and strain on each of these smaller elements. However, these methods are computationally expensive and can not be used in real-time simulations. They are fine for testing engineering structures, but can not be used in interactive applications.

One of the most popular group of methods are Penalty Methods, largely because they are easily implemented. When two objects are found to be colliding, these methods use information about the collision to calculate an appropriate force between them. A common example is the spring model, in which a spring force is used to force contacting points apart.

A problem with these methods is that they often require tuning of several variables to create realistic results, such as spring coefficients in the case of the spring model. There is also little evidence to suggest that these methods accurately simulate what happens in real life. However, they can readily be applied where accuracy is not as important as simply getting realistic looking results.

Impulse-Based Methods model the forces between bodies through a series of impulses. An object resting on the ground will actually be vibrating up and down in a tiny movement, due to repeated impulses from the ground.

2.2 Bullet Physics Library

Bullet Physics Library [1, 5] is an open source physics engine featuring collision detection, soft body and rigid body dynamics. The library is widely used for video games, movies and 3D modelling software. Examples of products that have used Bullet include the video game *Grand Theft Auto IV* by Rockstar Games, the benchmarking utility *3D Mark 2011* by Futuremark, the Hollywood movie *2012* by Sony Pictures Imageworks, and the 3D modelling tools *Blender 3D* and *Cinema 4D*.

We have chosen to compare our results with the Bullet Physics Library because it is free, open source, cross platform, and consequently in widespread use.

Chapter 3

Parallel computing and GPU programming

3.1 Parallel Computing

Parallel computing is a form of computation in which multiple calculations are performed simultaneously. This is based on the principle that large problems can often be divided into smaller problems, which can be solved concurrently, opposed to sequentially like traditional non-parallel computing.

The main driving force between the recent development into parallel computing comes from the stagnation of the frequency of modern processors. The power density in modern processors are approaching the limit of what silicon can handle with current cooling techniques. This is known as the *power wall*.

Parallel computing can be embodied in many different forms, many of which do not exclude each other. The following are some of the layers of parallelism exposed by modern hardware [4]:

Multi-chip parallelism This means having several physical processor chips in a single computer. These chips share resources such as system memory through which the chips can relatively inexpensively communicate.

Multi-core parallelism This is similiar to multi-chip parallelism, with the distinction that the cores are all contained in a single chip. This lets the cores share resources like on-chip cache, allowing for less expensive communication.

Multi-context (thread) parallelism This is when a single core can switch between multiple execution contexts with little or no overhead.

Instruction parallelism This is when a single processor can execute more than one instruction simultaneously.

Traditionally, floating-point operations were considered expensive, while memory accesses were considered cheap. These roles have since been reversed, and memory has become the limiting factor in most applications. Data has to be transferred through a limited number of pins at a limited frequency, causing what is known as the *von Neumann bottleneck*.

3.1.1 Heterogeneous Computing

Recent developments are also showing rising interests in heterogeneous computing. Heterogeneous computing refers to systems that use a combination of different types of computational units.

The motivation for using heterogeneous systems is that although problems can be divided into smaller problems, all these smaller problems might not be of the same nature, and might benefit from different hardware architectures.

In the past, advances in technology and frequency scaling allowed most applications to increase in performance without structural changes. However, today, the effect of these advances are less dramatic since new obstacles such as the von Neumann bottleneck and the power wall have been introduced. Brodtkorb et al. [4] mentions the combination of a *Central Processing Unit* (CPU) and a *Graphics Processing Unit* (GPU) as one of the most interesting examples of heterogeneous systems for node level heterogeneous computing.

While neither the CPU or the GPU are heterogenous systems by themselves, they form a heterogeneous system when they are used together.

3.2 GPGPU

Using the GPU for computations traditionally handled by the CPU is known as General Purpose computing on Graphics Processing Units (GPGPU). The GPU is a specialized accelerator, designed for the acceleration of graphics computations. Their parallel nature allows them to efficiently perform large numbers of floating-point operations.

In terms of Floating Point Operations per Second (FLOPS), the GPU has has far surpassed the CPU, as can be seen in figure 3.1.

The reason for this huge discrepancy in floating-point capability is that the GPU is specialized for intensive, highly parallel computation, and is designed such that more transistors are devoted to data processing rather than flow control and data caching.

An illustration of the distribution of transistors in a CPU and a GPU can be seen in figure 3.2.

The GPU also has much higher memory bandwidth than the CPU, which can be seen in figure 3.3.

These two factors make the GPU a very interesting platform for computationally intense applications. A GPU can almost be seen as a small supercomputer, allowing a single computer to perform simulations that previously belonged to the domain of larger clusters of computers.

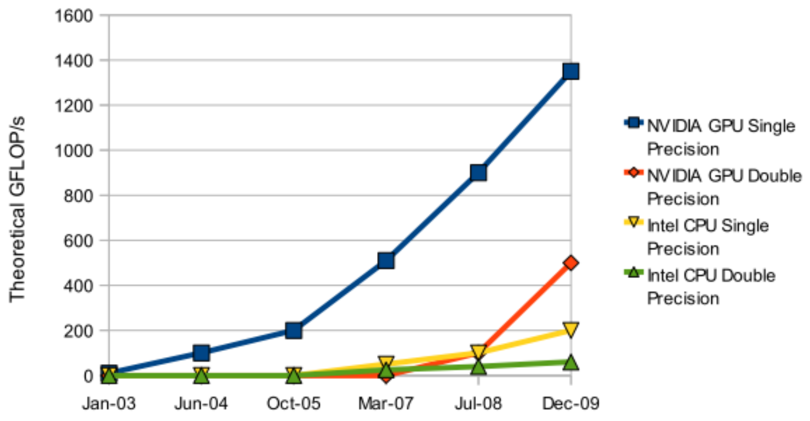

Figure 3.1: Development of Floating Point Operations per Second (FLOPS) in NVIDIA GPUs and Intel CPUs. [8]

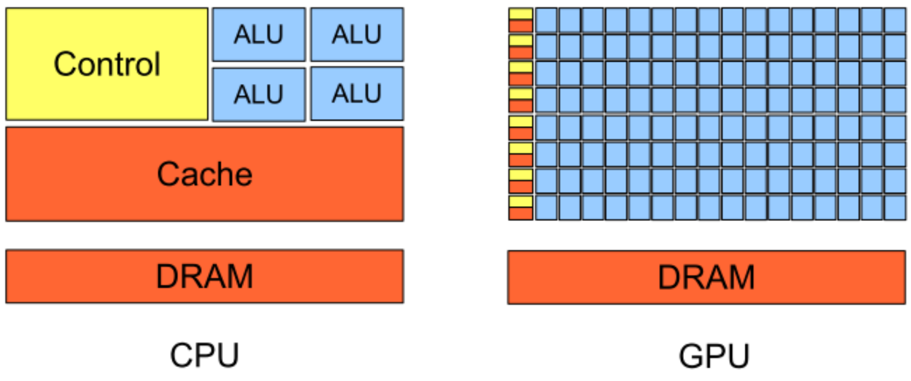

Figure 3.2: Distribution of transistors in a CPU and a GPU [8].

This has created ample new possibilities for the scientific community. Problems such as fluid simulation, molecular dynamics, medical imaging and many more are experiencing drastic performance improvements. This can often mean the difference between seeing the results of an operation almost immediately, as opposed to waiting for hours or even days.

However, a downside to using GPUs is that they typically use a PCI express bus, which can become a very serious bottleneck in cases where the entire problem does not fit in the GPUs memory. A second generation PCI express x16 bus allows a theoretical maximum of 8 GB/s of data transfer between CPU and GPU memory.

Initially, GPUs were not designed for GPGPU. GPUs were programmed using shaders, a set of software instructions performed on the GPU. These shaders are tightly knit to graphical concepts such as vertices and pixels. In order to perform operations that were not related to graphics, the problem had to be

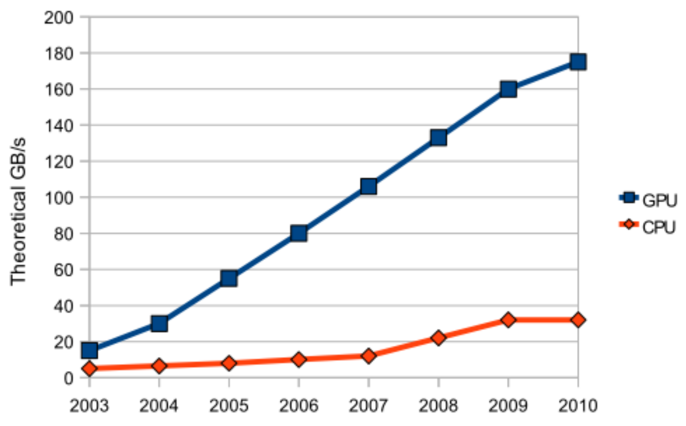

Figure 3.3: Development of Memory Bandwidth for NVIDIA GPUs and Intel CPUs. [8]

transformed so that the GPU could solve it as if it were graphics-related. This was often not only difficult, but also very limited, since a problem could not always be solved efficiently with the limitations of the shaders.

Today however, we have frameworks and languages tailored especially for GPGPU. Examples of these are NVIDIAs Compute Unified Device Architecture (CUDA), and OpenCL. While CUDA is limited to NVIDIA devices, OpenCL is a completely open standard, and has implementations across a wide range of devices and vendors, including NVIDIA and AMD. We go into more detail about OpenCL in chapter 4.

Chapter 4

OpenCL and OpenGL

In this chapter we briefly describe two of the core technologies we are using in our application. First we describe OpenCL, which we use to accelerate our simulation on the GPU. Then we describe OpenGL, which we use to render our simulation, and see how OpenCL and OpenGL can work together.

4.1 OpenCL

OpenCL (Open Computing Language) is a framework suited for parallel programming of heterogenous systems [12]. The framework includes the OpenCL C language, which is a language based on C99, for writing *kernels*, functions that execute on OpenCL devices such as a GPU.

An OpenCL device consists of multiple *Compute Units*, which in turn consists of multiple *Processing Elements.* These correspond to the streaming multiprocessors and scalar processors of a GPU respectively. When a kernel is to be executed it is put into a command queue, and then assigned to appropriate compute units and processing elements.

OpenCL supports two different programming models. They are a *Data Parallel*Âămodel, and a *Task parallel.* In the data parallel model the same kernel is executed simultaneously across the compute units or processing elements. In the task parallel model different kernels are executed. For our application we want to use the data parallel model. This is because we will be simulating many elements, and want to use the same kernel for every element. Filling the command queue with a kernel for each simulation element is very inefficient.

The data parallel kernel is divided into *Work Items.* A work item is the share of the kernel executed by a single processing element. When enqueueing a data parallel kernel the user has to specify the total number of work items the kernel should have, and also how many of these work items should be given to each compute unit. The group of work items given to a compute unit is called a *Workgroup.* The user has to take care to provide reasonable numbers, since the memory available in a compute unit is limited. The total number of work items also needs to be divisible by the number of work items given to each compute unit, in other words the size of the workgroup.

4.2 OpenGL

OpenGL, short for *Open Graphics Library*, is a software interface to graphics hardware [10]. It is a 3D graphics and modeling library, which is very fast, and also very portable. OpenGL implementations can be found on all major platforms and operating systems, including of course Windows, Mac OS and Linux.

OpenGL is intended to be used with hardware designed and optimized for displaying and manipulating 3D graphics. Software-only implementations do exist, however they generally do not perform as well, and may lack special effects.

OpenGL is procedural rather than descriptive. What this means is that instead of describing the scene and how it should look, the programmer instead writes step by step the operations required to create the desired appearance. These steps are calls to the many OpenCL functions which are used to draw graphics primitives such as points, lines and triangles in three dimensions.

OpenGL does not include any sort of window management, or user interaction. Each operating system has its own functions for this purpose and has the responsibility of giving OpenGL the control to draw images in a window. However there are cross platform libraries for this, such as GLUT (OpenGL utility toolkit) and SDL (Simple DirectMedia Layer).

4.2.1 Vertex Buffer Objects (VBOs)

A Vertex Buffer Object is an OpenGL extension that provides functions for uploading graphics data, such as vertex positions, normal vectors, color values etc., to a video device for non-immediate-mode rendering. Non-immediate-rendering basically means that the graphical elements are not rendered immediately as they are calculated. Instead the values are saved in a buffer on the device until all elements in the buffer are ready to be rendered. Then they are all rendered at once. This offers better performance, primarily because the data is located in video device memory rather than ordinary system memory, which allows the device to render it directly.

4.2.2 OpenCL/OpenGL interoperability

When using OpenCL there is an even greater incentive for using VBOs. Since calculations are performed on the GPU there should no longer be necessary to upload the data to the VBO buffers from system memory. Indeed, OpenCL and OpenGL offers interoperability which allows VBO buffers to be read and written to directly by the OpenCL kernels.

Chapter 5

Physical Model

In the following sections we present the physical model we have chosen for our implementation and go through the physics equations that govern the movement and behavior of the rigid bodies. Our physical model is mostly based on the work of Takahiro Harada in the book *GPU Gems 3* [7].

5.1 Particle approach

In our implementation we use particles to calculate the physical behavior of the rigid bodies, similar to the approach taken by Tanaka et. al [11], and Harada [7]. This approach offers both advantages and disadvantages compared to the alternative approach of using more complex geometry defined by vertices, edges and faces for calculations. Many of these advantages and disadvantages are discussed in the following sections.

The particles of a body have a fixed position relative to each other. It's worth noting that although we use the term *particle* throughout this report, a more correct term would have been *sphere* since the particles have a radius. However, in order to use the same terminology as previous works, and avoid confusion with a spherical rigid body, the term *particle* will be used.

We chose to let our rigid bodies be cubes, represented by 27 particles in a 3*3*3 formation. An illustration of this can be seen in figure 5.1.

The motivations for this are numerous. First of all cubes are quite simple to model, having only 6 faces. Secondly, humans are quite familiar with how a cubic rigid body moves in real life, namely a dice. This is makes it easy to spot whether the simulation is behaving as expected. Finally, and perhaps most importantly, cubes have the ability to stack on top of each other. Poor support for stacking rigid bodies is a weakness of using particles to represent the rigid bodies. By choosing a shape that normally stacks very easily we can explore this limitation and its implications.

However, the particle approach also has some benefits that warrants its use. Collision detection is greatly simplified. We only need to detect collisions be-

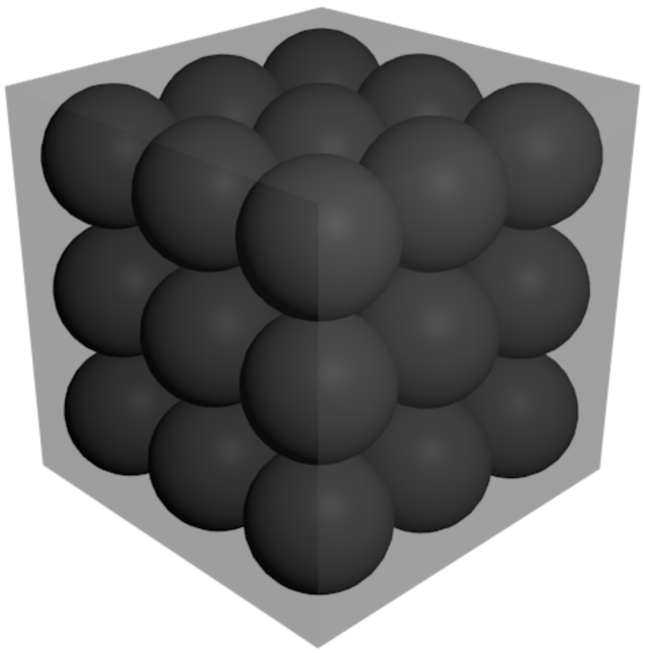

Figure 5.1: Cube with 27 internal particles

tween particles. Detecting a collision between to spheres is as simple as checking whether the distance between the center points is less than the sum of the radiuses of the spheres. The particle approach is also well suited for a GPU since it is easily parallelizable.

5.2 Physics

5.2.1 Movement

For the translation of a body, the following equations hold. When a force **F** acts on a rigid body, it gives the body *impulse*, which is change of the *linear* momentum of the rigid body, **P**. In other words, the time derivative of **P** is equal to **F**:

$$\frac{d\mathbf{P}}{dt} = \mathbf{F} \tag{5.1}$$

The definition of momentum is:

$$\mathbf{P} = M\mathbf{v} \tag{5.2}$$

where M is the mass of the rigid body, and $\mathbf{v}$ is its linear velocity. Through this definition we can obtain the velocity as

$$\mathbf{v} = \frac{\mathbf{P}}{M} \tag{5.3}$$

and, of course, velocity is the time derivative of the position $\mathbf{x}$:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \tag{5.4}$$

The following equations govern the rotation of a rigid body. When a force acts on a point of a rigid body that is different from the center of mass, it also gives the rigid body *torque*. Torque is the rate of change of *angular* momentum $\mathbf{L}$, like impulse is to linear momentum.

The amount of torque depends on the relative position $\mathbf{r}$ of the point where the force acts compared to the center of mass. More specifically, torque τ is defined as the cross product of $\mathbf{r}$ and the acting force $\mathbf{F}$.

$$\frac{d\mathbf{L}}{dt} = \tau = \mathbf{r} \times \mathbf{F} \tag{5.5}$$

The angular velocity $\mathbf{w}$ is obtained through the following equation:

$$\mathbf{w} = \mathbf{I}(t)^{-1}\mathbf{L} \tag{5.6}$$

where $\mathbf{I}(t)$ is the inertia tensor of the rigid body at time t, and is a $3 * 3$ matrix, and $\mathbf{I}(t)^{-1}$ is its inverse. Inertia is a measure of an objects resistance to change to its rotation, and the inertia tensor contains the objects inertia around all 3 axes.

Inertia, and thus the inertia tensor, depends on the shape, size and mass of an object. For a cuboid object, the inertia tensor looks like the following:

$$\mathbf{I} = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{bmatrix} \tag{5.7}$$

where h is the height, w is the width and d is the depth of the cuboid object. For a cube where height, width and depth are equal to the cube side s, the inertia tensor becomes:

$$\mathbf{I} = \begin{bmatrix} \frac{1}{6}ms^2 & 0 & 0 \\ 0 & \frac{1}{6}ms^2 & 0 \\ 0 & 0 & \frac{1}{6}ms^2 \end{bmatrix} \tag{5.8}$$

Inverting this matrix yields the inverse inertia tensor used in equation 5.6:

$$\mathbf{I}^{-1} = \begin{bmatrix} \frac{6}{ms^2} & 0 & 0 \\ 0 & \frac{6}{ms^2} & 0 \\ 0 & 0 & \frac{6}{ms^2} \end{bmatrix} \tag{5.9}$$

However, the inertia tensor only looks like this if the body is positioned such that its edges are parallel to the xyz-axes, like in figure 5.2.

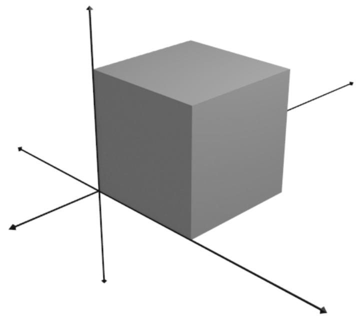

Figure 5.2: Cube with zero rotation

With time, as the rigid body rotates this will not be the case. The inertia tensor therefore has to be recalculated using the following equation:

$$\mathbf{I}(t)^{-1} = \mathbf{R}(t)\mathbf{I}(0)^{-1}\mathbf{R}(t)^T \tag{5.10}$$

where $\mathbf{R}(t)$ is the rotation matrix representing the rotation of the rigid body at time t. In our implementation, however, we don't use rotation matrices to store and calculate rotations. We instead use *quaternions*. A quaternion $\mathbf{q} = [s, \mathbf{v}]$ represents a rotation of s radians about an axis defined by the vector $\mathbf{v} = [v_x, v_y, v_z]$.

Quaternions are used for a number of reasons. They are more compact and faster to work with than rotation matrices since they only have 4 elements, compared to the 9 elements of the matrix. More importantly, however, there is no real difference between a rotation matrix and any other transformation matrix, which means a rotation matrix is capable of storing transformations other than rotation, such as translation, scaling and shearing. Numerical errors which add up during calculations can cause these transformations to build up in the rotation matrix, distorting the shape and size of the rigid body. Quaternions do not have this problem since they are not capable of representing anything other than rotation.

Since were not using a rotation matrix in the rest of our program, we have to convert our quaternion to a rotation matrix before calculating the inverse inertia tensor in equation 5.10. We calculate the rotation matrix $\mathbf{R}(t)$ from quaternion $\mathbf{q} = [s, v_x, v_y, v_z]$ as follows:

$$\mathbf{R}(t) = \begin{bmatrix} 1 - 2v_y^2 - 2v_z^2 & 2v_xv_y - 2sv_z & 2v_xv_z + 2sv_y \\ 2v_xv_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_yv_z - 2sv_x \\ 2v_xv_z - 2sv_y & 2v_yv_z + 2sv_x & 1 - 2v_x^2 - 2v_y^2 \end{bmatrix} \tag{5.11}$$

We can then calculate the inverse inertia tensor at time t using equation 5.10, and then calculate the angular velocity $\mathbf{w}$ using equation 5.6. We then want to use the angular velocity to update our rotation quaternion. The variation of quaternion $\mathbf{q}$ with angular velocity $\mathbf{w}$ is calculated using the following equation:

$$d\mathbf{q} = \left[\cos\left(\frac{\theta}{2}\right), \mathbf{a}\sin\left(\frac{\theta}{2}\right)\right] \tag{5.12}$$

where $\mathbf{a} = \frac{\mathbf{w}}{|\mathbf{w}|}$ is the rotation axis and $\theta = |\mathbf{w}dt|$ is the rotation angle. The quaternion at time $t + dt$ is calculated by the following equation:

$$\mathbf{q}(t + dt) = d\mathbf{q} \times \mathbf{q}(t) \tag{5.13}$$

where the multiplication of two quaternions $\mathbf{q}_0 = [s_0, \mathbf{v}_0]$ and $\mathbf{q}_1 = [s_1, \mathbf{v}_1]$ is defined as follows:

$$\mathbf{q}_0 \times \mathbf{q}_1 = [s_0 s_1 - \mathbf{v}_0 \cdot \mathbf{v}_1, s_0\mathbf{v}_1 + s_1\mathbf{v}_0 + \mathbf{v}_0 \times \mathbf{v}_1] \tag{5.14}$$

5.2.2 Collisions between particles

With the movement of the rigid bodies in place, the next thing to consider is the detection and reaction of collisions. As mentioned earlier, collision detection between particles is trivial. We only need to check whether the distance between two particles is less than the sum of their radiuses. In our implementation we subtract a small number from this check to avoid particles within the same cube to register a collision with each other. The particles within a cube are positioned perfectly close to each other, however numerical inaccuracies can cause this to be detected as a collision.

When a collision occurs the particles are pushed away from each other by a force modeled by a linear spring, as illustrated in figure 5.3.

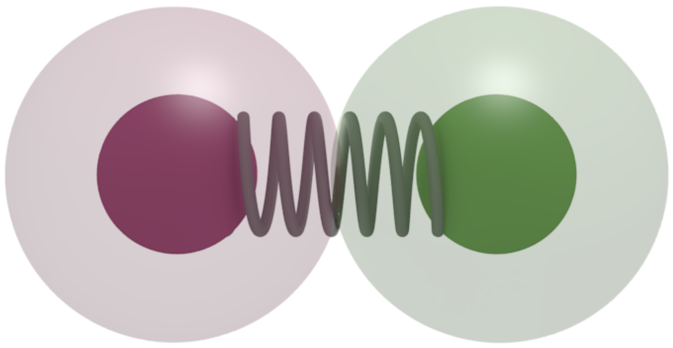

Figure 5.3: Collision between particles modeled by a spring

This force is calculated with the following equation:

$$\mathbf{f}_{i,spring} = -k\left(d - |\mathbf{r}_{ij}|\right)\frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \tag{5.15}$$

where k is the spring coefficient, d is the particle diameter, i is the index of the particle on which the force is acting, jÂăis the index of the colliding particle, and $\mathbf{r}_{ij}$ is the relative position of particle j with respect to particle i.

In addition to this spring force, we add a damping force which opposes the velocity of the particle. This force causes energy to dissipate and makes the rigid bodies come to rest, rather than bouncing around forever.

This force is modeled by a dashpot and is calculated using the following equation:

$$\mathbf{f}_{i,damping} = c\mathbf{v}_{ij} \tag{5.16}$$

where c is the damping coefficient, and $\mathbf{v}_{ij}$ is the relative velocity of particle jÂăwith respect to particle i.

To calculate this we need the velocities of the particles. The velocity of a particle is equal to the linear velocity of the rigid body it is a part of, plus the tangential velocity caused by the rotation of the rigid body.

We use the following equation to calculate the tangential velocity of a particle:

$$\mathbf{v}_{tangent} = \mathbf{w} \times \left(\mathbf{r} - \mathbf{w} \cdot \frac{\mathbf{w} \cdot \mathbf{r}}{|\mathbf{w}^2|}\right) \tag{5.17}$$

where $\mathbf{r}$ is the particle position relative to the center of the rigid body, and $\mathbf{w}$ is the angular velocity of the body.

5.2.3 Collision between a particle and a boundary

Similar forces are applied at the boundaries of the domain to keep the rigid bodies inside. However, since the domain does not move, the equations can be simplified somewhat. The boundary spring forces are calculated individually for each axis, since the force at a boundary only affects the force along the axis which is parallel to the normal of the boundary face, as seen in figure 5.4.

As an example, the equation for a particle colliding with the boundary whose normal is along the x-axis, and will be pushing the particle in the negative direction, looks like the following:

$$\mathbf{f}_{i,boundary-spring_x} = k(l_x - p_x - r) \tag{5.18}$$

where l_x is the boundary position on the x-axis, p_x is the particle position on the x-axis, r is the particle radius, and the expression inside the parentheses is the position of the colliding point of the particle with respect to the boundary. We don't have a minus in front of k in this equation because we multiply with

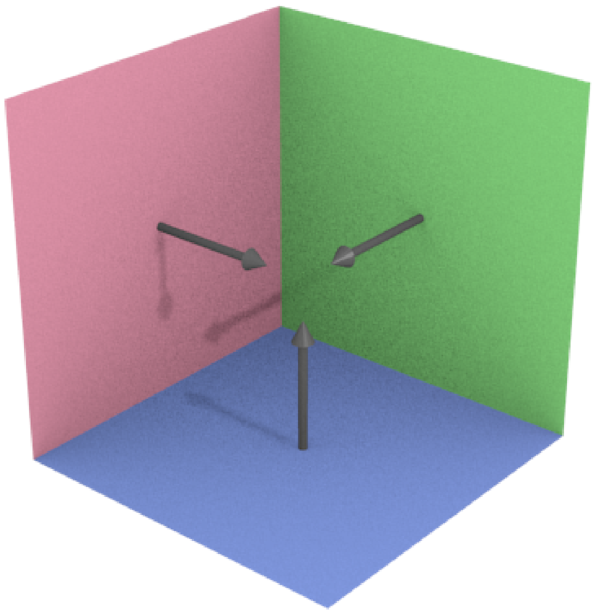

Figure 5.4: Three example boundaries and their normals

the position of particle i with respect to the colliding object, while in equation 5.15 we did it the other way around.

Similar forces work at each boundary wall, although for the opposite wall, where the particle will be pushed in the positive direction, the expression inside the parentheses changes slightly. The radius is negated, since we have to go the "other way" to find the point where the particle collides with the boundary.

$$\mathbf{f}_{i,boundary-spring_x-opposite} = k(l_x - p_x + r) \tag{5.19}$$

Lastly, as far as the boundaries are concerned, the damping force in equation 5.16 is applied. Again this is simplified somewhat by the fact that the boundaries don't move, and thus have a velocity of 0.

$$\begin{aligned} \mathbf{f}_{i,damping-boundary} &= c\,(0 - \mathbf{v}_i) \\ &= -c\mathbf{v}_i \end{aligned} \tag{5.20}$$

5.2.4 Summing the forces

When the forces on each particle have been calculated, the resulting total force and torque on the rigid body can be found. The force $\mathbf{F}$ on a rigid body is found by simply adding the forces on all its particles.

$$\mathbf{F} = \sum_{i \in RigidBody} \mathbf{f}_i \tag{5.21}$$

where $\mathbf{f}_i$ is the total force acting on particle i. The torque $\mathbf{T}$ on a rigid body is calculated by summing the cross product of the relative position of a particle to the center of the rigid body and the total force on the particle:

$$\mathbf{T} = \sum_{i \in RigidBody} (\mathbf{r}_i \times \mathbf{f}_i) \tag{5.22}$$

where $\mathbf{r}_i$ is the position of particle i relative to the center of the rigid body.

Chapter 6

Implementation

We have implemented a rigid body simulation in C++ and OpenCL. In the last chapter we presented the physical model, and the equations that govern the behavior of our implementation. In this chapter we describe how these equations come together to create a rigid body simulation. We begin by describing the sequential version written in C++. We then describe the OpenCL version, how it has been modified compared to the C++ version, and what considerations affected this process.

However, first of all we describe a grid structure we use in our application which greatly improves the time complexity of the simulation. This is vital for being able to simulate a large number of rigid bodies in real time.

6.1 Grid

Naively checking every other particle in the simulation when detecting collisions does not scale very well. In fact it carries a time complexity of *O*(n^2)*.* In order to improve this we implement a grid structure described in the paper *Particle Simulation using CUDA* by Simon Green [6].

The simulation world is divided into a uniform grid, and at the beginning of every iteration we check the position of every particle in order to determine which grid cell it belongs to. We find the cell index $\mathbf{i} = (i_x, i_y, i_z)$ with the following equation [7]:

$$\mathbf{i} = \frac{(\mathbf{p} - \mathbf{min})}{d} \tag{6.1}$$

where $\mathbf{p}$ is the position of the particle, and $\mathbf{min}$ is the position of the grid corner with the smallest coordinates in all dimensions. In our case the leftmost, backmost, lower corner. Finally, d is the cell size (cell side length).

By setting the grid cell size equal to twice that of the particle radius, we know that a cell will hold a maximum of 4 particles [6]. We also know that in order to find all particles colliding with a given particle, it suffices to search the cell which holds the particle itself, and the 26 cells that directly surround it. Figures 6.1, 6.3 and 6.2 show two-dimensional examples of the grid and illustrate how different cell sizes affect collision detection.

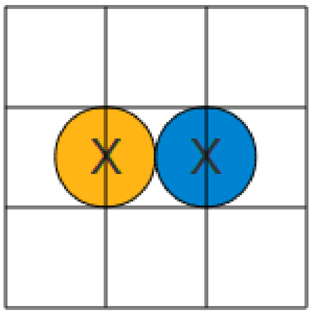

Figure 6.1: Grid with cell size equal to twice the particle radius

Figure 6.1 shows the grid cell size that we use, with cell size equal to twice the particle radius. If we imagine that neither particles reside in the center cell, we can see that the distance where the particles no longer reside in neighboring cells coincides perfectly with the distance where they no longer collide. Using a larger cell size would clearly result in checking unnecessary particles.

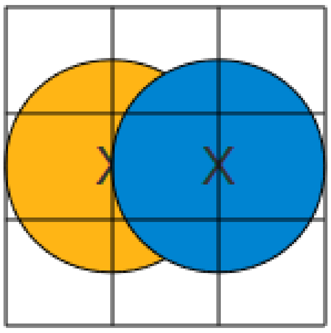

Figure 6.2: Grid with cell size equal to particle radius, with particles close

Figures 6.2 and 6.3 show what would happen if we used a smaller cell size, in this case equal to the particle radius. Imagine again that neither of the particles occupy the center cell. By only checking directly surrounding cells during collision detection these two particles would not be checked against each other, even though they clearly intersect. In figure 6.3 we see that there are now two cell lengths between two touching particles. We would therefore have to check an additional "layer" of cells when checking for colliding particles, for a total of $5 \cdot 5 \cdot 5 = 125$ cells.

The grid structure brings the simulation down to a complexity of $O(n)$, since each of the n particles no longer has to check all $n - 1$ other particles, but

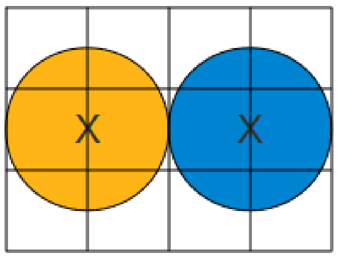

Figure 6.3: Grid with cell size equal to particle radius, with particles far apart

only a constant number. Although the grid generation at the beginning of each iteration adds a considerable overhead, the performance gained by using the grid during collision detection is much, much greater. The implementation with the grid outperforms the naive approach even at quite small problem sizes of only a couple hundred particles.

6.2 Sequential version

We have created a simple class hierarchy consisting of `Body`, `Cube` and `Particle` classes. The `Body` class is an abstract class and cannot be instantiated. It contains data structures to contain all properties that are present with all rigid bodies, such as mass, position, velocity, momenta, rotation and a list of `Particle` objects, representing the particles that belong to the rigid body. The `Body` class will make it easier to extend the application with differently shaped rigid bodies in the future.

The `Cube` class extends the `Body` class and contains functionality that is specific to cubic rigid bodies. Most of this is related to how to properly render a cube, but it also needs to set the inertia tensor to its correct initial non-rotated state, which as mentioned in section 5.2.1 will differ for differently shaped objects. The `Cube` class also needs to instantiate a `Particle` object for every particle that makes up the cube. Although the list of particles belong to the `Body` superclass, it is the responsibility of the subclass to fill this list with the correct number of particles and position these particles correctly.

Finally the `Particle` class, as mentioned, represents the particles of the rigid bodies. It contains properties such as mass, position and velocity, as well as functions for calculating the grid cell of a particle. This is also where we find all the calculations of collision detection and reaction.

A UML diagram for the three classes can be seen in figures 6.4 and 6.5. In these diagrams public fields and functions are denoted by a `+` sign, protected are denoted by `#` while private by a `-`. Static fields are underlined, while virtual functions that have to be overridden by a subclass are written in italics.

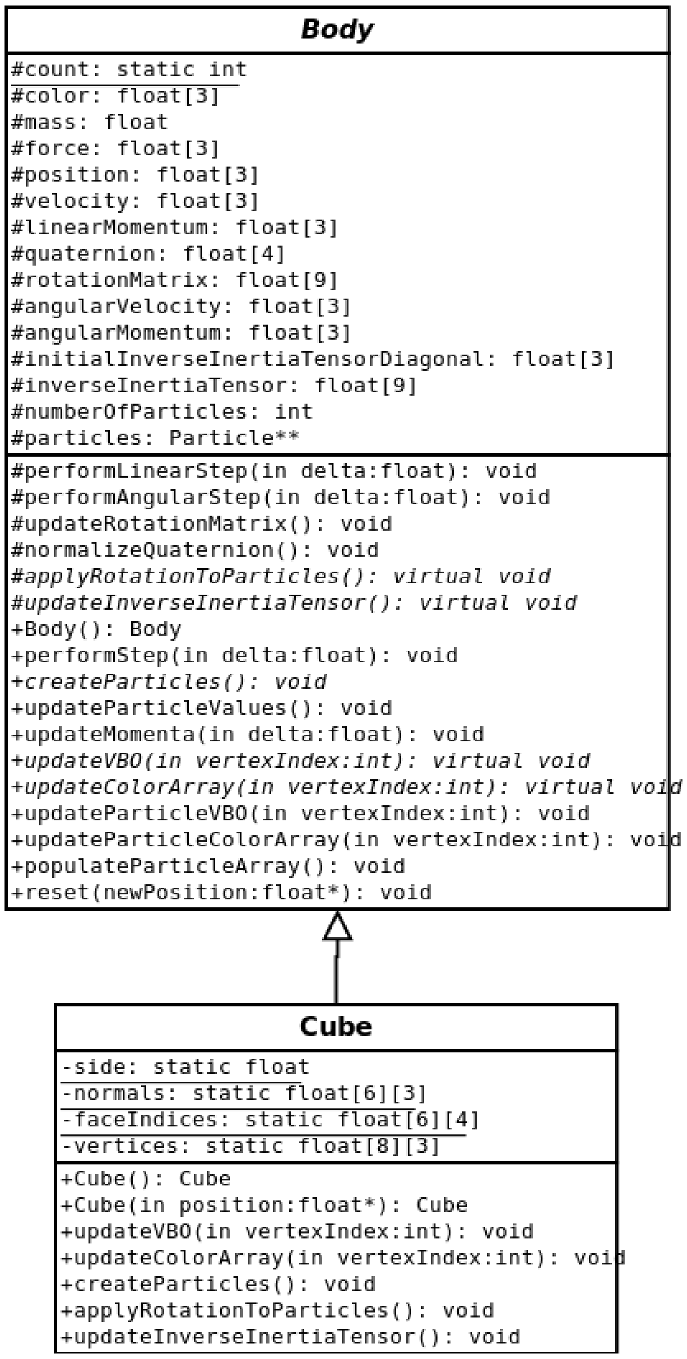
Body
#count: static int
#color: float[3]
#mass: float
#force: float[3]
#position: float[3]
#velocity: float[3]
#linearMomentum: float[3]
#quaternion: float[4]
#rotationMatrix: float[9]
#angularVelocity: float[3]
#angularMomentum: float[3]
#initialInverseInertiaTensorDiagonal: float[3]
#inverseInertiaTensor: float[9]
#numberOfParticles: int
#particles: Particle**
#performLinearStep(in delta:float): void
#performAngularStep(in delta:float): void
#updateRotationMatrix(): void
#normalizeQuaternion(): void
#applyRotationToParticles(): virtual void
#updateInverseInertiaTensor(): virtual void
+Body(): Body
+performStep(in delta:float): void
+createParticles(): void
+updateParticleValues(): void
+updateMomenta(in delta:float): void
+updateVBO(in vertexIndex:int): virtual void
+updateColorArray(in vertexIndex:int): virtual void
+updateParticleVBO(in vertexIndex:int): void
+updateParticleColorArray(in vertexIndex:int): void
+populateParticleArray(): void
+reset(newPosition:float*): void
Cube
-side: static float
-normals: static float[6][3]
-faceIndices: static float[6][4]
-vertices: static float[8][3]
+Cube(): Cube
+Cube(in position:float*): Cube
+updateVBO(in vertexIndex:int): void
+updateColorArray(in vertexIndex:int): void
+createParticles(): void
+applyRotationToParticles(): void
+updateInverseInertiaTensor(): void

Figure 6.4: Body and cube UML class diagram

6.2.1 Initialization

The user enters the number of rigid body cubes to simulate as an argument when running the application. E.g. executing the command

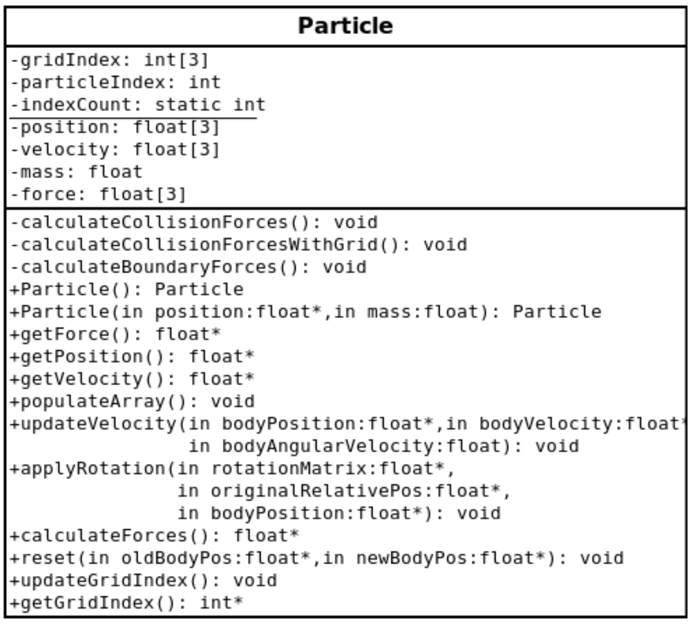
Particle
-gridIndex: int[3]
-particleIndex: int
-indexCount: static int
-position: float[3]
-velocity: float[3]
-mass: float
-force: float[3]
-calculateCollisionForces(): void
-calculateCollisionForcesWithGrid(): void
-calculateBoundaryForces(): void
+Particle(): Particle
+Particle(in position:float*,in mass:float): Particle
+getForce(): float*
+getPosition(): float*
+getVelocity(): float*
+populateArray(): void
+updateVelocity(in bodyPosition:float*,in bodyVelocity:float
in bodyAngularVelocity:float): void
+applyRotation(in rotationMatrix:float*,
in originalRelativePos:float*,
in bodyPosition:float*): void
+calculateForces(): float*
+reset(in oldBodyPos:float*,in newBodyPos:float*): void
+updateGridIndex(): void
+getGridIndex(): int*

Figure 6.5: Particle UML class diagram

```
./rigidbody 1000
```

will launch a simulation of 1,000 cubes, which corresponds to 27,000 particles. The first thing that happens is that `Cube` objects representing these cubes are instantiated, as well as `Particle` objects for their particles. Static counters are used to keep track of how many cubes and particles have been created at all times. This information is used to fill global arrays that hold references to every cube, and every particle. This is needed to iterate over all cubes and particles later. As they are created, we place the cubes above each other while moving them back and forth between 4 different positions along the ground plane. A screenshot of the resulting initial position for the cubes can be seen in figure 6.6.

The reason for this is simply to create interesting behavior. The cubes are moved only so much that their sides and corners still slightly overlap, causing inevitable rotation when they collide with each other at the ground. We use a random number generator to generate colors for the rigid bodies. Since this has no significant importance for the simulation other than visuals the standard C++ `rand` function is good enough for this. However, we seed the random number generator with a timestamp to avoid generating the same color combinations every time the application is executed.

6.2.2 Iteration walkthrough

In this section we walk through all the calculations that are performed in an iteration. A flowchart showing a quick overview of an iteration can be seen in

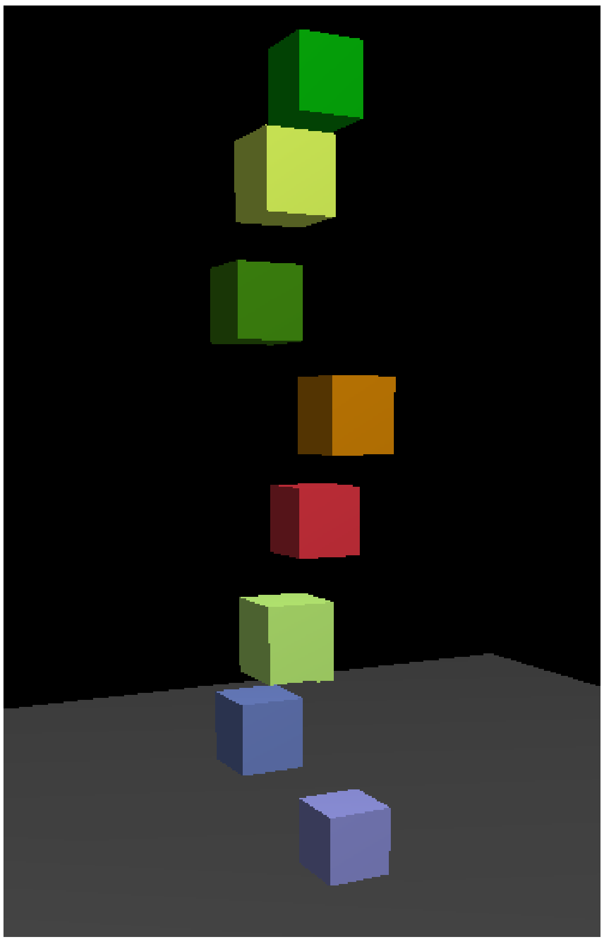

Figure 6.6: Initial simulation state

figure 6.7.

1. Update grid
   Every iteration begins with updating the grid structure described in section 6.1. This three-dimensional grid is represented by two one-dimensional arrays, which we call `indexGrid` and `countGrid`.

   Let's say there are m^3 cells in the grid. Then the `countGrid` is an integer array of size m^3, with one integer for each cell, representing the number of particles currently registered to each cell. `indexGrid` is an integer array of size $(m^3 \cdot 4)$, holding 4 integers for each cell, representing the indices of the particles currently registered to each cell. When we update the grid, `countGrid` first needs to have all its members reset to 0. We then iterate over all the particles in the simulation and calculate which cell they belong to. When we calculate the cell of a particle, we use the `countGrid` value for that cell to find out where to insert the particle index in the `indexGrid`.

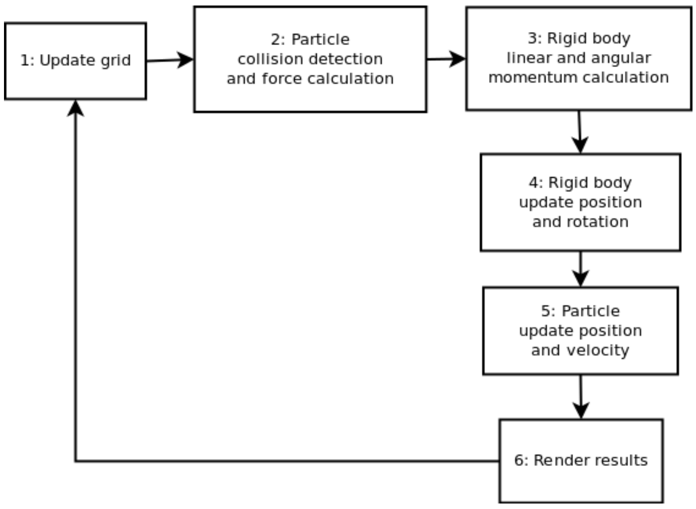
1: Update grid
2: Particle collision detection and force calculation
3: Rigid body linear and angular momentum calculation
4: Rigid body update position and rotation
5: Particle update position and velocity
6: Render results

Figure 6.7: Flowchart showing a simulation iteration

For example, we calculate that a particle i belongs to grid cell k and we check the value of `countGrid[k]` to find that there are 0 particles registered to cell k so far. We then put the index i into `indexGrid[4k + 0]`, and increment `countGrid[k]` by one. If there had been 2 particles registered to cell k already, we would insert i into `indexGrid[4k + 2]` instead. We multiply k by 4 when indexing `indexGrid` because the array holds 4 integers for each grid cell.

2. Particle collision detection and force calculation
   Once the grid has been updated, we can perform collision detection. For every particle we check its grid position and then iterate over its grid cell and the 26 grid cells directly surrounding it, extracting the indices of the contained particles.

   We then iterate over all the particles whose indies we extracted checking for collisions. In case of a collision we calculate the resulting force immediately using equations 5.15 and 5.16. We then add this force to the total force on the particle. Of course, we have to take care to reset the force on the particle to 0 before starting adding these new forces, so that the calculated force from a previous iteration does not propagate.

3. Rigid body linear and angular momenta calculation
   The previous step calculated all the forces for all the particles, however, to calculate the linear and angular momenta on the rigid bodies we need the force and torque on the rigid bodies themselves. For each rigid body we iterate over all 27 particles of the rigid body and add the contribution to force and torque from each particle to a total. Equations 5.21 and 5.22 are applied here.

   Once the total force and torque for each rigid body has been found, we have found the rate of change for linear and angular momenta respectively, according to equations 5.1 and 5.5. Combined with the value for the *time* *delta*, Δt, (which is defined by the user) we can then update the linear and angular momenta by $\mathbf{P} \leftarrow \mathbf{P} + \mathbf{F} \cdot \Delta t$ and $\mathbf{L} \leftarrow \mathbf{L} + \tau \cdot \Delta t$.

However, before we move on we check whether the absolute value of the linear momentum in each direction is greater than a maximum value calculated from a user defined terminal momentum. If it is, we set it to the maximum value, or negative the maximum value, depending on whether the momentum is positive or negative. This is called a *clamp* operation. In this case we clamp the momentum between the range $[-maximum\ momentum, maximum\ momentum]$. We do this to prevent the rigid bodies from accelerating to unrealistically great speeds when they fall for a long time. This is similar to how in real life air resistance would prevent objects from accelerating past a terminal velocity, however our approach is a little simplified. Air resistance would build up as the velocity of the object increased, slowing acceleration gradually. Our approach allows the object to accelerate unhindered up until it hits terminal velocity, upon which time acceleration halts completely.

4. Rigid body position and rotation update
   Once we have the linear and angular momenta we can find the linear and angular velocities of the rigid bodies. We iterate over all the rigid bodies in the simulation and for each we calculate their linear and angular velocity as follows. Finding the linear velocity is easily done simply by dividing the linear momentum by the mass of the rigid body, in accordance with equation 5.3. Finding the angular velocity is a more extensive operation.

   To calculate the angular velocity we need to update the inverse inertia tensor for the objects current rotation. To calculate the inverse inertia tensor we need the rotation matrix for the objects current rotation. But as mentioned in section 5.2.1 we use quaternions to represent rotation. Therefore we first have to convert our quaternion to a rotation matrix using equation 5.11. However, equation 5.11 requires that the quaternion $\mathbf{q}$ is normalized, meaning that $|\mathbf{q}| = 1$. We check whether $|\mathbf{q}|$ is more than a small threshold unequal to 1. If it is, we normalize it as if it were any other vector: $\mathbf{q} \leftarrow \frac{\mathbf{q}}{|\mathbf{q}|}$. We can then calculate the new inverse inertia tensor using equation 5.10, and consequently we can find the angular velocity with equation 5.6.

   We have the velocities of the rigid body and can update the position. There are many possible ways to do this, using different numerical integration methods. In this implementation we simply use Euler method, and update position $\mathbf{x}$ by $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v} \cdot \Delta t$. We update rotation using equation 5.13. Using a more advanced integration method could be a part of future work, explored in more detail in section 8.2.

5. Particle position and velocity update
   We now have everything we need to update the positions and velocities of the particles. This operation could technically be performed at the beginning of the iteration instead of at the end. The particles would then receive the changes from the previous iteration at the beginning of the next, which would be fine if we only rendered the cube geometry. However, we have added the ability to render the particles themselves instead of the

cubes for the purpose of visually demonstrating how the simulation works. If we did not update the particles before rendering there would be a single frame of delay of the visualized particles compared to their computation. This is minuscule of course, especially at small time steps, but there is no downside to updating the particles at the end of the iteration instead, so we might as well do so.

To calculate the position of a particle we need the current position and rotation of the rigid body it belongs to, as well as the *initial position of the particle relative to the center of the rigid body at zero rotation.* It is the responsibility of the rigid body subclass to convey these initial positions to the particles when updating their current positions. We therefore iterate over all rigid bodies, and for each rigid body iterate over their particles when performing this operation. The new particle position is found by applying the rotation to the initial relative position and adding the rigid body position. We apply the rotation by multiplying the initial relative position vector by the rotation matrix, yielding a new position vector. We then add the rigid body position vector to this.

We update the particle velocity by applying equation 5.17, which is straight forward to calculate tangential velocity, and then adding the linear velocity of the rigid body itself. The only thing to consider is that we must check whether the angular velocity is zero before dividing by the square of its size. This would of course would also be zero, causing an error due to division by zero. If the angular division turns out to be zero, the tangential velocity is simply zero also. However the linear velocity of the rigid body is still added.

6. Render results
   When all the calculations of an iteration are completed, we need to render the results to the screen. For rendering we use OpenGL and vertex buffer objects (VBOs) explained in section 4.2. We use two separate VBOs in our implementation. One for rendering cubes, and one for rendering particles. We go into more detail about rendering in section 6.2.3. For now, suffice to say that we update only the VBO for the currently chosen object representation. If complete cubes are chosen the vertex positions and normals have to be recalculated using their original position and the current rotation. If particles are chosen we already have the vertex positions, since these are simply the particle positions, and we only need to insert these values into the VBO. Once the VBO values have been updated we need to copy them to the device, before we draw the chosen VBO to the screen.

6.2.3 Rendering

We use three two-dimensional arrays when drawing cubes. We call them `normals`, `faceIndices` and `vertices`. We present these arrays as matrices in figure 6.8. To help make sense of these arrays we have figure 6.9 which shows a wireframe cube with numbered vertices.

The `vertices` array contains the positions of the vertices themselves. There are

8 vertices in a cube, and each vertex has 3 coordinate values. The vertices are ordered by ascending index in figure 6.8, with the top row representing vertex 0, the next representing vertex 1 and so on.

A cube has 6 faces, and each face is defined by 4 vertices. The array `faceIndices` holds the indices of the vertices that define each face. As you can see the first face is defined by vertices 0, 1, 2 and 3.

Finally the `normals` array holds the unit vectors that define the direction of the normal for each face. The first face has a normal which goes along the negative x-direction, represented by the values -1.0 for x, and 0.0 for y and z.

We use OpenGL *Quads* rather than *Triangles* when we draw our cubes, meaning we define four vertices for every face, rather than three. As mentioned earlier we use a VBO for rendering. For rendering cubes we put vertex data, normals and colors after each other in a single long array. With 6 faces in each cube, 4 vertices per face, and 3 values per vertex, each cube requires 72 values for their vertex data. Since the normals and color data also work with individual vertices rather than faces, they also need 72 values each. This results in a total of 216 values for each cube in the cube rendering VBO. With n cubes, we have a total of $216n$ values. The first $72n$ values are vertex data, the next $72n$ values are normal data and the final $72n$ are color data, and we set the OpenGL vertex pointer, normal pointer and color pointer accordingly.

When we draw our particles we use OpenGL *Points*. OpenGL then draws a point for every single vertex we define. Points do not have normals, so unlike when drawing cubes, we do not need space for normal data in our particle VBO. With 27 particles in every cube, and 3 values for each particle, each cube requires 81 values for their vertex data. Again, as with the cube VBO, color data works with individual vertices and also requires 81 values. Total values for each cube is then 162. Similar to before, with n cubes, there are a total of $162n$ values in the particle VBO. The first $81n$ of which are vertex data, and the last $81n$ are color data. We set the OpenGL vertex pointer and color pointer accordingly. We do not use the normal data, nor the normal pointer, in the case of rendering particles.

6.3 OpenCL version

In order to run our implementation on the GPU with OpenCL, a number of modifications had to be made. Perhaps the most significant difference is that OpenCL does not support classes in the kernel code, which is the code executed by the device. To circumvent this limitation we create numerous new arrays containing the attributes in the classes. We call these arrays *OpenCL arrays*. We create one OpenCL array for each attribute.

For example we have an array `clBodyMass` containing the mass values for all the rigid bodies. To get the mass belonging to rigid body i we look up `clBodyMass[i]`.

For attributes that consist of multiple values, such as position and velocity, which are vectors with 3 elements, we use the OpenCL vector types `cl_float`n,

$$\texttt{vertices[8][3]} = \begin{bmatrix} -0.5s & -0.5s & 0.5s \\ -0.5s & -0.5s & -0.5s \\ -0.5s & 0.5s & -0.5s \\ -0.5s & 0.5s & 0.5s \\ 0.5s & -0.5s & 0.5s \\ 0.5s & -0.5s & -0.5s \\ 0.5s & 0.5s & -0.5s \\ 0.5s & 0.5s & 0.5s \end{bmatrix}$$

$$\texttt{faceIndices[6][4]} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 3 & 2 & 6 & 7 \\ 7 & 6 & 5 & 4 \\ 4 & 5 & 1 & 0 \\ 5 & 6 & 2 & 1 \\ 7 & 4 & 0 & 3 \end{bmatrix}$$

$$\texttt{normals[6][3]} = \begin{bmatrix} -1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & -1.0 \end{bmatrix}$$

Figure 6.8: Two-dimensional arrays used to draw cubes, where s is the side length of the cube

where n is a power of 2, or the number 3. However, in the case of the inverse inertia tensor and rotation matrix, which are matrices with 9 elements, we would need to use `cl_float16` to fit all the values. Since we only need 9 values, this would be a considerable waste of space. We therefore use multiple `cl_float`, and make the OpenCL array 9 times as long instead, with 9 elements for each rigid body.

During initialization we add a phase where we iterate over all the bodies and particles and insert their attribute values into their respective OpenCL arrays. These arrays are located in host memory. The device, the GPU, which will be doing all the computations, cannot read from host memory directly. We therefore have to create memory buffers on the device corresponding to the OpenCL arrays on the host, and copy the OpenCL array contents to the device buffers.

6.3.1 Kernels

We saw in section 6.2.2 that during an iteration we sometimes have to iterate over the rigid bodies and sometimes over the particles. When it comes to iterating over particles, we have two basic choices when writing our OpenCL kernels.

Our first choice is to write kernels which spawns a thread for each rigid body.

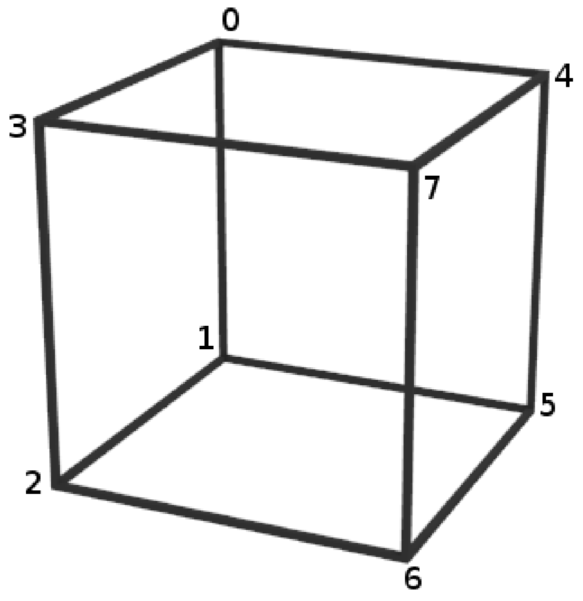

Figure 6.9: Cube wireframe with numbered vertices

In this case each thread has to iterate over all the particles of their respective rigid bodies during the iteration phases that require iteration over all particles. This results in a fewer amount of more computationally expensive kernels. Also, the iteration over particles drastically increases the number of memory accesses performed by the kernel, potentially decreasing performance.

An illustration of the resulting kernels and which parts of an iteration they complete can be seen in figure 6.10

Our second choice is to create kernels which spawn threads for each rigid body when required, but otherwise use kernels which spawn threads for each particle. This results in a larger quantity of kernels, however each kernel is less computationally expensive. An illustration of the resulting kernels and how they complete the iteration can be seen in figure 6.11.

We tried both approaches in our implementation, and discovered that the second approach, i.e. the one illustrated in figure 6.11, performed better in all circumstances, and will be the approach used in any further discussion about the implementation.

6.3.2 Grid

As can be seen in figures 6.10 and 6.11, the operation of updating the grid uses two kernels. One for resetting the grid and one for updating it. The first, which we call the `resetGrid` kernel spawn a thread for each grid cell, while the second,

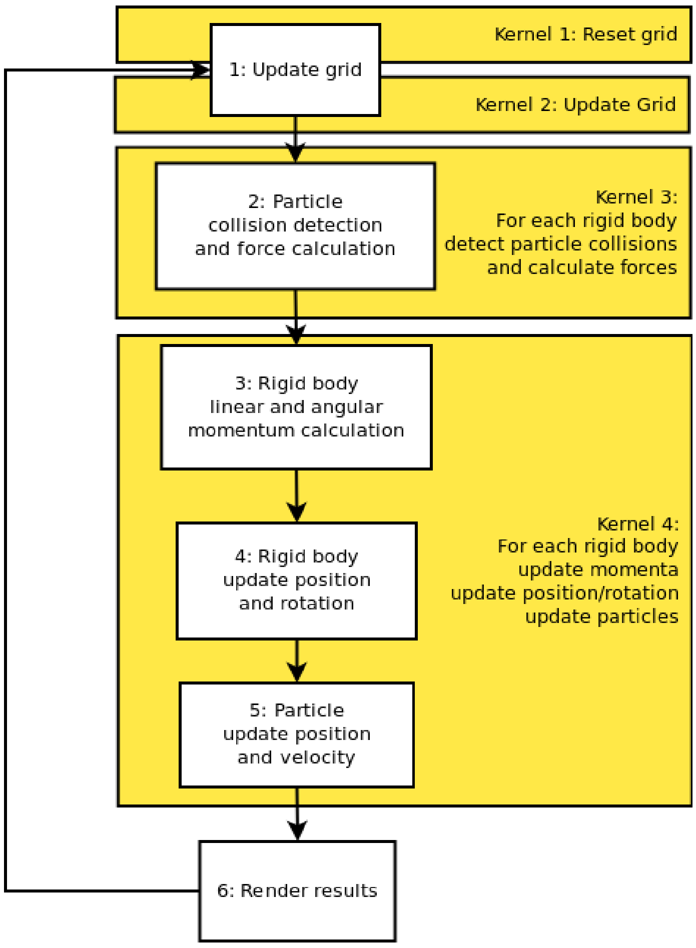
Kernel 1: Reset grid
1: Update grid
Kernel 2: Update Grid
2: Particle collision detection and force calculation
Kernel 3: For each rigid body detect particle collisions and calculate forces
3: Rigid body linear and angular momentum calculation
4: Rigid body update position and rotation
Kernel 4: For each rigid body update momenta update position/rotation update particles
5: Particle update position and velocity
6: Render results

Figure 6.10: OpenCL kernels performing a simulation iteration, with kernels iterating over rigid bodies

which we call the `updateGrid` kernel spawns a thread for each particle.

The `resetGrid` kernel simply sets every `countGrid` value to 0, and every `indexGrid` value to -1.

Then the `updateGrid` kernel updates the two arrays as explained in section 6.2.2. However, doing this update for each particle in parallel requires some synchronization. If two particles were to belong to the same grid cell and they tried to register themselves to this cell at the same time, we could experience problems. More specifically the problem occurs when both threads try to read and write to the `countGrid` array entry corresponding to the aforementioned grid cell. Each thread tries to read the value, increment it, and use the old value to know which entry of `indexGrid` to write the index of its particle to. The problem is that by using ordinary reads and writes this operation is not atomic. Another thread could read the `countGrid` value after the first thread reads it, but before the first thread increments it. The result is that both threads read

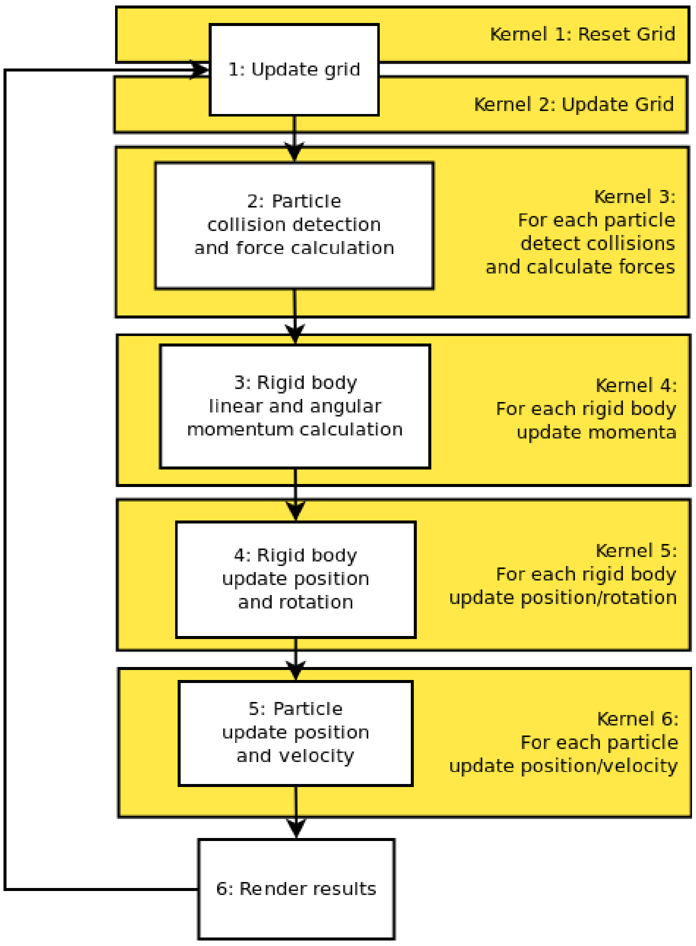
Kernel 1: Reset Grid
1: Update grid
Kernel 2: Update Grid
2: Particle collision detection and force calculation
Kernel 3: For each particle detect collisions and calculate forces
3: Rigid body linear and angular momentum calculation
Kernel 4: For each rigid body update momenta
4: Rigid body update position and rotation
Kernel 5: For each rigid body update position/rotation
5: Particle update position and velocity
Kernel 6: For each particle update position/velocity
6: Render results

Figure 6.11: OpenCL kernels performing a simulation iteration, with kernels iterating over rigid bodies when necessary, but otherwise over particles

the same value, and consequently write to the same `indexGrid` entry. This means that one unfortunate particle, the one that writes its index first, will get its index overwritten, and will not be available for collision detection by other particles.

6.3.3 Rendering

While there does not seem to be any change to the rendering according to figures 6.10 and 6.11, there is actually a very important difference in how the OpenGL VBOs are updated. In the sequential CPU version, the new values were calculated on the CPU and then copied over to the GPU for rendering. In the OpenCL version, however, since computation and rendering both happen on the GPU, it would be wasteful to copy back and forth from the host memory. We use the OpenCL/OpenGL interoperability described in section 4.2.2 to update the VBOs on the GPU directly.

Using the kernels in figure 6.11 we update the VBO for rendering cubes in kernel 5, while we update the VBO for rendering particles in kernel 6.

The final render stage is then only comprised of correctly setting vertex, color and normal pointers, and drawing the VBO arrays.

6.4 User Interface

We have created a simple user interface for our simulation using freeglut, an open source alternative to GLUT, the OpenGL Utility Toolkit [2].

The interface we have created allows us to spin around the scene and zoom in and out using the mouse. We have also added functionality for resetting the simulation without having to restart the application. By pressing the numbers 1 through 5 on the keyboard, the simulation is reset to one of five different formations. One of these is the formation shown in figure 6.6. The other options are to place the cubes in towers of various sizes. An image of these towers can be seen in figure 6.12.

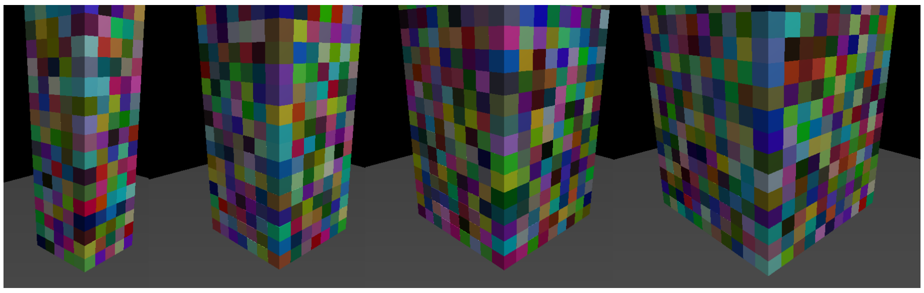

Figure 6.12: The user can reset the simulation to one of these four different size towers.

By pressing the letter *P* on the keyboard the user switches between rendering particles and cubes. A screenshot of the two different rendering options can be seen in figure 6.13.

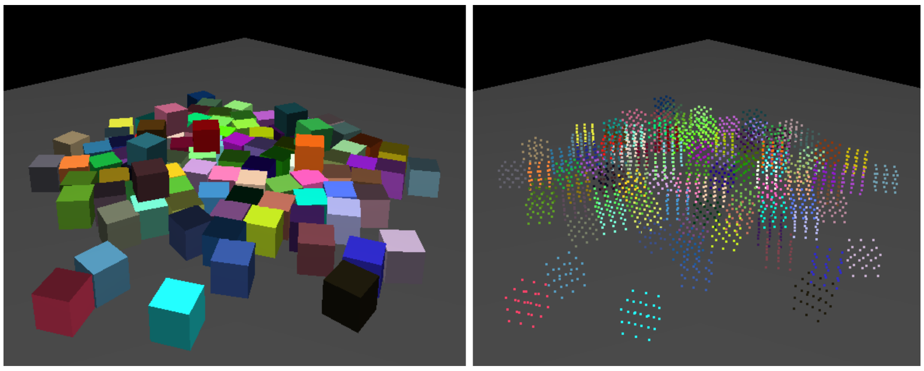

Figure 6.13: The two different render options in our application

Chapter 7

Results

In this chapter we present and discuss the results from our application. In the first section we focus on the behavior of the simulation. We explore some of the strengths and weaknesses of the choices we have made.

Next we take a look at the performance of our application, and especially how the OpenCL version performs compared to the CPU version.

Finally we compare our results to the Bullet Physics Library.

7.1 Behavior

Like we mentioned in section 5.1 people are very familiar with the movement of a rolling dice, so it's easy to visually determine whether the simulation is behaving as expected. When running our application we can indeed see that the cubes are rolling, colliding and coming to rest on one of their flat sides, as expected. A series of snapshots from a running simulation with 2000 cubes can be seen in figure 7.1. This is too many cubes to run in real time on the CPU, however the GPU can easily handle it. We will be explored in more detail in section 7.2

We have also recorded and uploaded two videos demonstrating our application.

- `http://www.youtube.com/watch?v=DvgEk_8DF0k`

  This video shows a simulation of 300 cubes running on an AMD X6 1055T CPU.

- `http://www.youtube.com/watch?v=48vq0GJaDgc`

  This video shows a simulation of 3000 cubes running on an AMD HD5870 GPU.

The videos illustrate the basic movement and behavior of the cube. They demonstrate how the user can spin around the simulation, zoom in and out, and

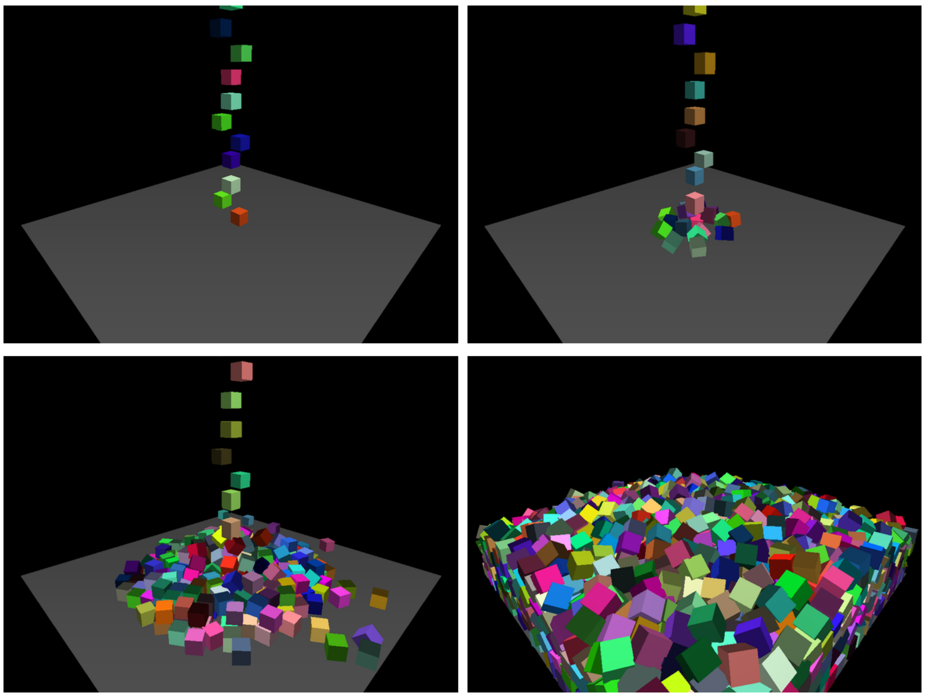
Figure 7.1: Four snapshots from a running simulation

reset the simulation to several different starting positions. They also demonstrate how the user can switch between the two different rendering methods, rendering cubes, and rendering particles.

The videos also show how the cubes are unable to stay stacked in a large tower, causing a collapse. In the rest of this section we will explore this issue, as well as some other problems with our simulation.

Let us first begin with the inability to stack properly. We mentioned this already in section 5.1, and it was indeed one of the reasons for choosing cubic rigid bodies for our simulation.

The inability to stack is caused by a combination of factors. We think one factor is the way forces are modeled between particles. Since they are modeled by a spring, stacking multiple cubes on top of each other is like creating a long vertical spring. This will inevitably become unstable, especially when the cubes are not placed perfectly straight on top of each other. This brings us over to another factor, which is probably the most influential one and has to do with the particle approach itself. Since the cubes are made out of particles, their sides are not completely flat, and when cubes rest on top of each other they will slide into the grooves created by the spherical particles. An illustration of this can be seen in figure 7.2.

An example of how this looks in the application itself when rendering cubes can be seen in figure 7.3.

Needless to say, this is not ideal behavior, and is one of the prices that has to

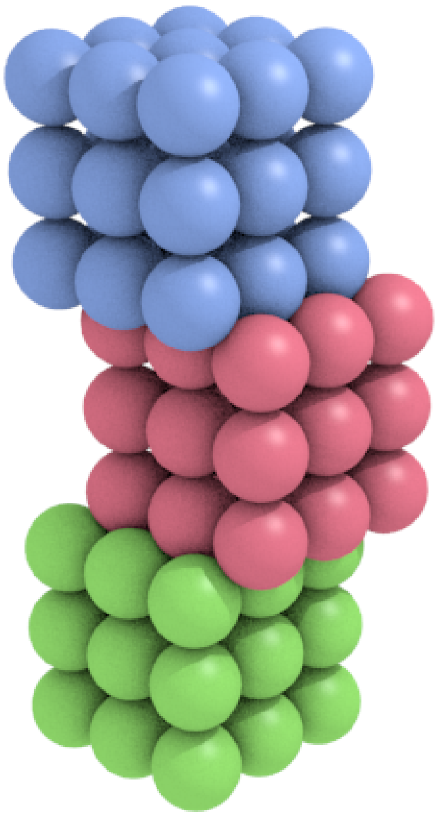

Figure 7.2: Particle cubes stacked on top of each other

be paid when using this particle approach.

Another problem we discovered with the particle approach was that the particle cubes could get interlocked with each other. An illustration of how this happens can be seen in figure 7.4, and an example of how it looks in the simulation can be seen in the screenshot in figure 7.5. Although not visible in these pictures, the biggest problem with this scenario is actually not the interlocking itself, but rather the erratic movement exercised by the interlocked bodies. This is caused by the fact that there is not enough room for the entire between the particles in the other rigid body. The particle will be pushed back and forth, and that same force will affect the rigid body.

The best way to fix this problem would be to prevent the interlocking from occuring in the first place. One way to do this is to limit the mass or speed of the rigid bodies. The rigid bodies need a certain amount of momentum in order for their particles to force their way into another rigid body. However, this is not an ideal solution since we would like to be able to simulate bodies of any

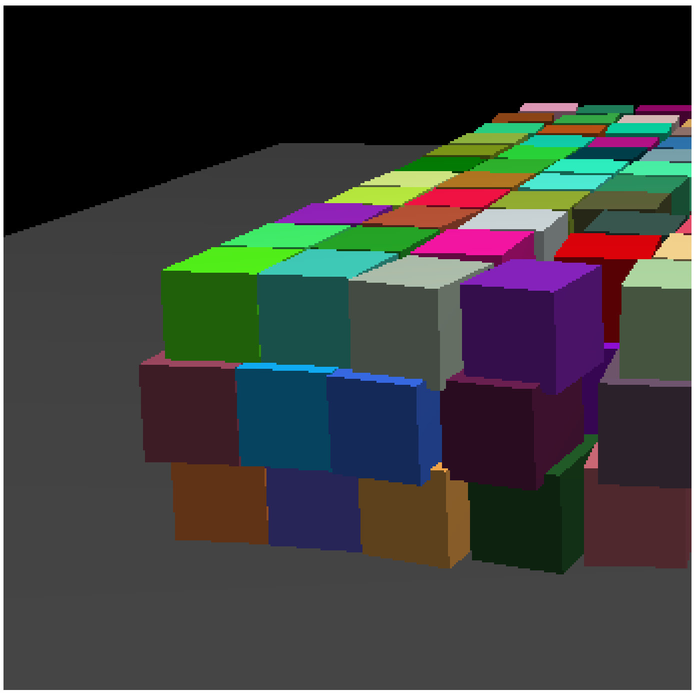

Figure 7.3: Screenshot of simulation where cubes are stacked on top of each other

mass and speed.

Using a different model for the forces between particles may produce a more stable situation when the interlock occurs. This could also help with the stacking problem mentioned earlier, and could an interesting topic for future work.

Fortunately this does not occur very often during normal conditions in our simulation. However, the final issue we will take a look at in this section unfortunately causes it to happen quite often.

Our final issue is what happens when a tower of cubes goes outside the grid borders. In our implementation we choose to ignore particles that are not inside the grid when it comes to collision detection. In our first simulation state this is not a problem since cubes will never collide outside the grid unless we create so many cubes that they eventually fill up the entire grid. However, when creating a tower, the cubes at the top of the grid will stay put, since they are standing on top of the cubes directly underneath. The cubes above them then fall into them, since they are not considered in collision detection. The result is that a layer of cubes start clumping up at the top of the grid. A screenshot of this in action can be seen in figure 7.6.

A possible solution to this could be to calculate collisions for particles outside the grid as if there were no grid at all. However, this would slow the simulation down to a halt when many cubes are outside the grid. We do not really think this is that much of a problem, since if this simulation was applied in any real scenario there should not be any cubes outside the grid at all.

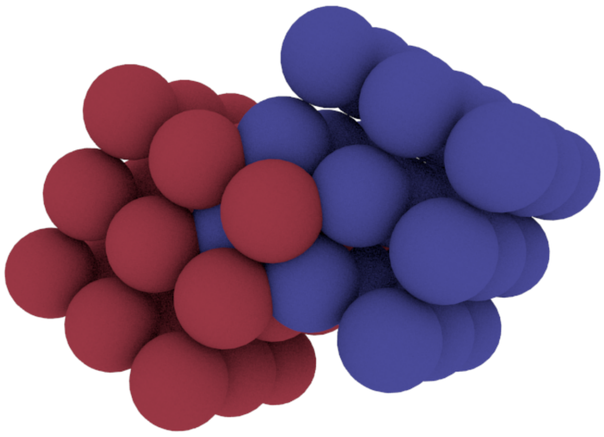
Figure 7.4: Two particle cubes interlocked

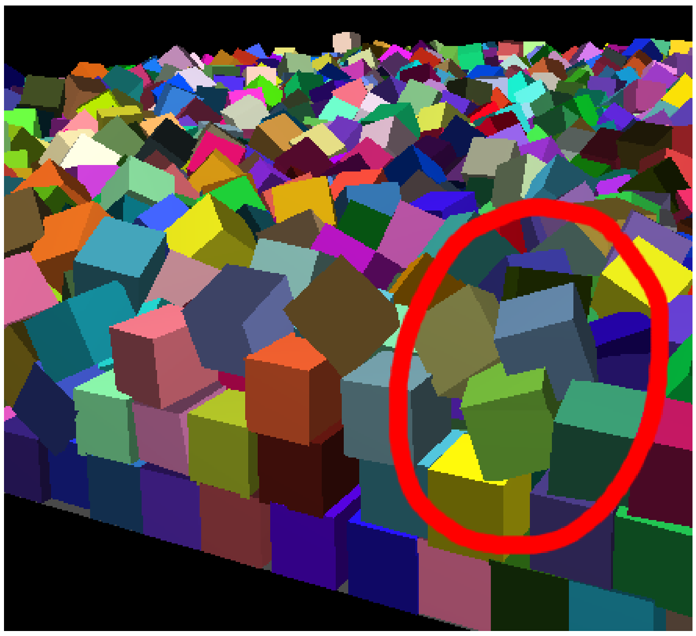
Figure 7.5: Screenshot of simulation where two cubes are interlocked

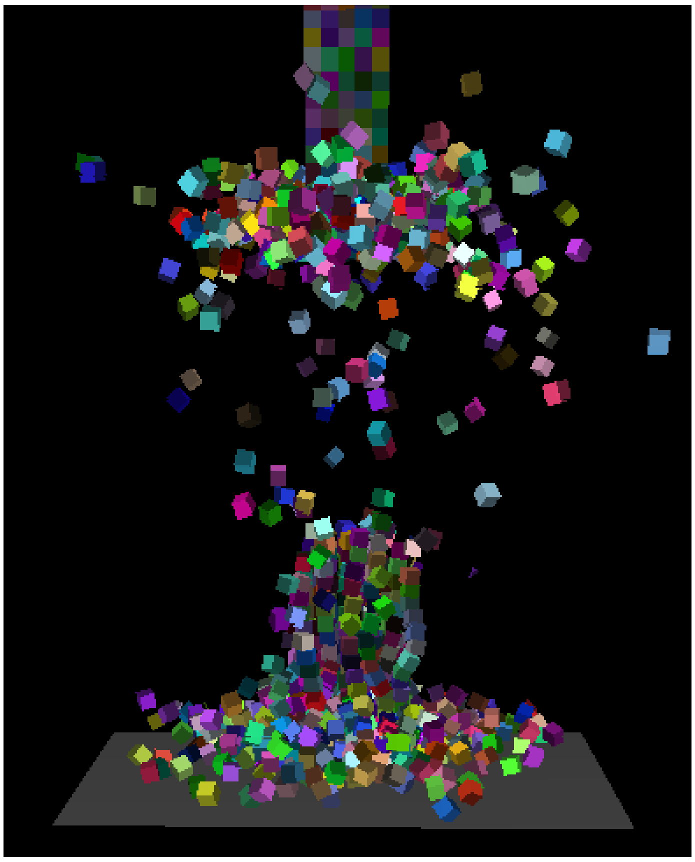

Figure 7.6: Screenshot of cubes colliding and clumping up at the grid border

7.2 Performance

To measure performance we use timer functions included in GLUT. One of the functions allows us to get the time elapsed since the application started. We get this time at the beginning of a frame, and then again at the end of a frame. The difference tells us how long this particular frame lasted. We then add this time to a sum of frame times. Every 50 frames we use this sum to calculate the average framerate, or FPS (*frames per second*), during the time used by those 50 frames. For the performance measurements in this section, we wait until all cubes are inside the grid, and then calculate an average FPS from ten of these 50 frame intervals.

Our implementation provides a stable simulation when the timestep is set to $0.01s$. In order for the simulation to be real-time it needs to render 100 frames per second.

All the tests were run on a computer with an AMD X6 1055T CPU, 4 GB of RAM, an AMD Radeon HD 5870 graphics card with Catalyst 10.12 drivers, and running Ubuntu Linux 10.10 32-bit.

The parameters used in the test simulation can be seen in table 7.1. These parameters create a nice and stable simulation.

Table 7.1: Parameters used for simulation performance testing

Parameter name	*Value*
worldSize	15.0
springCoefficient	100.0
dampingCoefficient	0.5
timeDelta	0.01
particleRadius	0.20
terminalVelocity	20.0

7.2.1 Sequential CPU version

A purely sequential version of the simulation running on the CPU manages to simulate 300 cubes at 100 frames per second, which corresponds to 8100 particles.

Performance results for four selected problem sizes can be seen in table 7.2 and figure 7.7.

Table 7.2: CPU performance for 4 selected problem sizes

Number of cubes	100	500	1000	3000
Frames per second	153.2	56.0	29.1	9.2

7.2.2 OpenCL version

The OpenCL version running on an AMD Radeon HD5870 manages to simulate 3300 cubes at 100 frames per second. This corresponds to a total of 89100 particles.

Performance results for four selected problem sizes can be seen in table 7.3 and figure 7.7

Table 7.3: GPU performance for 4 selected problem sizes

Number of cubes	100	500	1000	3000
Frames per second	571.4	377.8	253.3	110.5

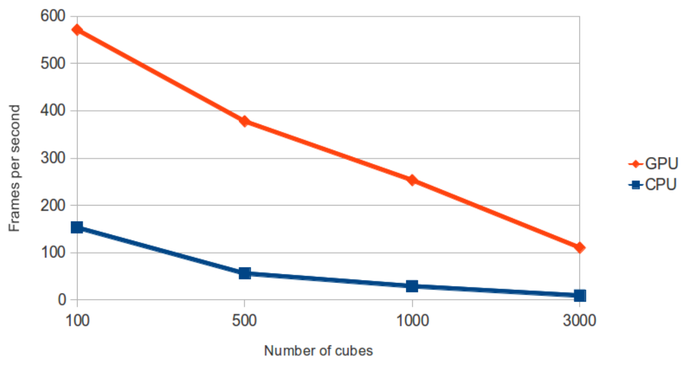

Figure 7.7: Graph showing difference in performance between CPU and GPU for 4 selected problem sizes

The GPU speedup over CPU from these test runs can be seen in table 7.4 and figure 7.8.

Table 7.4: GPU speedup over CPU from test runs

Number of cubes	100	500	1000	3000
Speedup	3.7	6.7	8.7	12.0

As expected we can see that the speedup rises as the problem size increases, which is in accordance with Gustafson's law.

7.3 Bullet Comparison

We have also implemented a variation of our application which uses the Bullet Physics Library to perform the rigid body simulation. All other parts of the application such as rendering and user interface remain the same. The first thing to notice about the Bullet simulation is that it uses the actual geometry of the cubes, and thus provides a more accurate simulation than our own. It

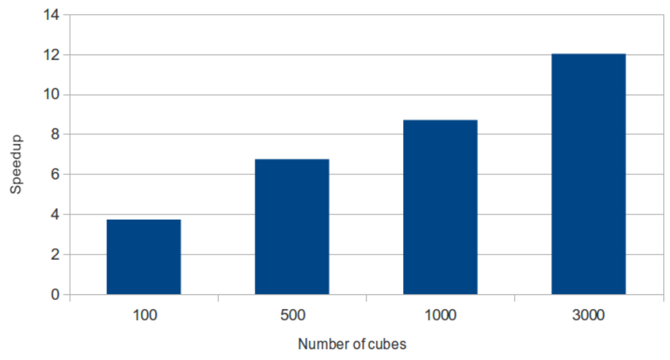

Figure 7.8: Graph showing GPU speedup over CPU simulation at various problem sizes

also supports stacking cubes far better, as can be seen in the screenshot in figure 7.9.

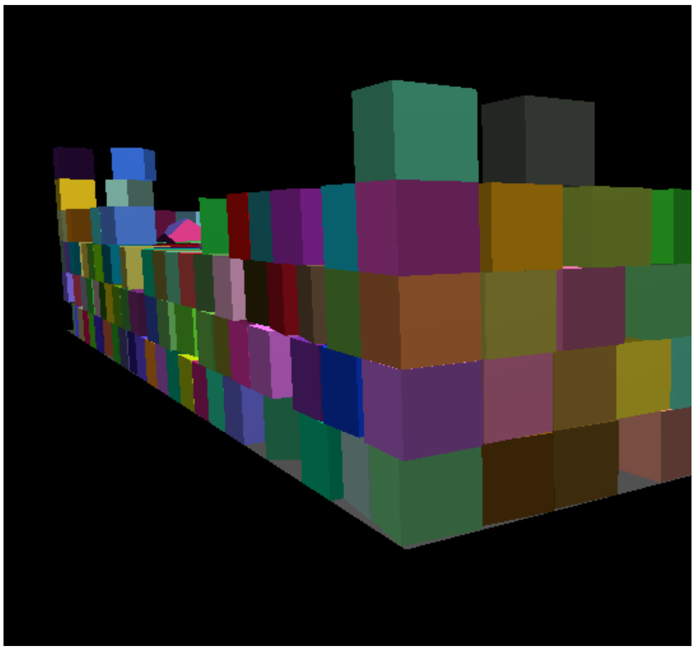

Figure 7.9: Screenshot of stacked cubes in Bullet rigid body simulation

Since the behaviour of the Bullet simulation is better than our simulation, our simulation should hopefully at least have better performance.

We test the performance of the Bullet simulation the same way we did our own, and compare the results. One thing we quickly discover is that the performance of the Bullet simulation greatly increases once the rigid bodies settle down and stop moving. This is likely the result of some sort of optimization performed by Bullet. When we measure the performance of the Bullet simulation we therefore test 2 times for each problem size. Once while the rigid bodies are still moving around, and then once after they have stopped moving. The resulting framerate results can be seen in table 7.5.

Table 7.5: Performance of Bullet rigid body simulation on CPU

Number of cubes	100	500	1000	3000
FPS during movement	417.2	132.1	44.8	11.0
FPS when still	796.7	381.3	136.4	55.2

The performance when the bodies are still greatly outperform our own, however, moving rigid bodies are far more interesting so we will focus on that. Bullet still outperforms our CPU simulation while the problem size is small, but the difference shrinks as the problem size increases. This probably has to do with the broadphase, the way collision detection is reduced by excluding bodies that cannot possibly be intersecting. At low problem sizes there are few collisions and Bullet successfully excludes many bodies.

While our grid stucture performs much the same function we currently have no way of distinguishing particles belonging to the same rigid body as those belonging to different ones. Therefore, even while no collisions are going on, all the particles discover several nearby particles which they check for collisions.

Although the Bullet simulation seems better than ours in all aspects, we may have given ourselves some unfair advantages. It turns out Bullet has functions written specifically for cuboid objects which are faster than other general shapes. One of the advantages of the particle approach is that the formation of the particles does not affect performance, and should therefore be good for simulating objects with shapes that are less efficient than cubes. While cubes are good for checking whether the simulation is behaving as expected, they are not what would ideally be simulated in a practical application. Secondly, we could have used a fewer number of particles to represent our bodies. A single particle for each cube would be no good, since it would never gain any rotation, however a 2x2x2 cube would work fine. This would of course lower the accuracy of the simulation, and cause even worse stacking problems. It might however be worth it if performance is important and accuracy is not. Lastly, Bullet is a big project which has been developed for years and is well optimized. Our simulation has received very little attention when it comes to optimization, so it is safe to say that there is a lot of untapped potential when it comes to performance.

Bullet is able to use OpenCL to accelerate simulations, however this is not supported under Linux. Our application currently only runs under Linux so we were unfortunately unable to try OpenCL Bullet to compare with our own OpenCL simulation.

Chapter 8

Conclusions

In this chapter we summarize what we have accomplished and what we have learned from it. We also discuss how the work could be improved by future work.

8.1 Summary

We have implemented a rigid body simulation using a particle approach in the hope of achieving high performance. We have shown that our simulation behaves well, with the exception of some acceptable expected problems. We discovered that the performance of our simulation running on a CPU is comparable to that of the Bullet Physics Engine for larger problem sizes. And, last but not least, we have managed to accelerate our simulation by up to a factor of 12 by porting it to OpenCL and running it on a GPU.

We believe the methods we have used for our rigid body simulation could be applied well to visual elements in video games that do not affect the gameplay itself. Such elements do not require a lot of accuracy, but it is important that they look and feel correct. Certain objects are better suited for the approach than others however. Using fewer particles to represent an object will result in a faster simulation. Therefore, objects that can be represented by few particles while still be able to largely retain their behavior are best. The objects should also not be able to stack easily, since stacking is not well supported by the particle approach.

Well suited objects tend to be rounded, and not too thin. An example of a very well suited object can be seen in figure 8.1.

An example of an object which is very poorly suited to the particle approach can be seen in figure 8.2.

This object, a desk, requires a lot of particles, which results in sub-optimal performance. The biggest problem is still that a desk should be able to have things placed on top of it, which is poorly supported by the particle method.

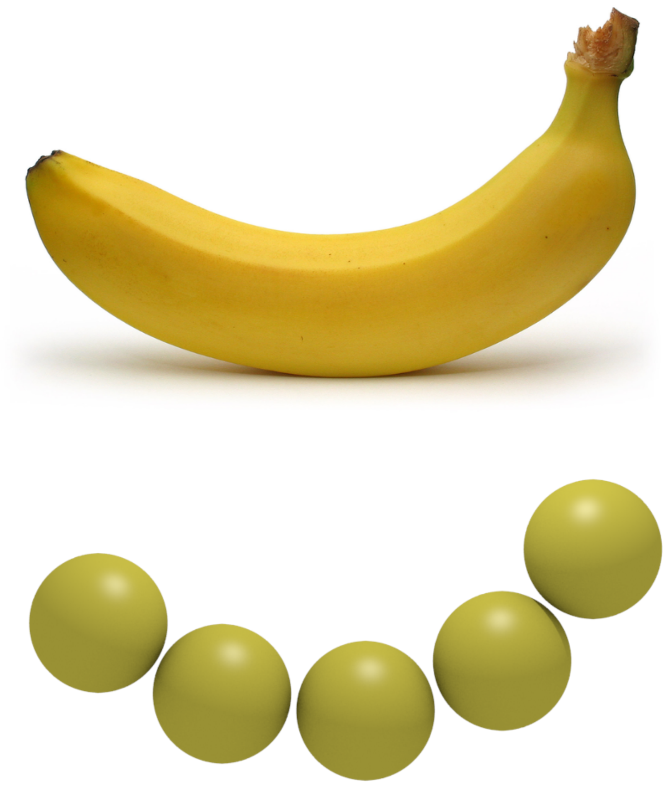

Figure 8.1: A banana is very well suited for particle representation.

Since many objects are so ill suited, the viability of the particle approach probably largely rests on its ability to interact with other rigid body methods. With different methods interacting in the same simulation, the particle method could deal with the small clumpy items such as bananas, while geometry based methods deals with the remaining objects.

8.2 Future work

There are a lot of ways in which our simulation could be improved and explored further. First of all is the interaction with geometry based methods, as mentioned in the previous section.

However, the simulation can also be improved as a standalone simulation in a number of ways. We currently use a simple Euler integration. Using a more sophisiticated integration method, such as Runge-Kutta, the simulation should become more stable and accurate. This could allow the simulation to use a larger timestep. This reduces the framerate needed to produce a real-time simulation, and might allow a larger problem size to be simulated in real-time.

Finding a different way to model the forces between particles might also increase

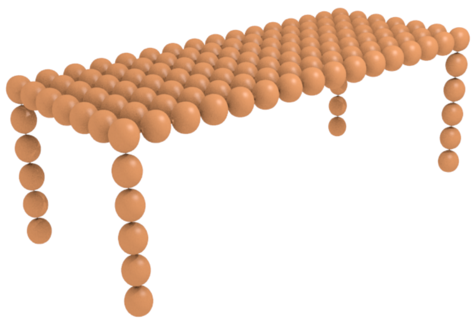

Figure 8.2: A desk is very poorly suited for particle representation.

the stability of the simulation, especially when it comes to stacking objects. Our simulation also lacks friction, which could make the behavior of the rigid bodies more realistic. With friction, the cubes in figure 8.3 will be at rest and stay in their current position. This is how the objects would behave in real life. However, in our simulation, since there is no friction, the middle cube will slowly push the other two cubes away from it, creating a space where it eventually slides down.

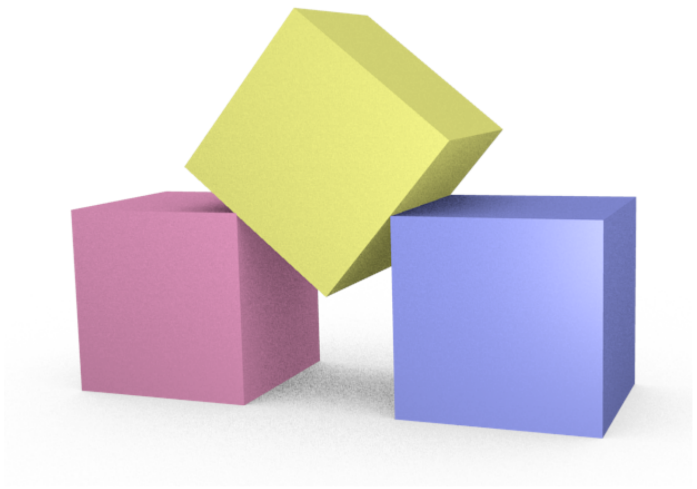

Figure 8.3: With friction, these 3 cubes will rest in the current position. Without friction, the middle cube will slowly slide down and push the other 2 cubes to the side.

Currently our simulation only supports cubes, and we use the assumption that there are only 27 particles in each rigid body in order to calculate which particle is the first particle of a rigid body based on its index. Extending the simulation to support arbitrarily shaped rigid bodies, with an arbitrary number of particles is an important next step. One way to do this could be to keep an additional array which holds the index of the first particle of each rigid body.

Lastly, of course, things can always go faster. We have not spent much time optimizing our code, so there is sure to be a lot of untapped potential in our simulation when it comes to performance.

Bibliography

[1] Bullet physics library. URL `http://www.bulletphysics.org/`.

[2] freeglut. URL `http://freeglut.sourceforge.net/`.

[3] Rahil Baber. Rigid body simulation. Master's thesis, University of Warwick, 2006.

[4] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18:1 – 33, 2010.

[5] Erwin Coumans. *Bullet 2.76 Physics SDK Manual*, 2010.

[6] Simon Green. *Particle Simulation using CUDA*. NVIDIA, May 2010.

[7] Takahiro Harada. *Real-Time Rigid Body Simulation on GPUs*, chapter 29. Addison-Wesley, 2007.

[8] *NVIDIA CUDA C Programming Guide*. NVIDIA, 3.2 edition, September 2010.

[9] M. Paterson and F. F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5:485 – 503, 1990.

[10] Jr. Richard S. Wright, Nicholas Haemel, Graham Sellers, and Benjamin Lipchak. *OpenGL SuperBible*. Addison-Wesley, fifth edition, November 2010.

[11] Masayuki Tanaka, Mikio Sakai, Ishikawajima-Harima, and Seiichi Koshizuka. Rigid body simulation using a particle method. *ACM SIGGRAPH 2006 Research posters*, 2006.

[12] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. *The OpenCL Programming Book*. Fixstars Corporation, first edition, April 2010.

Appendix A

Source Code Samples

A.1 Kernels

```
__kernel
void resetGrid(
  __global int* countGrid,
  __global int4* indexGrid
) {
  unsigned int flatGridIndex = get_global_id(0);

  countGrid[flatGridIndex] = 0;
  indexGrid[flatGridIndex].x = -1;
  indexGrid[flatGridIndex].y = -1;
  indexGrid[flatGridIndex].z = -1;
  indexGrid[flatGridIndex].w = -1;

}

__kernel
void updateGrid(
  __global int* countGrid,
  __global int4* indexGrid,
  __global float3* particlePosition,
  float3 minimumPosition,
  float voxelSideLength,
  int gridSideLength,
  __global int3* particleGridIndex
) {
  unsigned int particleIndex = get_global_id(0);

  particleGridIndex[particleIndex].x =  (int)((particlePosition[
      particleIndex].x - minimumPosition.x) / voxelSideLength);
  particleGridIndex[particleIndex].y =  (int)((particlePosition[
      particleIndex].y - minimumPosition.y) / voxelSideLength);
  particleGridIndex[particleIndex].z =  (int)((particlePosition[
      particleIndex].z - minimumPosition.z) / voxelSideLength);

  bool validIndex = (particleGridIndex[particleIndex].x > 0) &&
    (particleGridIndex[particleIndex].x < gridSideLength-1) &&
    (particleGridIndex[particleIndex].y > 0) &&
    (particleGridIndex[particleIndex].y < gridSideLength-1) &&
    (particleGridIndex[particleIndex].z > 0) &&
    (particleGridIndex[particleIndex].z < gridSideLength-1);

  if (validIndex) {
    int xStride = gridSideLength * gridSideLength;
    int yStride = gridSideLength;
    int flatGridIndex = particleGridIndex[particleIndex].x*xStride
        +
      particleGridIndex[particleIndex].y * yStride +
      particleGridIndex[particleIndex].z;

    int particlesInCell = atomic_inc(&countGrid[flatGridIndex]);

    if (particlesInCell == 3) {
      indexGrid[flatGridIndex].w = particleIndex;
    } else if (particlesInCell == 2) {
      indexGrid[flatGridIndex].z = particleIndex;
    } else if (particlesInCell == 1) {
      indexGrid[flatGridIndex].y = particleIndex;
    } else if (particlesInCell == 0) {
      indexGrid[flatGridIndex].x = particleIndex;
    }
  }
}


__kernel
void collisionDetection(
  __global float3* particleMass,
  __global float3* particlePosition,
  __global float3* particleVelocity,
  __global float3* particleForce,
  float particleRadius,
  float worldSize,
  float springCoefficient,
  float dampingCoefficient,
  __global int3* particleGridIndex,
  __global int* countGrid,
  __global int4* indexGrid,
  int gridSideLength
) {

  unsigned int particleIndex = get_global_id(0);

  particleForce[particleIndex].x = 0.0f;
  particleForce[particleIndex].y = 0.0f;
  particleForce[particleIndex].z = 0.0f;

  int3 gridIndex = particleGridIndex[particleIndex];

  //Pretend border cell is 1 position inwards to avoid checking
      outside bounds for neighbors
  gridIndex = clamp(gridIndex, 1, gridSideLength-2);

  int xStride = gridSideLength*gridSideLength;
  int yStride = gridSideLength;

  int flatGridIndex = gridIndex.x * xStride +
    gridIndex.y * yStride + gridIndex.z;

  int4 neighborCells[27];
  int cellIndexJ = 0;
```

```
flatGridIndex -= xStride;
flatGridIndex += 2*yStride;
flatGridIndex += 2; //zStride


for (int x=0; x<3; x++) {
  flatGridIndex -= 3*yStride;

  for (int y=0; y<3; y++) {
    flatGridIndex -= 3;

    for (int z=0; z<3; z++) {

      neighborCells[cellIndexJ] = indexGrid[flatGridIndex];
      cellIndexJ++;
      flatGridIndex++;
    }
    flatGridIndex += yStride;
  }
  flatGridIndex += xStride;
}

for (int j=0; j<27; j++) {
  int neighborParticles[4] = {
    neighborCells[j].x,
    neighborCells[j].y,
    neighborCells[j].z,
    neighborCells[j].w };

  for (int k=0; k<4; k++) {
    int otherParticle = neighborParticles[k];
    if ((otherParticle != particleIndex) && (otherParticle !=
        (-1))) {
      float3 distance = particlePosition[otherParticle] -
          particlePosition[particleIndex];

      float absDistance = sqrt(distance.x*distance.x +
          distance.y*distance.y +
          distance.z*distance.z);

      if ((absDistance + 0.000001f) < (2.0f * particleRadius)) {
        particleForce[particleIndex].x -= springCoefficient*
          (particleRadius+particleRadius - absDistance)*(distance
              .x/absDistance);
        particleForce[particleIndex].y -= springCoefficient*
          (particleRadius+particleRadius - absDistance)*(distance
              .y/absDistance);
        particleForce[particleIndex].z -= springCoefficient*
          (particleRadius+particleRadius - absDistance)*(distance
              .z/absDistance);

        float3 relativeVelocity = {
          particleVelocity[otherParticle].x - particleVelocity[
              particleIndex].x,
          particleVelocity[otherParticle].y - particleVelocity[
              particleIndex].y,
          particleVelocity[otherParticle].z - particleVelocity[
              particleIndex].z };

        particleForce[particleIndex].x += dampingCoefficient*
            relativeVelocity.x;
        particleForce[particleIndex].y += dampingCoefficient*
```

```
            relativeVelocity.y;
        particleForce[particleIndex].z += dampingCoefficient*
            relativeVelocity.z;

      }
    }
  }
}


//Boundary forces
{
  bool collisionOccured = false;
  // Ground collision
  if (particlePosition[particleIndex].y-particleRadius < 0.0f) {
    collisionOccured = true;
    particleForce[particleIndex].y += springCoefficient*
      (particleRadius-particlePosition[particleIndex].y);
  }

  // X-axis Wall Collision
  if (particlePosition[particleIndex].x-particleRadius < -
      worldSize) {
    collisionOccured = true;
    particleForce[particleIndex].x += springCoefficient*
      (-worldSize - particlePosition[particleIndex].x +
          particleRadius);

  } else if (particlePosition[particleIndex].x+particleRadius >
      worldSize) {
    collisionOccured = true;
    particleForce[particleIndex].x += springCoefficient*
      (worldSize - particlePosition[particleIndex].x -
          particleRadius);
  }

  // Z-axis Wall Collision
  if (particlePosition[particleIndex].z-particleRadius < -
      worldSize) {
    collisionOccured = true;
    particleForce[particleIndex].z += springCoefficient*
      (-worldSize - particlePosition[particleIndex].z +
          particleRadius);

  } else if (particlePosition[particleIndex].z+particleRadius >
      worldSize) {
    collisionOccured = true;
    particleForce[particleIndex].z += springCoefficient*
      (worldSize - particlePosition[particleIndex].z -
          particleRadius);
  }

  // Damping
  if (collisionOccured) {
    particleForce[particleIndex].x -= dampingCoefficient*
        particleVelocity[particleIndex].x;
    particleForce[particleIndex].y -= dampingCoefficient*
        particleVelocity[particleIndex].y;
    particleForce[particleIndex].z -= dampingCoefficient*
        particleVelocity[particleIndex].z;
  }
}
```

```
}


__kernel
void updateMomenta(
    __global float* mass,
    __global float3* force,
    __global float3* position,
    __global float3* linearMomentum,
    __global float3* angularMomentum,
    __global float3* particlePosition,
    __global float3* particleForce,
    float delta,
    float terminalVelocity
) {
  unsigned int bodyIndex = get_global_id(0);
  unsigned int totalNumberOfParticles = get_global_size(0) * 27;
  unsigned int particleIndex = bodyIndex * 27;

  force[bodyIndex].x = 0.0f;
  force[bodyIndex].y = mass[bodyIndex] * -9.81f; //force of gravity
  force[bodyIndex].z = 0.0f;

  float3 torque = {0.0f, 0.0f, 0.0f};

  //Calculate body force and torque
  for (int i=0; i<27; i++) {
    force[bodyIndex] += particleForce[particleIndex+i];
    float3 relativePos = particlePosition[particleIndex+i] -
        position[bodyIndex];
    torque += cross(relativePos, particleForce[particleIndex+i]);
  }

  float terminalMomentum = terminalVelocity * mass[bodyIndex];

  linearMomentum[bodyIndex].x += force[bodyIndex].x * delta;
  linearMomentum[bodyIndex].y += force[bodyIndex].y * delta;
  linearMomentum[bodyIndex].z += force[bodyIndex].z * delta;

  //Limit momentum by terminal momentum
  linearMomentum[bodyIndex] = clamp(linearMomentum[bodyIndex], -
      terminalMomentum, terminalMomentum);

  angularMomentum[bodyIndex].x += torque.x * delta;
  angularMomentum[bodyIndex].y += torque.y * delta;
  angularMomentum[bodyIndex].z += torque.z * delta;
}

__kernel
void performStep(
  __global float* mass,
  __global float3* position,
  __global float3* velocity,
  __global float3* linearMomentum,
  __global float4* quaternion,
  __global float* rotationMatrix,
  __global float3* angularVelocity,
  __global float3* angularMomentum,
  __global float3* initialIITDiagonal,
  __global float* inverseInertiaTensor,
  __global float* bodyVBO,
  float delta,
  float particleRadius,
  int bodyVBOStride
) {
  unsigned int bodyIndex = get_global_id(0);
  unsigned int bodyVBOIndex = bodyIndex * 24 * 3;
  unsigned int matrixIndex = bodyIndex * 9;

  //Update inverse inertia tensor
  {
    float a = rotationMatrix[matrixIndex];
    float b = rotationMatrix[matrixIndex+1];
    float c = rotationMatrix[matrixIndex+2];
    float d = rotationMatrix[matrixIndex+3];
    float e = rotationMatrix[matrixIndex+4];
    float f = rotationMatrix[matrixIndex+5];
    float g = rotationMatrix[matrixIndex+6];
    float h = rotationMatrix[matrixIndex+7];
    float i = rotationMatrix[matrixIndex+8];

    float u = initialIITDiagonal[bodyIndex].x;
    float v = initialIITDiagonal[bodyIndex].y;
    float w = initialIITDiagonal[bodyIndex].z;

    inverseInertiaTensor[matrixIndex] = u*a*a + b*b*v + c*c*w;
    inverseInertiaTensor[matrixIndex+1] = a*d*u + b*e*v + c*f*w;
    inverseInertiaTensor[matrixIndex+2] = a*g*u + b*h*v + c*i*w;
    inverseInertiaTensor[matrixIndex+3] = a*d*u + b*e*v + c*f*w;
    inverseInertiaTensor[matrixIndex+4] = u*d*d + e*e*v + f*f*w;
    inverseInertiaTensor[matrixIndex+5] = d*g*u + e*h*v + f*i*w;
    inverseInertiaTensor[matrixIndex+6] = a*g*u + b*h*v + c*i*w;
    inverseInertiaTensor[matrixIndex+7] = d*g*u + e*h*v + f*i*w;
    inverseInertiaTensor[matrixIndex+8] = u*g*g + h*h*v + i*i*w;
  }

  //Perform linear step
  {
    velocity[bodyIndex].x = linearMomentum[bodyIndex].x / mass[
        bodyIndex];
    velocity[bodyIndex].y = linearMomentum[bodyIndex].y / mass[
        bodyIndex];
    velocity[bodyIndex].z = linearMomentum[bodyIndex].z / mass[
        bodyIndex];

    position[bodyIndex].x += velocity[bodyIndex].x * delta;
    position[bodyIndex].y += velocity[bodyIndex].y * delta;
    position[bodyIndex].z += velocity[bodyIndex].z * delta;
  }

  //Perform angular step
  {
    //Update angular velocity
    {
      float a = inverseInertiaTensor[matrixIndex];
      float b = inverseInertiaTensor[matrixIndex+1];
      float c = inverseInertiaTensor[matrixIndex+2];
      float d = inverseInertiaTensor[matrixIndex+3];
      float e = inverseInertiaTensor[matrixIndex+4];
      float f = inverseInertiaTensor[matrixIndex+5];
      float g = inverseInertiaTensor[matrixIndex+6];
      float h = inverseInertiaTensor[matrixIndex+7];
      float i = inverseInertiaTensor[matrixIndex+8];
```

```
        float u = angularMomentum[bodyIndex].x;
        float v = angularMomentum[bodyIndex].y;
        float w = angularMomentum[bodyIndex].z;

        angularVelocity[bodyIndex].x = a*u + b*v + c*w;
        angularVelocity[bodyIndex].y = d*u + e*v + f*w;
        angularVelocity[bodyIndex].z = g*u + h*v + i*w;
      }
      float angularVelocitySize = sqrt(
          angularVelocity[bodyIndex].x*angularVelocity[bodyIndex].x +
          angularVelocity[bodyIndex].y*angularVelocity[bodyIndex].y +
          angularVelocity[bodyIndex].z*angularVelocity[bodyIndex].z )
              ;

      if (angularVelocitySize > 0.0f) {
        float3 rotationAxis = {
          angularVelocity[bodyIndex].x / angularVelocitySize,
          angularVelocity[bodyIndex].y / angularVelocitySize,
          angularVelocity[bodyIndex].z / angularVelocitySize};

        float rotationAngle = angularVelocitySize*delta;

        float ds = cos(rotationAngle/2.0f);
        float dvx = rotationAxis.x*sin(rotationAngle/2.0f);
        float dvy = rotationAxis.y*sin(rotationAngle/2.0f);
        float dvz = rotationAxis.z*sin(rotationAngle/2.0f);

        float s = quaternion[bodyIndex].x;
        float vx = quaternion[bodyIndex].y;
        float vy = quaternion[bodyIndex].z;
        float vz = quaternion[bodyIndex].w;

        quaternion[bodyIndex].x = s*ds - vx*dvx - vy*dvy - vz*dvz;
        quaternion[bodyIndex].y = ds*vx + s*dvx + dvy*vz - dvz*vy;
        quaternion[bodyIndex].z = ds*vy + s*dvy + dvz*vx - dvx*vz;
        quaternion[bodyIndex].w = ds*vz + s*dvz + dvx*vy - dvy*vx;
      }
    }

    //Update rotation matrix
    {
      //Normalize quaternion
      {
        float mag2 = quaternion[bodyIndex].x*quaternion[bodyIndex].x
            +
          quaternion[bodyIndex].y*quaternion[bodyIndex].y +
          quaternion[bodyIndex].z*quaternion[bodyIndex].z +
          quaternion[bodyIndex].w*quaternion[bodyIndex].w;

        if (mag2!=0.0f && (fabs(mag2 - 1.0f) > 0.00001f)) {
          float mag = sqrt(mag2);
          quaternion[bodyIndex].x /= mag;
          quaternion[bodyIndex].y /= mag;
          quaternion[bodyIndex].z /= mag;
          quaternion[bodyIndex].w /= mag;
        }
      }
      float w = quaternion[bodyIndex].x;
      float x = quaternion[bodyIndex].y;
      float y = quaternion[bodyIndex].z;
      float z = quaternion[bodyIndex].w;
    float xx = x * x;
    float yy = y * y;
    float zz = z * z;
    float xy = x * y;
    float xz = x * z;
    float yz = y * z;
    float wx = w * x;
    float wy = w * y;
    float wz = w * z;

    rotationMatrix[matrixIndex] = 1.0f-2.0f*(yy+zz);
    rotationMatrix[matrixIndex+1] = 2.0f*(xy-wz);
    rotationMatrix[matrixIndex+2] = 2.0f*(xz+wy);
    rotationMatrix[matrixIndex+3] = 2.0f*(xy+wz);
    rotationMatrix[matrixIndex+4] = 1.0f-2.0f*(xx+zz);
    rotationMatrix[matrixIndex+5] = 2.0f*(yz-wx);
    rotationMatrix[matrixIndex+6] = 2.0f*(xz-wy);
    rotationMatrix[matrixIndex+7] = 2.0f*(yz+wx);
    rotationMatrix[matrixIndex+8] = 1.0f-2.0f*(xx+yy);

}

//Update body VBO
{

    float side = particleRadius * 3.0f;

    float normals[6][3] = { // Cube face normals
      { -1.0f, 0.0f, 0.0f }, { 0.0f, 1.0f, 0.0f }, { 1.0f, 0.0f,
          0.0f },
      { 0.0f, -1.0f, 0.0f }, { 0.0f, 0.0f, 1.0f }, { 0.0f, 0.0f,
          -1.0f } };
    int faceIndices[6][4] = { // Cube faces' vertex indices
      { 0, 1, 2, 3 }, { 3, 2, 6, 7 }, { 7, 6, 5, 4 },
      { 4, 5, 1, 0 }, { 5, 6, 2, 1 }, { 7, 4, 0, 3 } };
    float vertices[8][3] = { // Cube vertex positions
      { -side, -side, side }, { -side, -side, -side }, { -side,
          side, -side },
      { -side, side, side }, { side, -side, side }, { side, -side,
          -side },
      { side, side, -side }, { side, side, side } };

    for (int i=0; i<6; i++) { //for every face
      for (int j=0; j<4; j++) { //for every vertex in the face

        float r0 = rotationMatrix[matrixIndex];
        float r1 = rotationMatrix[matrixIndex+1];
        float r2 = rotationMatrix[matrixIndex+2];
        float r3 = rotationMatrix[matrixIndex+3];
        float r4 = rotationMatrix[matrixIndex+4];
        float r5 = rotationMatrix[matrixIndex+5];
        float r6 = rotationMatrix[matrixIndex+6];
        float r7 = rotationMatrix[matrixIndex+7];
        float r8 = rotationMatrix[matrixIndex+8];

        float v0 = vertices[faceIndices[i][j]][0];
        float v1 = vertices[faceIndices[i][j]][1];
        float v2 = vertices[faceIndices[i][j]][2];

        bodyVBO[bodyVBOIndex] = r0*v0 + r1*v1 + r2*v2 + position[
            bodyIndex].x;
        bodyVBO[bodyVBOIndex+1] = r3*v0 + r4*v1 + r5*v2 + position[
            bodyIndex].y;
        bodyVBO[bodyVBOIndex+2] = r6*v0 + r7*v1 + r8*v2 + position[
            bodyIndex].z;

        float n0 = normals[i][0];
        float n1 = normals[i][1];
        float n2 = normals[i][2];

        bodyVBO[bodyVBOIndex+bodyVBOStride] = r0*n0 + r1*n1 + r2*n2
            ;
        bodyVBO[bodyVBOIndex+bodyVBOStride+1] = r3*n0 + r4*n1 + r5*
            n2;
        bodyVBO[bodyVBOIndex+bodyVBOStride+2] = r6*n0 + r7*n1 + r8*
            n2;

        bodyVBOIndex += 3;
      }
    }
  }
}

__kernel
void updateParticles(
  __global float3* bodyPosition,
  __global float3* bodyVelocity,
  __global float* rotationMatrix,
  __global float3* angularVelocity,
  __global float* particleVBO,
  __global float3* particlePosition,
  __global float3* particleVelocity,
  float particleRadius
) {
  unsigned int particleIndex = get_global_id(0);
  unsigned int bodyIndex = particleIndex / 27;
  unsigned int matrixIndex = bodyIndex * 9;

  float3 originalRelativePos;
  //Calculate original relative position
  {
    int relativeIndex = particleIndex % 27;

    int xIndex = relativeIndex / 9;
    relativeIndex -= xIndex * 9;

    int yIndex = relativeIndex / 3;
    relativeIndex -= yIndex * 3;

    int zIndex = relativeIndex;

    float space = 2.0f*particleRadius;

    xIndex--;
    yIndex--;
    zIndex--;

    originalRelativePos.x = xIndex*space;
    originalRelativePos.y = yIndex*space;
    originalRelativePos.z = zIndex*space;
  }

  //Update particle position
  {
```

```
  particlePosition[particleIndex].x =
    originalRelativePos.x*rotationMatrix[matrixIndex] +
    originalRelativePos.y*rotationMatrix[matrixIndex+1] +
    originalRelativePos.z*rotationMatrix[matrixIndex+2];

  particlePosition[particleIndex].y =
    originalRelativePos.x*rotationMatrix[matrixIndex+3] +
    originalRelativePos.y*rotationMatrix[matrixIndex+4] +
    originalRelativePos.z*rotationMatrix[matrixIndex+5];

  particlePosition[particleIndex].z =
    originalRelativePos.x*rotationMatrix[matrixIndex+6] +
    originalRelativePos.y*rotationMatrix[matrixIndex+7] +
    originalRelativePos.z*rotationMatrix[matrixIndex+8];

  particlePosition[particleIndex].x += bodyPosition[bodyIndex].x;
  particlePosition[particleIndex].y += bodyPosition[bodyIndex].y;
  particlePosition[particleIndex].z += bodyPosition[bodyIndex].z;

}

//Update particle velocity
float scalar = sqrt(
    angularVelocity[bodyIndex].x*angularVelocity[bodyIndex].x +
    angularVelocity[bodyIndex].y*angularVelocity[bodyIndex].y +
    angularVelocity[bodyIndex].z*angularVelocity[bodyIndex].z );

scalar *= scalar;

particleVelocity[particleIndex].x = bodyVelocity[bodyIndex].x;
particleVelocity[particleIndex].y = bodyVelocity[bodyIndex].y;
particleVelocity[particleIndex].z = bodyVelocity[bodyIndex].z;

if (scalar > 0.0f) {
  float3 relativePosition = {
    particlePosition[particleIndex].x-bodyPosition[bodyIndex].x,
    particlePosition[particleIndex].y-bodyPosition[bodyIndex].y,
    particlePosition[particleIndex].z-bodyPosition[bodyIndex].z
        };

  float scalar2 = (
      angularVelocity[bodyIndex].x*relativePosition.x +
      angularVelocity[bodyIndex].y*relativePosition.y +
      angularVelocity[bodyIndex].z*relativePosition.z
      ) / scalar;

  float3 term = {
    relativePosition.x - angularVelocity[bodyIndex].x*scalar2,
    relativePosition.y - angularVelocity[bodyIndex].y*scalar2,
    relativePosition.z - angularVelocity[bodyIndex].z*scalar2 };

  particleVelocity[particleIndex].x += (angularVelocity[bodyIndex
      ].y*term.z - angularVelocity[bodyIndex].z*term.y);
  particleVelocity[particleIndex].y += (angularVelocity[bodyIndex
      ].z*term.x - angularVelocity[bodyIndex].x*term.z);
  particleVelocity[particleIndex].z += (angularVelocity[bodyIndex
      ].x*term.y - angularVelocity[bodyIndex].y*term.x);
}

//Update particle VBO
unsigned int particleVBOIndex = particleIndex * 3;
```

```
  particleVBO[particleVBOIndex] = particlePosition[particleIndex].x
      ;
  particleVBO[particleVBOIndex+1] = particlePosition[particleIndex
      ].y;
  particleVBO[particleVBOIndex+2] = particlePosition[particleIndex
      ].z;
}
```