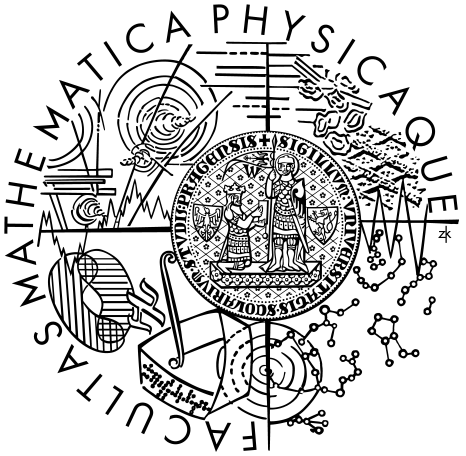
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Szabolcs Cséfalvay

Dynamic simulation of rigid bodies using programmable GPUs

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Petr Kmoch

Study programme: Informatics

Specialization: Software Systems

Prague 2011

I would like to thank my supervisor Petr Kmoch for accepting this thesis and for his helpful suggestions, and Chris Hecker for his excellent article on physics simulation. I would also like to thank my parents for supporting my years at college.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature

Název práce: Dynamická simulace tuhých těles na programovatelných GPU

Autor: Szabolcs Cséfalvay

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Petr Kmoch

Abstrakt: Cílem této práce je vytvořit program simulující dynamiku tuhých těles a jejich soustav pomocí GPGPU se zaměřením na rychlost a stabilitu. Výsledkem je fyzikální engine využivající architekturu CUDA. Celý engine běží na GPU, obsahuje detekci a zpracování kolizí a také různé síly jako tření, gravitace, kontaktní síla apod. Podporuje objekty typu koule, tyč, pružina, kvádr a rovina. Také umožňuje tato primitiva kombinovat do složitějších celků.

Klícová slova: CUDA, GPGPU, simulace fyziky

Title: Dynamic simulation of rigid bodies using programmable GPUs

Author: Szabolcs Cséfalvay

Department: Department of Software and Computer Science Education

Supervisor: Petr Kmoch

Abstract: The goal of this work is to create a program which simulates the dynamics of rigid bodies and their systems using GPGPU with an emphasis on speed and stability. The result is a physics engine that uses the CUDA architecture. It runs entirely on the GPU, handles collision detection, collision response and different forces like friction, gravity, contact forces, etc. It supports spheres, rods (which are similar to cylinders), springs, boxes and planes. It's also possible to construct compound objects by connecting basic primitives.

Keywords: CUDA, GPGPU, physics simulation, physics engine, game physics

Contents

Introduction

Today we can see a revolution happen in computing. Up until now most computationally intensive tasks were run on CPUs, but with the advent of modern graphics cards, which have a significantly larger raw processing power, the emphasis is gradually shifted towards the GPU.

For many years the GPU was only used to accelerate some parts of the graphics rendering pipeline. As the hardware became more sophisticated, it became possible to execute more and more complex tasks besides computer graphics computations. The current generation of GPUs have the same complexity as CPUs (in the sense that they are capable of executing while-loops and conditional statements), thus they are a good target to use for intensive computations.

A GPU's structure differs significantly from the structure of a CPU. They're fit to execute simple programs in a parallel fashion, instead of executing a single program with a serial structure as fast as possible. Interestingly, algorithms that require large computational resources are mostly fit to run a lot faster on a GPU. Besides rasterization these algorithms include signal processing, sorting, matrix operations, fast Fourier transformation, physics simulations, video encoding/decoding, etc.

An interesting area that can benefit from GPU computing is physics simulation. A piece of software performing physics calculations is called a phsics engine. It can have many uses including scientific calculations and interactive simulations, it can be used to create special effects for movies and has other uses in computer graphics. A physics engine is also an important part of most games, and computers that are sufficiently powerful to run these games neccessarily contain graphics cards that could also be used for physics calculations.

This work focuses on creating a real-time physics engine that can handle a large number of objects. It runs entirely on the GPU, and all data structures and algorithms were tailored to exploit the GPU's capabilities.

The work consists of 2 main chapters. The first one gives an overview of the architecture of the GPU and discusses the issues connected to designing algorithms for the GPU. The second one explores the specific problems of physics engine design more closely and describes the algorithms used to solve them.

1. About GPGPU computing

GPGPU stands for general-purpose computing on graphics processing units. It's a technique of using the GPU for computing tasks that are generally handled by the CPU. There are many frameworks currently available for GPGPU computing. One of them is CUDA, which was developed by NVidia and is a mature system with great support. On the other hand it only supports Nvidia hardware.

OpenCL is another such framework, which supports a wide range of hardware including newer Nvidia and AMD GPUs, IBM's Cell processor, some of Intel's integrated graphics processors and some newer CPUs.

Direct Compute is an API developed by Microsoft that supports GPGPU on Windows Vista and Windows 7. It's part of the DirectX collection of APIs.

I mainly considered OpenCL and CUDA for this project. The interface that CUDA provides is less abstract that that of OpenCL, and the Nvidia forums already have lots of CUDA related questions answered. Furthermore, at the time of writing the performance of CUDA was slightly better than the performance of OpenCL, and since porting CUDA code to OpenCL can be done with minimal modifications, I chose CUDA for this project for it's ease of use, performance and great support.

1.1 CUDA overview

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing architecture that exposes the capabilities of the graphics card through a language very similar to C. To understand how it works we must first take a look at how a GPU's structure differs from a CPU's.

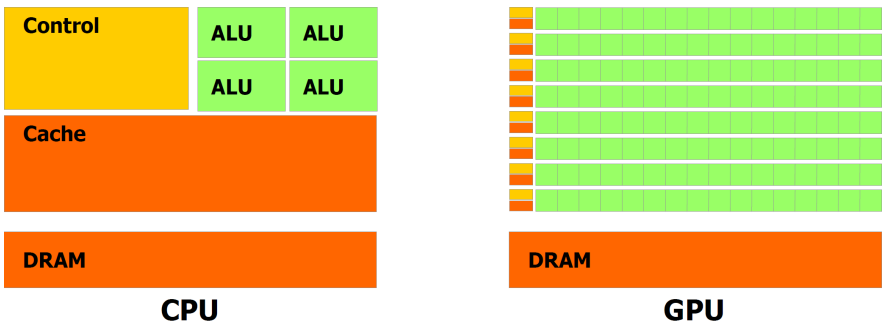


Figure 1.1: CPU vs. GPU (taken from[10, p. 162])

As the figure shows, in the GPU a lot more transistors are devoted to data processing than storage and flow control. Unlike CPUs, GPUs have a parallel architecture that emphasizes executing many concurrent threads slowly, instead of running a small number of threads quickly. This means that to solve a problem

fast using GPUs, it must be divided into parts where the same program can be executed on many data elements in parallel.

A GPU consists of an array (usually about 4-16) of *Streaming Multiprocessors* (*SM*s). Each SM is capable of running hundreds of threads concurrently. To do that it employs an architecture called *SIMT* (*Single-Instruction, Multiple-Thread*). The *CUDA C Programming Guide* explains it very well:

"The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. [...] When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a *warp scheduler* for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. [...] A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths."[10, p. 97]

The CUDA programming model is built around this architecture. "At its core are three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization – that are simply exposed to the programmer as a minimal set of language extensions."[10, p. 4] These are all very important from the aspect of algorithm design for CUDA:

Barrier synchronization The CUDA architecture offers intrinsic functions which enable threads to coordinate their memory accesses. This makes it possible to access memory in a thread-safe fashion for threads inside the same thread block. Such synchronization is not supported by default between threads from different thread blocks, but some newer devices offer atomic functions which enable thread safety between threads in different thread blocks.

Shared memories CUDA differentiates the memory levels that are accessible for a program running on the device. The contents of host memory (what is usually called RAM) can only be accessed by explicitly copying it to device memory (the so called video RAM) first. Global (device) memory offers a large area for storage (usually around 1 GB) and can be accessed by all threads, but the latency is quite high (400-800 clock cycles [10, p. 91]). Each multiprocessor also has on-chip memory where data can be stored for faster access. This cache is relatively small, 16 KB for devices of compute capability 1.x, and 48 KB for devices of compute capability 2.x. This can be accessed fast (8 reads / cycle), but can only be shared by threads inside the same thread block.

Hierarchy of thread groups Threads in a single group (called a *thread block*) have the ability to access memory in a coalesced fashion. The requirements for coalesced access can be strict, especially with earlier card models.

CUDA capable devices can be subdivided into groups according to their capabilities. These are called *compute capabilities*. The compute capability of a device is defined by a major and a minor revision number. Devices based on the Fermi architecture have a major revision number of 2. Prior devices have a major revision number of 1. The minor revision number differentiates devices further according to their capabilities. Figure 1.2 shows the details.

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Feature Support
(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Section B.11)</td><td rowspan="2">No</td><td colspan="4" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (Section B.11.1.3)</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Section B.11)</td><td colspan="2" rowspan="4">No</td><td colspan="3" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (Section B.11.1.3)</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Section B.11)</td></tr>
<tr><td>Warp vote functions (Section B.12)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Section B.11)</td><td colspan="4" rowspan="8">No</td><td rowspan="8">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (Section B.11.1.1)</td></tr>
<tr><td>__ballot() (Section B.12)</td></tr>
<tr><td>__threadfence_system() (Section B.5)</td></tr>
<tr><td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td></tr>
<tr><td>Surface functions (Section B.9)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
</table>

Figure 1.2: Compute capabilities (taken from[10, p. 158])

As the table shows, devices with compute capability 1.0 don't support atomic operations, which means that threads in different thread blocks can not access memory in a thread safe manner. For the sake of backward compatibility (and partially out of necessity, because my card only supported compute capability 1.1) I chose to conform to compute capability 1.0.

2. The physics engine

2.1 Design goals

When developing a physics engine one must decide: What types of scenes should the engine be optimized for? Will there be a lot of static objects and just a few dynamic objects? Or will the engine be used for things like particle simulation, where any object can collide with others? Will the objects move around a lot, or can we exploit temporal coherence to speed up collision detection? Will there be lots of collisions happening each frame, or will collisions happen only occasionally? Will objects be approximately the same size, or will they differ significantly? These questions all affect the choice of the algorithms.

2.1.1 Collision Detection

A traditional approach to collision detection using the CPU is to use some kind of space partitioning technique, like *octrees* or *kd-trees*. This means building a tree data structure by successively subdividing space by (usually axis-aligned) planes. The downside to this method is that the tree must be rebuilt for each frame (because objects' positions could change between frames), and BSP tree building is not fit well for the GPU because of it's recursive nature. A more fitting structure for GPUs are grids, and they have been used for parallelized collision detection already [4]. Unfortunately grids don't work well for irregularly placed objects which also vary in size.

Another common approach for collision detection is the *sweep and prune* algorithm. It is used by major physics engines like Bullet [2] (which is open source and has an OpenCL branch) and PhysX [9] (which is a proprietary engine developed by Nvidia). The good thing about this algorithm is that it exploits temporal coherence. Temporal coherence means the fact that the arrangement of objects does not change much between frames. The downside of this algorithm is that it doesn't work that well for a large number of objects, and also does not lend itself for parallelization that easily.

We would like to create an algorithm that's fit for the CUDA architecture, and general enough to support different object types. So let's approach the problem from a different aspect: Which algorithm would be the easiest to parallelize? Checking every object against every other object of course! Any decent programmer would frown at the idea of using an $O(N^2)$ algorithm in this case. Surely anything must be better that *that*! But as a matter of fact this approach can be excellently parallelized. In fact a very similar algorithm is used for N-body simulation in one of the CUDA SDK code samples [11], where it is used to calculate the force of gravity between all object pairs in a system. There it is shown that an algorithm like this can get close to the theoretical peak performance of the GPU. To briefly summarize the algorithm: The set of all colliding pairs can be thought of as a matrix. To check for each possible collision, each element of the matrix must be evaluated. This evaluation is sped up by subdividing the matrix

to (square) sub-matrices. In the end each thread evaluates one line of the matrix, which means that the nth thread detects all the collisions with the nth object. This algorithm can also be modified to require less than $O(N^2)$ operations to check for all possible collisions (for details on this see section 2.5.1).

2.1.2 Collision pair representation

The output of the collision detection phase are object pairs that are close enough to be candidates for collision. There are multiple ways to represent these pairs. One of the possibilities is to create a single list of pairs where every thread can add the pairs it detects. Since synchronizing the way threads write to this list would be unoptimal, each thread must write to the list independently, and thus have a fixed number of reserved places to write to. This means that a large part of this list would be left empty:

Figure 2.1: List of collisions - single array

After this is done, either the whole array must be compacted (which takes a long time because of the large number of necessary memory copies involved), or it must be passed as-is to the collision handling phase. This would mean that a lot of unnecessary threads would be launched because of the holes in the list, and this again would slow down the algorithm. Worst of all, resolving collisions would take a long time even if a small number of collisions would be detected in that frame.

A faster approach (as practice shows about 5-10 times faster) is to create multiple lists, where the nth element on each list belongs to the nth object. The first list contains the first colliding pair, the second list contains the second, etc. Since the threads in the collision handling phase will try to read the first colliding pair in parallel, these reads can be coalesced into a singe memory transaction within a warp, which leads to a significant performance boost compared to random access. Also, if all the threads inside a warp determine that no more collision pairs need to be read, no more arrays will be read. Therefore, if there are no collisions for any of the threads in a warp, only a single memory read operation is needed to determine it. This leads to a fast best-case scenario when there are no collisions during a single frame.

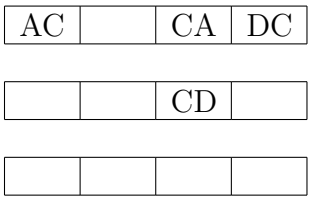


Figure 2.2: List of collisions - multiple arrays

In the situation above there would be 3 memory accesses for the whole warp (for a warp size of 4) to determine all the collisions that happened with objects inside that warp (i.e. objects 1-4). This way the parameters of the objects handled by a warp are also read in a single coalesced memory read.

2.1.3 Representation of object parameters

Since we want out engine to handle a large number of objects, we can not expect to fit all our data into the cache memory of the GPU. This means we must store object data in device memory, which has a significantly greater latency. On the other hand device memory can be read in a coalesced manner, which must be exploited as much as possible.

Normally one would represent object parameters using *struct*s, and an array of these structs would be used to represent multiple objects. This would mean a memory arrangement like this for example:

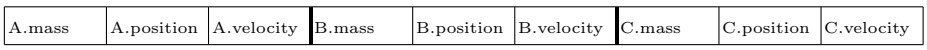


Figure 2.3: Object parameter representation using structs

With this memory arrangement it would not be possible to read object data in a coalesced fashion, because it does not fulfill the requirements for coalesced access. According to the *CUDA C Programming guide*[10, p. 162] for devices of Compute Capability 1.0 :

"To coalesce, the memory request for a half-warp must satisfy the following conditions:

1. The size of the words accessed by the threads must be 4, 8, or 16 bytes;
2. If this size is:
   (a) 4, all 16 words must lie in the same 64-byte segment,
   (b) 8, all 16 words must lie in the same 128-byte segment,
   (c) 16, the first 8 words must lie in the same 128-byte segment and the last 8 words in the following 128-byte segment;

3. Threads must access the words in sequence: The kth thread in the half-warp must access the kth word.

If the half-warp meets these requirements, a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the words accessed by the threads is 4, 8, or 16, respectively. Coalescing is achieved even if the warp is divergent, i.e. there are some inactive threads that do not actually access memory.

If the half-warp does not meet these requirements, 16 separate 32-byte memory transactions are issued."

Therefore the data fields must be separated into individual arrays, where a single member of the array is either 4, 8 or 16 bytes long. In practice this simply means that a separate array is needed for each parameter, like this:

A.mass	B.mass	C.mass	D.mass

A.position	B.position	C.position	D.position

A.velocity	B.velocity	C.velocity	D.velocity

Figure 2.4: Object parameter representation using separate arrays

It is also necessary to use 4, 8 or 16 byte fields (these are already defined, i.e. `float`, `float2`, `float4`). And since according to the requirements above the kth thread must access the kth word, for the best performance thread k should calculate everything related to the kth object (the alternative would be to launch a thread for each interaction). This is also good for a second reason: If the parameters of object k are only changed by thread k, there can be no access conflict, therefore this approach is thread-safe. This means that a single *kernel*[10, p. 7] can evaluate all forces that affect an object, because there is no need to synchronize the threads. This is faster, because the parameters of the kth object (mass, position, velocity, etc., there is quite a lot) must only be read once, and can be reused for each phase. Alternatively, a new thread would need to be launched for each phase.

On the downside, each interaction must be evaluated multiple times, depending on the number of objects affected by the interaction. To do that, each object must have references to the object(s) it is part of. Since part of a rod's definition are the 2 enclosing spheres at it's ends, collisions of rods cannot be separately evaluated from collisions of spheres, because a rod's collision also affects the spheres at it's ends. Therefore these interactions must always be evaluated from the perspective of the spheres. Specifically: The forces acting directly on sphere A and the forces acting on a rod which A is part of must both be calculated to determine the total force acting on A. If separate threads would calculate these forces (say threads α and β), then their effect could not be combined, because they might be

located in different thread blocks, and Compute Capability 1.0 does not support atomic operations for variables residing in global memory. By using a function supported by Compute Capability 1.1 though it would possible to combine these forces (by using the function `atomicAdd()` [10, p. 119]) This would eliminate the need to evaluate rod interactions through spheres by making it possible to launch separate threads for rod interactions. Also, each interaction would need to be evaluated by one thread only (only once). This in turn would speed up force calculations.

Since the approach when the kth thread evaluates (only) the kth sphere only is inherently thread-safe, I used this approach for most algorithms in this engine, namely the *collision detector* and the *integrator/force calculator*. The *collision handler* is an exception. Since collisions are relatively rare and phase synchronization is not an issue (since there is only one phase), I chose a model where each interaction is evaluated only once. In case of multiple collisions per object per frame some collisions might be disregarded, but with collisions it is a rare scenario, unlike with force calculations. There is also a method to reduce the prevalence of problems arising from simultaneous access, which is described at the section about collision handling (2.6).

2.2 Basic structure

The engine consists of different parts (kernels) which are called in sequence to compute a single step of physics simulation. The first phase updates the AABB boundaries of all the objects in the scene. It launches a thread for each sphere, rod and cuboid in the scene and calculates their minimal and maximal extent along the axes. The results of this phase are saved as a `float2` for each axis (representing the minimum and maximum along that axis). The parallelization of this phase is very straightforward and its runtime is very short (about 5-10% of total runtime).

The second phase is called broad-phase collision detection, which detects overlaps between the AABBs of the objects and writes the results to a list for each object. Fore details on the implementation see the corresponding section (2.5).

The next phase resolves collisions between the objects. It calculates the collision impulse between objects and directly changes their velocities accordingly. For more information see 2.6.

The last phase calculates the forces affecting each object, and changes the velocity and position of each object accordingly. Furthermore, it uses 4th order Runge-Kutta integration to calculate the force more accurately. For more information see section 2.7.

2.3 Engine features

The basic purpose of this engine is to support the interaction of a large number of objects and be modular.

The basic physical primitive is the *sphere*. Spheres can collide, stay in contact, rotate if affected by torque, etc.

Another primitive is the *rod*, which is defined as an object connecting 2 spheres. It's similar to a cylinder, except that there is no enclosing circle – this role is fulfilled by the spheres. By default their length is not defined and can change freely. It's also possible to emulate capsules using rods by specifying their length.

Other physical primitives are the *box* and the *polygon*. These are only static objects, meaning the can't move. Springs are also supported, which can be placed between 2 spheres. It's possible to connect multiple springs and rods to a sphere, which enables the simple emulation of soft bodies. There is a hard-coded limit to the number of objects of each type that can be connected to a single object, but this limit can be changed by changing a `#DEFINE` parameter in the engine code and recompiling. Increasing this limit increases the memory requirements of the engine.

The basic interactions between objects include collision, contact (repulsion and damping) and friction.

There's also gravity, which has a fixed direction (downward), but it's strength can be changed. There are also other constants defined which can be changed (at runtime).

The engine uses fourth order Runge-Kutta numeric integration to increase stability.

2.4 Physical primitives

The engine supports multiple types of physical primitives. These can be either static or dynamic. The movement of static objects is not calculated, but they can still interact with moving objects.

The parameters of the objects are stored in separate arrays, where the nth member of each array belongs to object n (see (2.1.3 page 8)).

2.4.1 Spheres

The most basic object type is the *sphere*. It is characterized by the following parameters:

radius Each sphere has a radius, which is represented by a single `float`.

position The 3-dimensional position of the sphere. It's represented by a `float4` type, where the `x`, `y` and `z` fields represent its location, and the `w` field stores a flag which determines whether it's a static or a moving object (a positive `w` value means it's static).

velocity The 3-dimensional velocity vector of the sphere, also represented by a `float4`. The first 3 fields represent the vector itself, and the `w` field is unused (it's still needed for padding).

orientation The orientation of a sphere is characterized by a rotation matrix. This matrix is stored as 3 orthonormal vectors: 3 `float4`s with an unused w field. The 3 `w` fields could be used to store a 3-dimensional vector (like the angular momentum), but then 3 separate reads would be required to access that vector. Although there exist more compact representations of rotations (like quaternions), a matrix representation was chosen for its ease of use (it can be directly transcribed to an OpenGL rotation for example).

angular momentum The angular momentum of the sphere, represented by a 3 dimensional vector. The vector's direction determines the axis of rotation, and its length determines the magnitude of the angular momentum. Angular momentum can be visualized using the *right-hand rule* (if the extended thumb of the right hand points in the direction of the angular momentum vector, the bent fingers indicate the direction of rotation.).

rod index array For reasons described at the end of the previous chapter a sphere must contain a reference to the rods it is part of. These references are integers which refer to the position of the rod in the array of rods. These indexes are stored in one of n arrays, where the index of the kth rod the ith sphere is part of is stored in the ith member of the kth array, and n is the maximum number of rods a sphere can be part of. The reason for this representation is to support coalesced access, as described in the previous chapter (2.1.3 page 9).

spheres collision index array This contains the indexes of spheres this sphere collides with, in a structure similar to the previous one.

rod collision index array This contains the indexes of rods that the sphere collides with.

box collision index array This contains the indexes of boxes the sphere collides with.

There are additional variables used in the internal representation of spheres which are just auxiliary variables for the numerical integrator. The values of these do not carry over to the next frame.

2.4.2 Rods

The next physics primitives that are handled by the engine are rods. The representation of rods is a bit unusual: They are defined by two spheres which

enclose the rods, and a radius. Therefore when I mention rods in this work I mean a cylinder without its caps. The reason for this representation is twofold:

First, cylinders are hard to handle inside a physics engine, because there are many special cases. They have sides, ends and rims which must be handled separately. Having cylinders inside a physics engine would mean supporting every feature related to them, like cylinder-cylinder collisions. This is a missing feature even in the ODE physics engine due to its difficulty (see the table at the top of the page at [13]). Therefore cylinders are usually enclosed by spheres instead of circles. The resulting object is called a *capsule*.

Second, using enclosing spheres makes it possible to use the same sphere as an enclosure for multiple rods. This creates the possibility for compound objects which behave as soft bodies if supported by springs. As a side note: The ability to connect components also opens up the possibility to use the engine (after extension) for virtual creature evolution [14] which requires large computational resources and thus would benefit from the performance CUDA provides.

A downside of this representation is that rods themselves don't have a mass of their own, they just act as a conveyor of forces and impulses to spheres.

All the parameters of a rod are stored in a single `float4` field. Since the references to spheres are integers, they are stored in the `x` and `y` fields as integers (without type conversion), and read inside kernels using the `__float_as_int()` type casting function. Rods have the following parameters:

enclosing sphere indexes The indexes of the 2 enclosing spheres of the rod, stored inside the `x` and `y` fields.

radius Each rod has a radius represented by a single `float`, stored inside the `z` field.

length Rods don't have a fixed length by default, their length is given by the distance of their enclosing spheres. If this parameter is set to positive though, it's considered to be a constraint on the distance of the 2 enclosing spheres, and is enforced. Therefore this parameter can be used to create capsules, which have a fixed length. This is stored inside the `w` field.

2.4.3 Springs

Springs are not independent primitives. Just like rods, they must also be defined between 2 spheres. These spheres act like point masses the spring force acts on by attracting or repelling them. Since it must be possible to quickly access all springs the sphere is attached to, each sphere has a list of springs attached to it. This has the advantage that 1 less dereferencing is required to read all the data for a spring force calculation. The downside is that it takes up more space (using a pointer/index: (2×`sizeof`(index) + 5×`sizeof`(float) = 28 bytes, saving it twice: 2×`sizeof(float4)` = 32 bytes) A spring's data is stored in a single `float4` structure, just like with rods. Springs are characterized by their

attachment points, their length, the spring constant (how "strong" the spring is) and the damping coefficient (how fast the "bouncing" of the spring stops).

attachment point The index of the other sphere this sphere is attached to by the spring, stored inside the `x` field as an `int` and recovered using the `__float_as_int()` type casting function.

spring constant The spring constant defines the elasticity of the spring through *Hooke's law*, stored as a `float` in the `y` field.

length The length of the spring at rest. The displacement from this length and the spring constant are used to calculate the *spring force*. It's stored as a `float` in the `z` field.

damping coefficient The damping coefficient is used to dampen the movement of a spring. The *spring damping force* is calculated from the damping coefficient and the approach velocity of the two ends of the spring. It's stored as a `float` in the `w` field.

2.4.4 Capsules

The definition of a capsule, as I already mentioned, is a cylinder enclosed by spheres. Capsules are internally handled as a special case. The difference between a capsule and a sphere-enclosed rod is the following: During collisions when an impulse acts on one of the spheres, in case of capsules, the impulse is propagated along the axis to the other endpoint (the impulse is projected onto the axis first). This way when a capsule falls vertically on its end it bounces up, while normally it would just stop (after collision the sum of the momenta of the 2 spheres would be close to 0). The other difference is that when all the force calculations have ended, several constraints are enforced on the spheres at the endpoints:

- Their orientation is set to the orientation of the axis
- Their angular momentum is set to the average of their angular momenta projected onto the capsule axis. The thus "absorbed" angular momentum is transferred as momentum to the opposing sphere (because of conservation of angular momentum).
- Their position is set to be *length* while preserving the center of mass
- Their approach velocity is set to be 0 while preserving total angular momentum.

The result of this approach is that forces are propagated along the axis, even though the force calculations themselves don't consider the constraints on the capsules.

2.4.5 Boxes

Boxes are handled as static objects, which means that they don't move, and they don't "absorb" forces and impulses but reflect them at the objects that get into contact with them. Boxes are represented using 4 vectors. The first vector is the location of the corner of the box, and the other vectors are the 3 orthogonal edges that are connected to that corner (vertex). This representation, although redundant (a box could be represented using only 9 values instead of 12), is straightforward and makes calculations easy without having to convert from a different representation.

corner The 3-dimensional position of a corner vertex of the box, saved as a `float4`.

edges The 3 edge vectors connected to the corner, saved as `float4`s. They must be orthogonal.

This representation could also be used for cuboids.

2.4.6 Polygon

Polygons can be used simply as walls inside the simulation. More specifically they are parallelograms. Their edge is not physically defined, meaning that collision with the edge is not possible. They are defined using 3 vectors: The first is a vertex of the polygon. The second 2 define the 2 edges of the polygon connected to this vertex, and the area between them defines the surface of the Polygon object:

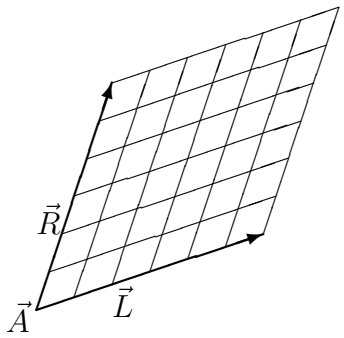


Figure 2.5: Representation of a polygon

The parameters of a polygon are the following:

corner The 3-dimensional position of the corner vertex of the polygon (shown as $\vec{A}$ on the figure), saved as a `float4`.

edges The 2 vectors defining the polygon (shown as $\vec{L}$ and $\vec{R}$ on the figure), saved as `float4`.

The points of the polygon can be defined parametrically as

$$\vec{A} + l\vec{L} + r\vec{R}; \quad l, r \in (0, 1)$$

2.5 Collision detection

In a physics engine collisions must be detected before they can be handled. This is done by the collision detector. There are 2 main types of collision detection algorithms: *A priori* and *a posteriori*.

In the *a priori* case collisions are detected *before* they happen, thus the physical bodies never actually interpenetrate. This approach is more difficult to solve because it must also take into account the time variable. The trajectories of objects must be calculated precisely. Sometimes this problem does not have a closed form solution, and a numerical root finder is necessary.

In the *a posteriori* case collisions are detected by detecting the interpenetration of objects, that is *after* the collision has happened. This is easier, faster and more easily parallelizable that the a *a priori* method, and in this engine this approach is used. A downside of this algorithm is that it's sometimes necessary to correct the interpenetration of the objects. We do this by adjusting the positions of the objects in such a way that

1. The objects' distance becomes 0
2. The center of mass of the system of objects does not change

Since the scene changes every frame, collision detection must be run each frame too. Therefore the performance of the collision detector is critical to the performance of the engine.

The collision detection algorithm used in this engine was designed for the CUDA architecture. As input it gets the AABB (Axis-Aligned Bounding Box) of each object, and returns a list of overlapping AABB pairs (this is usually called broad-phase collision detection). More specifically, for each object a list of objects is returned whose AABB overlaps the object's AABB. This list is made to be accessible through coalesced memory operations.

2.5.1 The algorithm

The collision detection algorithm works by comparing the AABB of an object against the AABB of every other object. The algorithm could be summarized like this:

1. Read the next `block_size` AABBs to shared memory. The nth thread in the block reads the nth AABB in the sequence, thus this memory access is coalesced.

2. For each thread a `for` loop compares the AABB of the object with the next AABB in shared memory, and adds a new object to that object's collision list if an overlap is found. Since each thread reads the same AABB from shared memory at the same time, this is done in a single memory broadcast operation.

3. If there are still unchecked AABBs, repeat the previous steps.

A description of the terms used:

`block_size` The number of blocks in a thread, specified at kernel launch.

AABB An AABB is represented by a minimum and maximum extent along the 3 axes, thus needs 6 floats to store. These are organized into 3 `float2` fields.

Step 2 basically evaluates a square sub-matrix in the matrix of all possible AABB pairs. To put it differently: Let's say that `block_size` is 32. If the scene contains N objects, then the collision detection algorithm launches N threads (actually a little more because of padding). The nth thread stores the AABB of the nth object in a local variable. There are 32 threads in a thread block. Let's say they are numbered 33 to 64. At the beginning each thread in the block reads the first 32 AABBs in parallel (thread 33 reads the 1st, thread 34 the 2nd etc.). Next, each thread compares it's locally stored AABB to the 1st AABB in shared memory, then the second, etc. using a `for` loop, and in case of an overlap puts the overlapping object's index in a list. When all the object have been looped through, a new set of 32 objects is read into shared memory and the steps are repeated.

This way we can effectively check each AABB pair for collision without the need for any space-partitioning data structure.

This far the basics of the algorithm are the same as the one used for N-body gravity simulation described in [11].

Although not yet implemented, it's possible to speed up this algorithm.

Since the list of objects does not need to be in any particular order, we can rearrange it to speed up the algorithm. A good way to do that would be to sort the entries according to one of the coordinates, then determine for each block the interval along that axis the objects in that block occupy. This way by comparing the block interval we could completely skip over some blocks.

This would mean that we have to calculate the minimum and maximum coordinate values inside blocks each frame. This could be incorporated into the AABB update phase, thus it would minimally impact performance.

The other necessity would be to sort the entries and thus narrow the interval each block occupies. Since objects' parameters are stored in multiple arrays, this would require swapping the entries in all these arrays, which can be slow. Also, if objects refer to each other (like springs for example), these references must also be changed. Fortunately we do not need to sort the entries completely each frame, it's sufficient if we just sort them partially. We also don't need to do this each frame, and the entries can be presorted before the simulation begins.

This approach would practically mean that instead of evaluating the whole matrix, we evaluate only the parts near the diagonal. This way the originally $O(N^2)$ algorithm becomes an $O(N)$ algorithm for objects that are approximately the same size.

2.6 Collision response

After we determined which objects could overlap, we must model their interaction. In this engine there are 2 different types of interactions for overlapping objects: Collisions (modelled as impulses) and contacts (modelled as forces).

Collisions are instantaneous events. They can be thought of as a large force that affects the objects for a very short time. When determining the effects of a collision we must calculate the impulse that the objects affect each other with. An impulse is a change in momentum, and is used to directly change an object's velocity.

When we change the velocity of an object there is one minor thing we must pay attention to. To quote the CUDA C Programming Guide[10, p. 164]:

"If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined."

This means that if an object collides with two other objects in the same frame, it might be possible that one of the changes will not be effective. Example: Object 1 collides simultaneously with objects 2 and 3. If both collisions are handled by threads in the same half-warp, both threads will try to change the velocity of object 1 at the same time. According to the quote above this means that only 1 write is performed. To remedy this problem we can use the following method to guarantee that all the threads' changes are visible:

```
__device__ void threadSafeAddVec(
  float4 &base,
  float3 add,
  unsigned int tid
){
  short guard = 0;
  base.w = -1;
  while(base.w != tid) {
```

```
        if(++guard > BLOCK_SIZE) break;
        base = make_float4(make_float3(base) + add, tid);
        __syncthreads();
    }
    __syncthreads();
    base.w = -1;
}
```

This function increments the vector `base` by the vector `add`. `tid` is the global thread index of the thread attempting the increment, and `BLOCK_SIZE` is the size of one block. The reason this works is that when we increment the vector (stored in a `float4` of which we only use the first 3 fields) we store the thread index of the thread in the `w` field, thus when a thread reads back this value it can determine whether its write was successful and give up trying.

The reason for the `guard` variable is to not let an infinite loop occur when threads from different thread blocks (residing on different multiprocessors) try to increment a vector. Although it is theoretically possible that after `BLOCK_SIZE` number of loops still not all threads managed to successfully perform a write, it can only happen in very badly defined scenes when lots of objects overlap. It this case the least of our concerns is to properly increment the velocity vector...

There is also the possibility that 2 threads residing in different warps try to increment a vector at the same time, in which case the above trick does not work. Realistically, the possibility that an object collides with 2 other objects during the same frame, and both collisions are evaluated at the same time in different half-warps is very slim, so the inconsistencies that arise from these situations are negligible. This is not the case for contacts. In the case of stacked objects the contact forces are calculated each frame, and it is quite likely that there would be situations where the increment operation would cause problems, and for multiple consecutive frames. That's why we use different approaches with collisions and forces.

In the following section each interaction between objects is described. The interaction between each type of object pair (spheres-spheres, spheres-rods, etc.) is sufficiently different to be handled separately. We could do this by launching as many threads as there are objects and let each thread handle an object, like we do during the force calculation phase. But since here we don't need to make sure that only a single thread accesses a global variable (for reasons described in section 2.1.3), we can create more threads that will do the calculations. So instead of calculating the area above the diagonal in the matrix of all object pairs by launching 1 thread per object (a total of $\#spheres + \#rods + \#boxes$ threads):

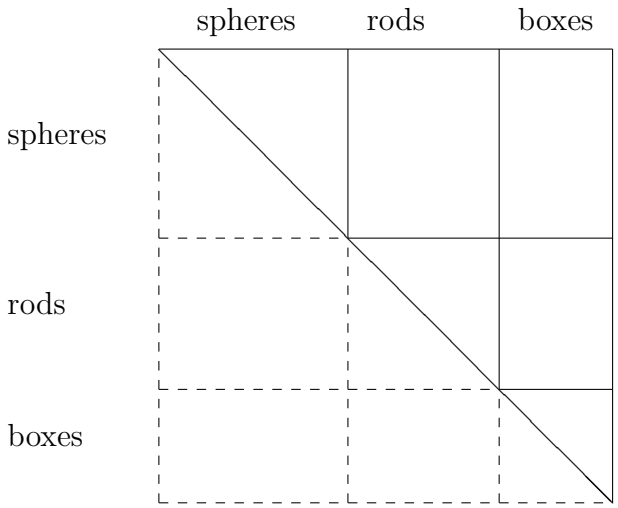


Figure 2.6: Matrix of collisions

We can evaluate each piece one-by one:

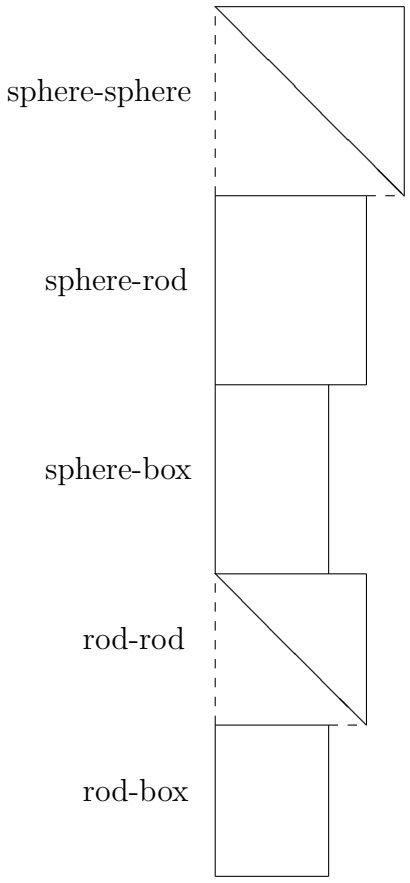


Figure 2.7: Disassembled matrix of collisions

This way we launch a total of $3\times\#spheres + 2\times\#rods$ threads. This distributes the load between multiprocessors leading to faster execution. When there is a

small number of objects in the scene, more multiprocessors are utilized this way.

2.6.1 Collision with static objects

The main characteristic of collisions with static objects is that considering only the colliding object, the momentum of the object is not conserved. Technically it should be transferred to the static object, but we don't consider static objects to have momentum.

There are 2 main types of collisions with static objects: Collisions with walls or polygons (defined for spheres) and collisions with boxes (defined for spheres and rods).

Sphere-polygon collision

When a sphere collides with a polygon, its momentum changes by an amount that is determined by its initial momentum, the normal of the polygon and the coefficient of restitution. The input parameters of the algorithm are:

$\vec{A}$	The position of the center of the sphere.
$\vec{v}$	The velocity vector the sphere.
m	The mass of the sphere.
R	The radius of the sphere.
e	Coefficient of restitution.
$\vec{P}$, $\vec{L}$ and $\vec{R}$	The corner and 2 edges defining the polygon.

First we check whether the sphere touches the polygon by solving the following equation:

$$\vec{P} + l\vec{L} + r\vec{R} + d\vec{D} = \vec{A}$$

for l, r and d, where $D = \frac{\vec{L}\times\vec{R}}{\|\vec{L}\times\vec{R}\|}$. Collision only happens when l and r are between 0 and 1, and $d < r$. The solution for this system is:

$$l = (\vec{A} - \vec{P}) \cdot \vec{L}$$

$$r = (\vec{A} - \vec{P}) \cdot \vec{R}$$

$$d = (\vec{A} - \vec{P}) \cdot \vec{D}$$

Next, we check whether the approach velocity v of the sphere is large enough for collision to occur by projecting it onto the $\vec{D}$ vector:

$$v = \vec{v} \cdot \vec{D}$$

If v is less than a predefined constant, we will not handle the interaction as a collision, but as a contact force in a later phase. If a collision occurs, we calculate the sphere's change of velocity like this:

$$\Delta\vec{v} = (1 + e)v\vec{D}$$

and we change the sphere's velocity accordingly.

After this is done we need to do one more thing: Make sure that the polygon and the sphere no longer overlap, otherwise the contact interaction would still occur in a later phase. To do this we simply change the position of the sphere by $\vec{D}(R-d)$.

Sphere - box collision

The input parameters of the algorithm are:

$\vec{A}$	The position of the center of the sphere.
$\vec{v}$	The velocity vector the sphere.
m	The mass of the sphere.
r	The radius of the sphere.
e	Coefficient of restitution.
$\vec{P}$, $\vec{I}$, $\vec{J}$ and $\vec{K}$	The position and the 3 edge vectors defining the box. $\vec{I}$, $\vec{J}$ and $\vec{K}$ are orthogonal.

Since boxes are static objects, we only need to calculate how the velocity of the sphere changes. To do that we need to calculate the point of collision, and from that the collision normal. To do that we first transform the position of the sphere's center into the coordinate system of the box by solving the following equation:

$$\vec{P} + a\vec{I} + b\vec{J} + c\vec{K} = \vec{A}$$

for a, b and c. The solution is:

$$a = \frac{\vec{I} \cdot \vec{T}}{\vec{I} \cdot \vec{I}}$$

$$b = \frac{\vec{J} \cdot \vec{T}}{\vec{J} \cdot \vec{J}}$$

$$c = \frac{\vec{K} \cdot \vec{T}}{\vec{K} \cdot \vec{K}}$$

where $\vec{T} = \vec{A} - \vec{P}$.

$\vec{I} \cdot \vec{I}$ could also be written as $\|\vec{I}\|^2$, etc.

There are 2 main situations we must handle: when the sphere center is inside the box and when it is outside. It is inside when a, b and c are between 0 and 1. Then we must determine which face it lies closest to, and set the normal vector to point toward that face. When it's outside the box we find the closest point inside the box to the sphere using the following algorithm (x, y and z are the coordinates of the closest point $\vec{X}$ in box coordinates):

if $a < 0$ **then**
  $x \leftarrow 0$
else if $a > 1$ **then**
  $x \leftarrow 1$
else
  $x \leftarrow a$
end if
if $b < 0$ **then**
  $y \leftarrow 0$
else if $b > 1$ **then**
  $y \leftarrow 1$
else
  $y \leftarrow b$
end if
if $c < 0$ **then**
  $z \leftarrow 0$
else if $c > 1$ **then**
  $z \leftarrow 1$
else
  $z \leftarrow c$
end if

Then we convert box coordinates to world coordinates:

$$\vec{X} = \vec{T} - (x\vec{I} + y\vec{J} + z\vec{K})$$

The normal vector of the collision points from $\vec{X}$ to $\vec{A}$. Using this data we can calculate the change in velocity the same way we did in the case of sphere-polygon collisions. We must also correct the position of the sphere to not overlap with the box after collision, doing this is also simple because the amount of overlap is easy to compute.

Rod-box collision

The input parameters of the algorithm are:

$\vec{S}_1$, $\vec{S}_2$	The positions of the enclosing spheres.
$\vec{v}_1$, $\vec{v}_2$	The velocities of the enclosing spheres.
m_1, m_2	The masses of the spheres.
r	The radius of the rod.
e	Coefficient of restitution.
$\vec{P}$, $\vec{I}$, $\vec{J}$ and $\vec{K}$	The position and the 3 edge vectors defining the box. $\vec{I}$, $\vec{J}$ and $\vec{K}$ are orthogonal.

Our goal here is to find the contact point on both the box and the rod. On the rod it must lie somewhere on the circumference (since we don't model the rod's ends like a cylinder's). For the box there are 2 situations to consider: When the

contact point is on one of the edges of the box, or when it is one of its vertexes. Let's first consider the first case, which is equivalent to finding the closest points on 2 line segments. Let's say the 2 line segments are given by the coordinates of their end points. Let's call these $\vec{A_1}$ and $\vec{A_2}$ for line segment A, and $\vec{B_1}$ and $\vec{B_2}$ for line segment B. Let's define the following vectors:

$$\vec{A} := \vec{A_2} - \vec{A_1}$$

$$\vec{B} := \vec{B_2} - \vec{B_1}$$

$$\vec{C} := \vec{B_1} - \vec{A_1}$$

We want to find vector $\vec{D}$ which connects the 2 lines and is perpendicular to both, so we need to solve the following system of equations:

$$a\vec{A} + \vec{D} = b\vec{B} + \vec{C}$$

$$\vec{A} \cdot \vec{D} = \vec{0}$$

$$\vec{B} \cdot \vec{D} = \vec{0}$$

for a, b and $\vec{D}$. The results are:

$$a = \frac{\vec{C} \cdot (\vec{B} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})} \tag{2.1}$$

$$b = \frac{\vec{C} \cdot (\vec{A} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})} \tag{2.2}$$

$$\vec{D} = \vec{C} + b\vec{B} - a\vec{A} \tag{2.3}$$

Let $\vec{A}$ be the vector defining the rod and $\vec{B}$ the edge of the box we want to test against. A box has 12 edges, that means we need to do 12 different test. The edges can be grouped into 3 groups of 4 parallel edges, where for each edge inside the group the vector $\vec{B}$ is the same, only $\vec{C}$ (the displacement) changes. Therefore for each group we can precompute a large part of the formula for a and b. We only compute $\vec{C}$ for each edge, and only calculate the dot product of $\vec{C}$ with the precomputed parts to get a and b. If both a and b lie between 0 and 1 (which means there is a potential collision) we calculate the vector $\vec{D}$. If its length is less than r, there is a collision, in which case we will use a and $\vec{D}$ later.

If we fail to find a collision in this phase, we must also check whether there is a collision with a vertex. To do that we must calculate a line-vertex distance between the rod and each vertex of the box. Let's define the end points of the line as $\vec{A_1}$ and $\vec{A_2}$, and $\vec{A}$ as $\vec{A_2} - \vec{A_1}$. We want to calculate the distance between this line and a point $\vec{P}$. Let's define the relative position of $\vec{A_1}$ and $\vec{P}$ as $\vec{C} := \vec{P} - \vec{A_1}$. To calculate the distance vector $\vec{D}$ we must solve the following system of equations:

$$a\vec{A} + \vec{D} = \vec{C}$$

$$\vec{A} \cdot \vec{C} = \vec{0}$$

for a and $\vec{D}$ (a specifies the position of the contact point on the rod). The results are:

$$a = \frac{\vec{A} \cdot \vec{C}}{\vec{A} \cdot \vec{A}} \tag{2.4}$$

$$\vec{D} = a\vec{A} - \vec{C} \tag{2.5}$$

We use this equation to calculate a for each vertex (we can precompute $\vec{A}/\vec{A} \cdot \vec{A}$ and only need to compute a dot product (which is compiled into a single FMAD instruction). If a lies between 0 and 1 we calculate $\vec{D}$. If its length is less than r, there is a collision with the vertex.

If we determined in the previous phase that there was a collision, we must calculate the change of velocities of the enclosing spheres of the rod. To do that we must calculate the effective mass of the rod at the point of collision:

$$M = \frac{m_1 m_2}{m_2 - 2am_2 + a^2(m_1 + m_2)} \tag{2.6}$$

We also need the velocity of the rod at the point of contact, which we calculate using linear interpolation:

$$\vec{v} = (1 - a)\vec{v}_1 + a\vec{v}_2 \tag{2.7}$$

Through projecting this vector onto the collision normal we can determine the collision impulse (we define the collision normal as $\vec{n} := \frac{\vec{D}}{\|\vec{D}\|}$):

$$\vec{I} = M(\vec{n} \cdot \vec{v})\vec{n}$$

Now we need to redistribute the impulse among the spheres, we again do that using linear interpolation. So the changes in velocity are:

$$\Delta\vec{v}_1 = -(1 + e)(1 - a)\frac{\vec{I}}{m_1} \tag{2.8}$$

$$\Delta\vec{v}_2 = -(1 + e)(a)\frac{\vec{I}}{m_2} \tag{2.9}$$

One more thing we must do before we're done is to adjust the coordinates of the enclosing spheres so that the surface of the rod just touches the box. We do this in a similar way as the previous case, but instead of distributing *momentum* (which is *mass* times *velocity*), we distribute *mass displacement* (which we define as *mass* times *displacement*). The total mass displacement we want to distribute is $\vec{D} = oM\vec{n}$, where o is the overlap we want to compensate. So the final changes of position are:

$$\Delta\vec{v}_1 = (1 - a)\vec{D}\frac{M}{m_1}$$

$$\Delta\vec{v}_2 = (a)\vec{D}\frac{M}{m_2}$$

2.6.2 Collisions between moving objects

When calculating collisions between moving objects, there are 2 laws we must consider: One is the *law of conservation of momentum*, which states that the momentum of a closed system does not change. A closed system can be the whole simulation, or just a pair of objects at the moment of collision. The second law is not a physical law since in nature only velocity can change an objects' positions, but in our case it is necessary to change the position of objects to correct their overlap. For this purpose we use the *law of conservation of center of mass*. We could state this law like this: The position of the center of mass of a closed system does not change inside its moving reference frame. (We could even create the analogy of a force in relation to this law: the displacement, which would be defined as $m\vec{p}$.)

Moving objects inside the simulation are spheres and rods, so we will calculate collisions between them.

Sphere-sphere collision

The input parameters of the algorithm are:

$\vec{S}_1$, $\vec{S}_2$	The positions of the colliding spheres.
$\vec{v}_1$, $\vec{v}_2$	The velocities of the colliding spheres.
m_1, m_2	The masses of the spheres.
r_1, r_2	The radii of the spheres.
e	Coefficient of restitution.

To determine the changes in velocity for both spheres, we will use 2 equations. The conservation of momentum in 1 dimension:

$$m_1 v_1 + m_2 v_2 = m_1(v_1 + \Delta v_1) + m_2(v_2 + \Delta v_2)$$

and the definition of the coefficient of restitution:

$$e = \frac{v_{2+} - v_{1+}}{v_{1-} - v_{2-}}$$

where v_{1-} and v_{2-} are the 1-dimensional velocities before the collision, and v_{1+} and v_{2+} after the collision.

Using these we can determine the magnitude of the impulse j, which is the change of momentum of the objects (An excellent article about this including the derivation of this formula can be found in [5]):

$$j = \frac{-(1+e)(\Delta v)}{\frac{1}{m_1} + \frac{1}{m_2}} \tag{2.10}$$

where Δv is the approach velocity:

$$\Delta v = (\vec{v}_1 - \vec{v}_2) \cdot \frac{\vec{S}_2 - \vec{S}_1}{\|\vec{S}_2 - \vec{S}_1\|}$$

Now that we have the impulse, we just have to distribute it among the 2 spheres. The changes of velocity of the spheres thus are:

$$\Delta v_1 = \frac{j}{m_1}\vec{n}$$

$$\Delta v_2 = \frac{j}{m_2}\vec{n}$$

where $\vec{n}$ is the collision normal (we can calculate it as $\vec{n} = \frac{\vec{S}_2 - \vec{S}_1}{\|\vec{S}_2 - \vec{S}_1\|}$).

The only thing left to do is the correction of object positions. To calculate the change of positions we will use 2 equations. The first is the conservation of the center of mass:

$$m_1\vec{S}_1 + m_2\vec{S}_2 = m_1(\vec{S}_1 + \Delta\vec{S}_1) + m_2(\vec{S}_2 + \Delta\vec{S}_2)$$

The second is the assumption that after displacement the centers of the spheres will lie on the same line:

$$\vec{S}_2 - \vec{S}_1 = s((\vec{S}_2 + \Delta\vec{S}_2) - (\vec{S}_1 + \Delta\vec{S}_1))$$

We can calculate s like this:

$$s = \frac{(r_1 + r_2)}{\|\vec{S}_2 - \vec{S}_1\|} \tag{2.11}$$

After solving the above equations we get:

$$\Delta\vec{S}_1 = \frac{m_2(S_2 - S_1)(1 - s)}{m_1 + m_2} \tag{2.12}$$

$$\Delta\vec{S}_2 = \frac{-m_1(S_2 - S_1)(1 - s)}{m_1 + m_2} \tag{2.13}$$

So we change the positions accordingly.

Sphere-rod collision

The input parameters of the algorithm are:

$\vec{A}_1$, $\vec{A}_2$	The positions of the enclosing spheres of the rod.
$\vec{v}_1$, $\vec{v}_2$	The velocities of the enclosing spheres.
m_1, m_2	The masses of the enclosing spheres.
R	The radius of the rod.
$\vec{B}$	The position of the colliding sphere.
$\vec{v}_B$	The velocity of the colliding sphere.
m	The mass of the colliding sphere.
r	The radius of the colliding sphere.
e	Coefficient of restitution.

First, we need to check whether the 2 objects are overlapping. So we use 2.4 to determine where the collision point lies on the rod:

$$a = \frac{\vec{A} \cdot \vec{C}}{\vec{A} \cdot \vec{A}}$$

If the result is between 0 and 1, we calculate the distance vector of the sphere and the rod using 2.5:

$$\vec{D} = a\vec{A} - \vec{C}$$

If its length is less than the sum of the radii, no collision occurs. Otherwise we continue by calculating the effective mass of the rod at the point of collision using 2.6:

$$M = \frac{m_1 m_2}{m_2 - 2am_2 + a^2(m_1 + m_2)}$$

To calculate the impulse we also need to know the approach velocity of the 2 bodies at the point of collision. So we project the difference of the velocities of the surface points that collide onto the axis of collision. In case of the rod this velocity is calculated using 2.7:

$$\vec{v} = (1 - a)\vec{v}_1 + a\vec{v}_2$$

Now we project this onto the axis of collision, which is defined as $\vec{n} = \frac{\vec{D}}{\|\vec{D}\|}$:

$$v_\delta = \vec{n} \cdot (\vec{v}_B - \vec{v})$$

Now we can calculate the collision impulse using 2.10:

$$j = \frac{-(1 + e)v_\delta}{\frac{1}{m_1} + \frac{1}{M}}$$

From this we can determine the change of velocity of the sphere:

$$\Delta v = \frac{j\vec{n}}{m}$$

where $\vec{n}$ is the (normalized) axis of collision. In the case of the rod we must redistribute the impulse among the rod's enclosing spheres, and for that we use equations 2.8 and 2.9:

$$\Delta\vec{v}_1 = \frac{-(1+e)(1-a)\vec{n}j}{m_1}$$
$$\Delta\vec{v}_2 = \frac{-(1+e)(a)\vec{n}j}{m_2}$$

Now we have to correct the positions of the objects. First, we calculate s (the ratio of the displacement that must happen between the objects) according to 2.11:

$$s = \frac{(r+R)}{\|\vec{S}_2 - \vec{S}_1\|}$$

We can use this to change the position of the sphere using 2.12:

$$\Delta\vec{S} = \frac{M((\vec{S} + a(\vec{S}_2 - \vec{S}_1)) - \vec{S})(1-s)}{m+M}$$

The case of the rod is more complicated. First, we calculate the displacement of the contact point on the rod using 2.12:

$$\Delta\vec{D} = \frac{m((\vec{S}_1 + a(\vec{S}_2 - \vec{S}_1)) - \vec{S})(1-s)}{m+M}$$

Now we want to distribute this displacement among the endpoints of the rod. Do do that we use an equation similar to 2.8 and 2.9, which we just used above to distribute impulse. We modify these equations to work with displacement instead:

$$\Delta\vec{S}_1 = \frac{(1-a)\vec{D}(m_1+m_2)}{m_1} \tag{2.14}$$
$$\Delta\vec{S}_2 = \frac{-a\vec{D}(m_1+m_2)}{m_2} \tag{2.15}$$

Now we move the points accordingly, and we're done.

Rod-rod collision

The input parameters of the algorithm:

$\vec{A}_1$, $\vec{A}_2$	The positions of the enclosing spheres of the first rod.
$\vec{v}_{A_1}$, $\vec{v}_{A_2}$	The velocities of the enclosing spheres of the first rod.
m_{A_1}, m_{A_2}	The masses of the enclosing spheres of the first rod.
r_A	The radius of the first rod.
$\vec{B}_1$, $\vec{B}_2$	The positions of the enclosing spheres of the second rod.
$\vec{v}_{B_1}$, $\vec{v}_{B_2}$	The velocities of the enclosing spheres of the second rod.
m_{B_1}, m_{B_2}	The masses of the enclosing spheres of the second rod.
r_B	The radius of the second rod.
e	Coefficient of restitution.

Let's first introduce some auxiliary quantities:

$$\vec{A} = \vec{A}_2 - \vec{A}_1$$

$$\vec{B} = \vec{B}_2 - \vec{B}_1$$

$$\vec{C} = \vec{B}_1 - \vec{A}_1$$

Now we can use equation 2.1, 2.2 and 2.3 to calculate the parameters a, b and $\vec{D}$:

$$a = \frac{\vec{C} \cdot (\vec{B} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})}$$

$$b = \frac{\vec{C} \cdot (\vec{A} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})}$$

$$\vec{D} = \vec{C} + b\vec{B} - a\vec{A}$$

Let's define the axis of collision (which is also the collision normal) as:

$$\vec{n} = \frac{\vec{D}}{\|\vec{D}\|}$$

Collision only occurs when a and b are between 0 and 1, and the length of $\vec{D}$ is less than $r_A + r_B$. If there is a collision, we need to calculate the effective masses of both rods at the point of collision using 2.6:

$$M_A = \frac{m_{A_1} m_{A_2}}{m_{A_2} - 2am_{A_2} + a^2(m_{A_1} + m_{A_2})}$$

$$M_B = \frac{m_{B_1} m_{B_2}}{m_{B_2} - 2bm_{B_2} + b^2(m_{B_1} + m_{B_2})}$$

Now we need to determine the approach velocity of the 2 rods, so we use 2.7 to determine the velocities of the contact points on the rods:

$$\vec{v}_A = (1 - a)\vec{v}_{A_1} + a\vec{v}_{A_2}$$

$$\vec{v}_B = (1 - b)\vec{v}_{B_1} + b\vec{v}_{B_2}$$

We project the difference of these vectors onto the axis of collision to get the approach velocity:

$$v = \vec{n} \cdot \vec{v}_A - \vec{v}_B$$

Now we have everything we need to substitute into the collision equation 2.10 to get the collision impulse:

$$I = \frac{-(1 + e)v\vec{n}}{\frac{1}{M_A} + \frac{1}{M_B}}$$

Next, we distribute the collision impulse among the spheres similar to equations 2.8 and 2.9:

$$\Delta\vec{v}_{A_1} = \frac{(1-a)\vec{n}j}{m_{A_1}}$$

$$\Delta\vec{v}_{A_2} = \frac{(a)\vec{n}j}{m_{A_2}}$$

$$\Delta\vec{v}_{B_1} = \frac{(1-b)\vec{n}j}{m_{B_1}}$$

$$\Delta\vec{v}_{B_2} = \frac{(b)\vec{n}j}{m_{B_2}}$$

And we're done calculating the change in velocity, but we still need to correct the positions. First let's calculate the location of the contact points on the rods (more specifically the contact point's projection onto the rod's axis):

$$\vec{C}_A = \vec{A}_1 + a\vec{A}$$

$$\vec{C}_B = \vec{B}_1 + b\vec{B}$$

And the ratio of the displacement using 2.11:

$$s = \frac{(R_A + R_B)}{\|\vec{C}_B - \vec{C}_A\|}$$

Now we can use we use 2.12 and 2.13 to calculate the displacement of the surface points:

$$\Delta\vec{C}_A = \frac{(1-s)M_B(\vec{C}_B - \vec{C}_A)}{M_A + M_B}$$

$$\Delta\vec{C}_B = \frac{-(1-s)M_A(\vec{C}_B - \vec{C}_A)}{M_A + M_B}$$

We only need to distribute the displacement using equations 2.14 and 2.15 and we're done:

$$\Delta\vec{A}_1 = \frac{\vec{C}_A(1-a)(m_{A_1} + m_{A_2})}{m_{A_1}}$$

$$\Delta\vec{A}_2 = \frac{\vec{C}_A(a)(m_{A_1} + m_{A_2})}{m_{A_1}}$$

$$\Delta\vec{B}_1 = \frac{\vec{C}_B(1-b)(m_{B_1} + m_{B_2})}{m_{B_1}}$$

$$\Delta\vec{B}_2 = \frac{\vec{C}_B(b)(m_{B_1} + m_{B_2})}{m_{B_1}}$$

2.7 Forces

In a simulation there are many kinds of forces that can affect an object. Gravitational force, springs, etc. It's also quite common that multiple forces affect an object at once whose effects must be combined. In the following sections the computation of different forces is described. Note: The result is always a force, the parameters of the objects are not changed.

2.7.1 Springs

Springs exert forces on the objects they connect. They do this by applying Hooke's Law:

$$F = -kx \tag{2.16}$$

where F is the resulting force, k is the spring constant and x is the displacement from the equilibrium. Springs also have a damping coefficient, which is proportional to the relative velocity of the objects:

$$F_f = -cv \tag{2.17}$$

where c is the viscous damping coefficient.

A spring force calculation algorithm has the following parameters:

$\vec{A}$, $\vec{B}$	The positions of the 2 point masses (in this case spheres) that are connected by the spring.
$\vec{v}_A$, $\vec{v}_B$	The velocities of the spheres.
m_A, m_B	The masses of the spheres.
k	The spring constant.
l	The length of the spring at rest.
c	damping coefficient.

First we need to determine the axis of the spring force:

$$\vec{D} = \vec{B} - \vec{A}$$

From this we can calculate the distance of the objects and the force normal:

$$d = \|\vec{D}\|$$

$$\vec{n} = \frac{\vec{D}}{d}$$

Now we have the parameters to substitute into 2.16 to calculate the spring force:

$$F = -k(d - x)\vec{n}$$

We also need to calculate the damping. To do that we need to determine the approach velocity of the objects by projecting the difference of their velocities onto the fore normal:

$$v = (\vec{v}_B - \vec{v}_A) \cdot \vec{n}$$

Using this we can determine the damping force using 2.17:

$$F_f = -cv\vec{n}$$

2.7.2 Gravity

Gravity is easy to compute. The only parameters are:

m	The mass of the object.
G	The gravitational constant.
$\vec{g}$	The direction of gravity, implicitly assumed to be {0,-1,0}.

The gravitational force that affects the body (according to Newton[8]) is:

$$F_g = mG\vec{g}$$

2.7.3 Contact, friction and torque

In a scene sometimes objects come into resting contact. An example is an object resting on the table. There are 3 main types of algorithms solving this problem. All 3 are research topics and so it's hard to find introductory material on them.

The microcollision method [6] does not distinguish between collision and contact, collisions just become smaller and smaller. This method needs to register many collisions and seems to be quite slow for the number of objects we want to model.

Analytical methods solve contact between objects exactly, on the downside they are mathematically complex [3] [1].

Penalty force methods are easy to implement, but it's hard to keep objects from interpenetrating [7] [12].

In this engine I chose to use the penalty force method to handle contact, because it's relatively easy to combine with friction forces and torque. On the downside there's visible interpenetration when multiple objects are stacked on top of each other, or when heavy objects are put on top of light objects.

In the following section I will detail the calculation of contact forces between each type of object pair. Each force consists of a repulsion component (similar to a spring), a damping component (which dampens the relative movement of the objects), and a friction component (which can have both a translation and a rotation effect). We calculate each force only for the first object for reasons described in section 2.1.3.

Sphere-sphere contact

The input parameters of the algorithm are the following:

$\vec{S}_A$, $\vec{S}_B$	The positions of the spheres.
$\vec{v}_A$, $\vec{v}_B$	The velocities of the spheres.
$\vec{L}_A$, $\vec{L}_B$	The angular momenta of the spheres.
m_A, m_B	The masses of the spheres.
r_A, r_B	The radii of the spheres.

Constants used:

C_c	A constant that specifies the strength of the contact force.
C_d	A constant that specifies the strength of damping.
μ	Coefficient of friction

First we calculate the sum of the radii and the distance vector:

$$r_{AB} = r_A + r_B$$
$$\vec{D} = \vec{S}_B - \vec{S}_A$$

Now we check whether the spheres overlap by comparing r_{AB} with $\|\vec{D}\|$. If they do, we continue by calculating the force normal (the direction of the force) and the overlap of the objects:

$$\vec{n} = \|\vec{D}\|$$
$$o = r_{AB} - \|\vec{D}\|$$

From this we can calculate the repulsive force between the objects:

$$\vec{F_{c_A}} = -\frac{C_c \; o \; m_A \; m_B}{m_A + m_B}\vec{n} \tag{2.18}$$

Next we want to calculate the damping between the objects. For that we need their relative velocity:

$$v = (\vec{v}_A - \vec{v}_B) \cdot \vec{n}$$

Now we can substitute this into the damping equation, which is similar to 2.17, except it must take into account the mass of the objects so that the behavior is the same when masses are scaled:

$$\vec{F_{d_A}} = -\frac{C_d \; v \; m_A \; m_B}{m_A + m_B}\vec{n} \tag{2.19}$$

Next we need to calculate the friction that affects the object. We first calculate the angular velocity of the spheres from their angular momentum, radius and mass:

$$\omega_A = \frac{\vec{L}_A}{\frac{2}{5} m_A \, r_A^2}$$

$$\omega_B = \frac{\vec{L}_B}{\frac{2}{5} m_B \, r_B^2}$$

We also need the vectors connecting the centers of the objects and the points of contact on their surfaces:

$$\vec{r_\alpha} = \vec{n} r_A$$

$$\vec{r_\beta} = \vec{n} r_B$$

From this we can calculate the velocities of the points of contact on the surface of the spheres:

$$v_\alpha = \vec{v}_A + (\omega_A \times \vec{r_\alpha})$$

$$v_\beta = \vec{v}_B + (\omega_B \times \vec{r_\beta})$$

Next we want to project their relative velocity into the contact plane:

$$v_{\alpha\beta} = (\vec{n} \times ((v_\beta - v_\alpha) \times \vec{n}))$$

Now that we have the velocity we can calculate the friction between the spheres. Realistically friction should not depend on the relative velocity of the surfaces. This poses a problem in a simulation: After a while the relative velocity of the surfaces would oscillate around 0:

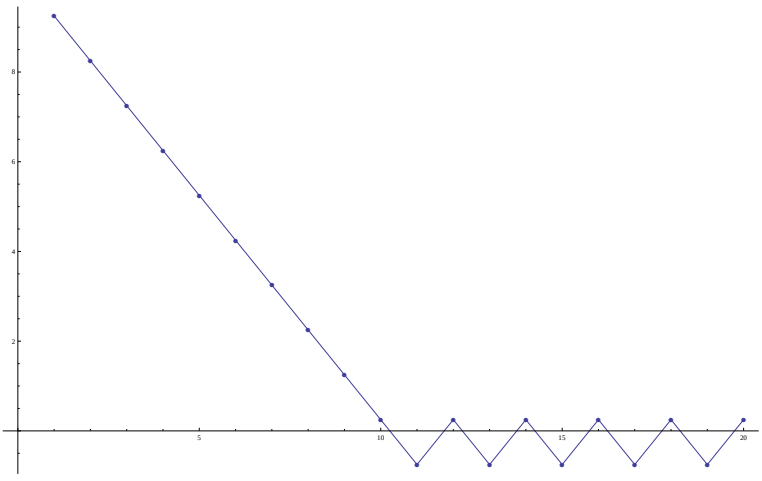


Figure 2.8: Method 1: The value never reaches 0

The correct behavior would be to linearly decrease until reaching 0. To solve this problem we make friction dependent on the relative velocity. This way the relative velocity nicely converges toward 0:

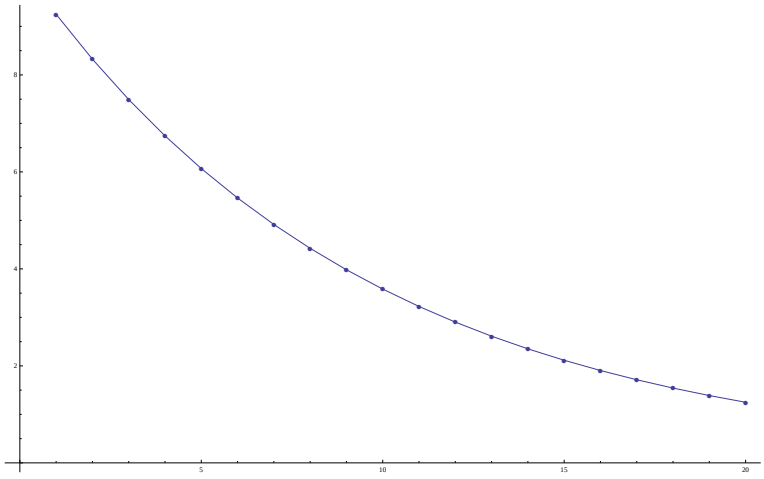


Figure 2.9: Method 2: The value converges toward 0 exponentially

The problem with this approach is that it takes a long time for the velocity to become 0, and it looks unrealistic. To mitigate this problem we can divide with the square root of the length of the relative velocity vector. The result is more similar to linear convergence:

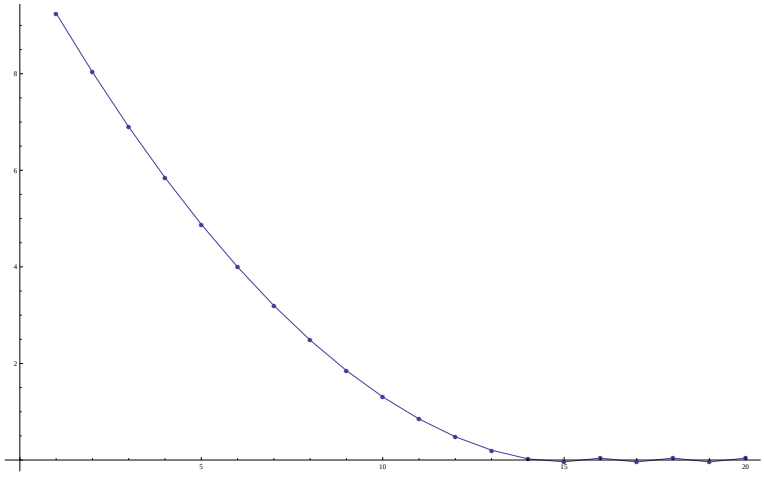


Figure 2.10: Method 3: A combination of the first 2 methods

There is still some oscillation, but in practice it is so small that it's unnoticeable. Also, the average of the oscillating values is 0. This method looks quite realistic, so I'll be using this method in this work. It is also possible to combine these methods. For example when the length of the relative velocity vector is more than 1, *Method 1* would be used, otherwise *Method 3*.

So the formula for friction is:

$$\vec{F_{f_A}} = \frac{\mu \; m_A \; m_B \; o \; v_{\alpha\beta}}{(m_A + m_B)\sqrt{\|v_{\alpha\beta}\|}}$$

Friction force also creates torque on the sphere, which we calculate like this:

$$\tau_A = r_\alpha \times \vec{F_f}$$

The total force affecting the sphere is $\vec{F_{c_A}} + \vec{F_{d_A}} + \vec{F_{f_A}}$ and the torque is τ_A.

Sphere-rod contact

There are two ways a sphere can be affected during a sphere-rod contact: When it gets into direct contact with a rod, and when a rod containing the sphere gets into contact with a sphere. Since the two cases are very similar, I describe both in the same section.

The parameters of the algorithm:

$\vec{A}$	The position of the sphere.
$\vec{v}_A$	The velocity of the sphere.
$\vec{L}_A$	The angular momentum of the sphere.
m_A	The mass of the sphere.
r_A	The radius of the sphere.
$\vec{B}_1$, $\vec{B}_2$	The positions of the spheres enclosing the rod.
$\vec{v}_{B_1}$, $\vec{v}_{B_1}$	The velocities of the enclosing spheres.
m_{B_1}, m_{B_2}	The masses of the enclosing spheres.
r_B	The radius of the rod.

Constants used:

C_c	A constant that specifies the strength of the contact force.
C_d	A constant that specifies the strength of damping.
μ	Coefficient of friction

Since we have a rod we define some auxiliary parameters:

$$\vec{B} = \vec{B}_2 - \vec{B}_1$$

$$\vec{C} = \vec{A} - \vec{B}_1$$

and calculate the appropriate parameters using equations 2.4, 2.5 and 2.6:

$$b = \frac{\vec{B} \cdot \vec{C}}{\vec{B} \cdot \vec{B}}$$

$$\vec{D} = b\vec{B} - \vec{C}$$

$$\vec{n} = \|\vec{D}\|$$

$$m_B = \frac{m_{B_1} m_{B_2}}{m_{B_2} - 2bm_{B_2} + b^2(m_{B_1} + m_{B_2})}$$

where $\vec{D}$ is the distance vector and $\vec{n}$ is the contact force normal. Next we calculate the overlap of the objects:

$$o = r_A + r_B - \|\vec{D}\|$$

Now we have everything to calculate the contact force:

$$\vec{F_{c_s}} = -\frac{C_c \; o \; m_A \; m_B}{m_A + m_B}\vec{n}$$

If we are calculating the force acting on the sphere A, we can use this directly. If we're calculating the force on B_1 or B_2, we must determine what portion of the force affects the sphere. For this we simply use linear interpolation:

$$\vec{F_{c_r}} = (1-b)\vec{F_{c_s}} \tag{2.20}$$

To calculate the damping force we need the relative velocity of the objects (we use equation 2.7 to calculate the velocity of the rod's contact point):

$$\vec{v}_B = (1-b)\vec{v}_{B_1} + b\vec{v}_{B_2})$$

$$v = (\vec{v}_A - \vec{v}_B) \cdot \vec{n}$$

Now we can substitute into 2.19 to determine the damping affecting the sphere:

$$\vec{F_{d_s}} = -\frac{C_d \; v \; m_A \; m_B}{m_A + m_B}\vec{n}$$

In case of the rod we determine the force by interpolating by b:

$$\vec{F_{d_r}} = (1-b)F_{d_s}$$

The only thing left to do is to calculate the friction between the sphere and the rod. First, we need to determine the lateral velocity component of the objects' relative velocity (how fast the slide on each other), so we calculate the velocity of the contact points on the surface of both objects. First, we calculate the velocity of the sphere's contact point:

$$\omega_A = \frac{\vec{L}_A}{\frac{2}{5}m_A \; r_A^2}$$

$$\vec{r_\alpha} = \vec{n} \; r_A$$

$$v_\alpha = \vec{v}_A + (\omega_A \times \vec{r_\alpha})$$

Next we calculate the the velocity of the rod's surface point:

$$v_\beta = (1-b)\vec{v}_{B_1} + b\vec{v}B_2$$

Next we want to project their relative velocity into the contact plane:

$$v_{\alpha\beta} = (\vec{n} \times ((v_\beta - v_\alpha) \times \vec{n}))$$

Now we can finally calculate the friction between the objects:

$$\vec{F_{f_A}} = \frac{\mu \; m_A \; m_B \; o \; v_{\alpha\beta}}{(m_A + m_B)\sqrt{\|v_{\alpha\beta}\|}}$$

This directly affects the sphere and also causes rotation, which we calculate like this:

$$\tau_A = r_\alpha \times \vec{F_f}$$

In case of the rod we consider only the part of the force that affects this end:

$$\vec{F_{f_B 1}} = (1 - b)\vec{F_{f_A}}$$

So the total force affecting the sphere is $\vec{F_{c_s}} + \vec{F_{d_s}} + \vec{F_{f_A}}$ and the torque is τ_A. The total force affecting the sphere at the end of the rod is $\vec{F_{c_r}} + \vec{F_{d_r}} + \vec{F_{f_B 1}}$

Rod-rod contact

The parameters of the algorithm:

$\vec{A}_1$, $\vec{A}_2$	The positions of the spheres enclosing the rod.
$\vec{v}_{A_1}$, $\vec{v}_{A_1}$	The velocities of the enclosing spheres.
m_{A_1}, m_{A_2}	The masses of the enclosing spheres.
r_A	The radius of the rod.
$\vec{B}_1$, $\vec{B}_2$	The positions of the spheres enclosing the rod.
$\vec{v}_{B_1}$, $\vec{v}_{B_1}$	The velocities of the enclosing spheres.
m_{B_1}, m_{B_2}	The masses of the enclosing spheres.
r_B	The radius of the rod.

Constants used:

C_c	A constant that specifies the strength of the contact force.
C_d	A constant that specifies the strength of damping.
μ	Coefficient of friction

Since we have 2 rods we define some auxiliary parameters:

$$\vec{A} = \vec{A}_2 - \vec{A}_1$$
$$\vec{B} = \vec{B}_2 - \vec{B}_1$$
$$\vec{C} = \vec{B} - \vec{A}$$

Using equations 2.1, 2.2 and 2.3 we calculate the parameters a, b and $\vec{D}$:

$$a = \frac{\vec{C} \cdot (\vec{B} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})}$$
$$b = \frac{\vec{C} \cdot (\vec{A} \times (\vec{A} \times \vec{B}))}{(\vec{A} \times \vec{B}) \cdot (\vec{A} \times \vec{B})}$$
$$\vec{D} = \vec{C} + b\vec{B} - a\vec{A}$$

The contact force direction will be:

$$\vec{n} = \frac{\vec{D}}{\|\vec{D}\|}$$

Now we need to calculate the effective masses of the rods:

$$m_A = \frac{m_{A_1} m_{A_2}}{m_{A_2} - 2am_{A_2} + a^2(m_{A_1} + m_{A_2})}$$

$$m_B = \frac{m_{B_1} m_{B_2}}{m_{B_2} - 2bm_{B_2} + b^2(m_{B_1} + m_{B_2})}$$

And the repulsive contact force is:

$$\vec{F_c} = -\frac{C_c \; o \; m_A \; m_B}{m_A + m_B}\vec{n}$$

of which the force affecting the enclosing sphere is:

$$\vec{F_{c_A 1}} = (1 - a)\vec{F_c}$$

To calculate damping we need to determine the approach velocity of the contact points on the 2 rods, for that we use 2.7:

$$\vec{v}_A = (1 - a)\vec{v}_{A_1} + a\vec{v}_{A_2}$$

$$\vec{v}_B = (1 - b)\vec{v}_{B_1} + b\vec{v}_{B_2}$$

Using these parameters we can determine the damping force:

$$\vec{F_d} = -\frac{C_d \; v \; m_A \; m_B}{m_A + m_B}\vec{n}$$

And the force affecting the enclosing sphere is:

$$\vec{F_{d_A 1}} = (1 - a)\vec{F_d}$$

2.7.4 Contact with static objects

This section is similar to the section called **Collision with static objects** (2.6.1), so we'll handle all 3 cases (sphere-polygon, sphere-box, rod-box) in the same section.

The main input parameters of the algorithm, these will be derived below:

$\vec{A}$	The position of the moving object.
$\vec{n}$	The contact force normal, pointing from the moving object toward the static object.
m	The mass of the object.
v	The approach velocity of the object (scalar quantity).
o	The overlap between the moving and the static object.

Constants used:

C_c	A constant that specifies the strength of the contact force.
C_d	A constant that specifies the strength of damping.
μ	Coefficient of friction

In case of a sphere m is considered to be the mass of the sphere. In case of a rod m will be the effective mass at the point of collision, calculated using 2.6

Next, we want to find a point on the static object that is closest to the moving object. In case of a sphere-polygon pair it's simply the sphere's center projected onto the plane of the polygon. In case of a sphere-box pair we find this point using the algorithm from the **Sphere-box collision** section (2.6.1). In case of a rod-box pair we follow the procedure described in the **rod-box collision** section (2.6.1) and also calculate parameter a. This way we can determine the contact force normal $\vec{n}$ and the overlap. We also need the mass of the objects. If it's a sphere we directly use its mass, in case of a rod we need to calculate its effective mass m at the point of collision using 2.6. From this we can calculate the repulsive contact force affecting the object:

$$\vec{F_c} = -C_c \; o \; m \; \vec{n}$$

If the moving object is a rod, the force affecting the enclosing sphere is:

$$\vec{F_{c_A 1}} = (1 - a)\vec{F_c}$$

Next we want to calculate the damping affecting the moving object, and for that we need the approach velocity of the moving object. In case of a sphere it will simply be the velocity of the sphere projected onto the n vector:

$$v = \vec{n} \cdot \vec{v_s}$$

In case of a rod we calculate the velocity using 2.7:

$$\vec{v} = (1 - a)\vec{v_1} + a\vec{v_2}$$

(where $\vec{v_1}$ and $\vec{v_2}$ are the velocities of the enclosing spheres of the rod.) and then project it onto n:

$$v = \vec{v} \cdot \vec{n}$$

Using these parameters we can determine the damping force:

$$\vec{F_d} = -C_d \; v \; m \; \vec{n}$$

The only thing left to do is to calculate the friction affecting the object. So we first project the velocity of the object onto the contact plane:

$$\vec{v}_{\alpha\beta} = \vec{n} \cdot (\vec{v} \cdot \vec{n})$$

From that we can calculate the friction:

$$\vec{F_f} = \frac{\mu \; m \; o \; \vec{v}_{\alpha\beta}}{\sqrt{\|v_{\alpha\beta}\|}}$$

This directly affects the sphere and also causes rotation, which we calculate like this:

$$\tau_A = r_\alpha \times \vec{F_f}$$

In case of the rod we need to calculate the forces affecting the ends:

$$\vec{F_{f_A 1}} = (1 - a)\vec{F_f}$$

$$\vec{F_{f_A 2}} = (a)\vec{F_f}$$

2.7.5 Numerical integration

In a physics simulation our goal is to model the physical behavior of a system of physical objects in time. The problem is the following: Given the current state of the system (at time t) and the factors that affect the its state, determine the state of the system at time $t + \Delta t$.

In this simulation we consider 2 types of factors: impulses and forces. The case of impulses is simple: Since they are instantaneous, they either happen or they don't: The cause of their effect is easy to determine. The case of forces this is more complicated: They affect the objects for a longer time, probably over multiple frames, and they can change over time, so their effect changes throughout the frame. In mathematical terms we want to find the *displacement* of the object, which we can express through its velocity:

$$p_{t+\Delta t} = p_t + \int_t^{t+\Delta t} v(t)\,\mathrm{d}t$$

Velocity in turn can be expressed using acceleration:

$$v_{t+\Delta t} = v_t + \int_t^{t+\Delta t} a(t)\,\mathrm{d}t$$

where the acceleration a is calculated by using Newton's second law:

$$a = \frac{F}{m}$$

From this we can see that integration is necessary to determine the position of the object in the next frame. Since symbolic integration is not possible for anything but the simplest of frames, we need to use numerical integration. The simplest method for this purpose is Euler's method. It's defined as:

$$y_{t+\Delta t} = y_t + h\, f(t_n, y_n)$$

where h is the time step. In this case the state of the object in time step $t + \Delta t$ would be calculated like this:

$$p_{t+\Delta t} = p_t + (\Delta t)v$$

$$v_{t+\Delta t} = v_t + (\Delta t)a$$

Although this method is fast and simple, it's quite inaccurate and forces that are large and/or change rapidly over time can lead to the instability of the system.

Therefore in physics engines numerical integration is handled using more sophisticated methods. Maybe the most widespread of these is the fourth-order Runge–Kutta method (RK4). Its definition is the following:

Let an initial value problem be specified as follows:

$$y' = f(t, y), \quad y(t_0) = y_0$$

Then, the RK4 method for this problem is given by the following equations:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = tn + h$$

where y_{n+1} is the RK4 approximation of $y(t_{n+1})$, and

$$\begin{aligned} k_1 &= h\, f(t_n, y_n) \\ k_2 &= h\, f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \\ k_3 &= h\, f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \\ k_4 &= h\, f(t_n + h, y_n + k_3) \end{aligned}$$

Thus, the next value (y_{n+1}) is determined by the present value (y_n) plus the weighted average of 4 deltas. The above formulae are valid for both scalar- and vector-valued functions (i.e., y can be a vector and f an operator).

Therefore to calculate the state of the system in the next step, we need to calculate the 4 deltas, which means we must calculate the state of the system 4 times at different points, and use the weighted average of these results.

In practice this means that we need 4 kernel calls, one for each phase. Each kernel calculates the next delta (the next state of the system) using the previous one. The *state of the system* is the position, velocity and angular momentum of every sphere (we exclude orientation since it does not influence force calculation in any way). So we need to store the state of the original system (y_n), the last delta (k_n), and we also need to accumulate the weighted average of the deltas. This means we must store the state of the system 2 additional times.

Although this approach need more computattions than the Euler method, it's more robust, enabling us to calculate less physics frames each second and keep the simulation stable.

2.8 Constants

Several constants are used by the engine, which control different parameters of the simulation. Some of these affect memory allocation and are used during compilation, some others can be changed at runtime. We will only discuss the latter.

The constants that can be changed at runtime can be used to tune the simulation to be more stable or realistic. Their range of usability is determined by the accuracy of the numerical integration that is used. RK4 is quite good at handling stiff equations, but after a while the system might become unstable even with RK4. The parameters that can be changed are the following:

Name: Contact activation threshold

Default value: 1.0

Description: This constant specifies when to use contact forces and when to use collision between objects. If the product of this constant and the radius of the object (or the minimum of the radii if there are multiple objects) is less than the approach velocity, the collision phase is performed (at the end of which the objects stop overlapping, so contact does not happen at the contact phase). Otherwise collision is ignored. This constant is not used during contact force calculation, contact always happens when objects overlap. Increasing this constant means that collisions will be less likely to happen. Decreasing it means that collisions will be more likely, and might never become contact.

Name: Coefficient of restitution

Default value: 0.75

Description: The coefficient of restitution determines how much energy is lost during collision. 0 means the collision is completely inelastic ant the objects "stick together" at collision. A value of 1 means the collision is completely elastic and no energy is lost during collision. A value higher than 1 means energy is added to the system during each collision and is not recommended.

Name: Contact damping

Default value: 12.5

Description: This constant specifies how much damping should be applied when objects are in contact. Decreasing this constant will lead to more "bouncing" between objects, increasing it might cause objects to stick together and "pull" other objects with them. In extreme cases a high value can cause divergence from the point of equilibrium.

Name: Contact force

Default value: 1000

Description: This constant specifies how strong the repulsive force should be between objects. A lower value means more overlap, a higher value might lead to a stiff differential equation and the contact force might push objects strongly apart.

Name: Friction

Default value: 50

Description: This determines how strong friction is between objects, and between objects and the ground.

Name: Gravity

Default value: 1

Description: This determines the strength of gravity. A higher value means stronger gravity.

Name: Collisions

Default value: TRUE

Description: Should collisions happen? Because of the modular design of the engine collisions can be easily disabled, this switch enables this feature. The system might be less stable when this is turned off because collisions limit the approach velocity of objects during the contact phase (contact only happens when the approach velocity is below a certain threshold).

Conclusion

In this work we have shown how to create a physics engine for the CUDA architecture. It can handle a large number of objects by providing fast collision detection, the performance of which is not affected by the arrangement or the speed of change of the scene, only its size. The advanced numerical integration method makes the simulation more stable and thus allows stronger forces to be used. It's created as modular and is extensible. A good example of this is the possibility to turn off collision handling simply by not calling a kernel.

As future work the next logical step would be to accelerate collision detection using algorithms already described in this work. It should be possible to achieve significant speed improvements, especially for a large number of objects. Another improvement would be to use the features of later Compute Capabilities to improve the speed of contact handling. This would improve speeds especially when handling compound objects. The interface could also be extended to interact with the simulation more directly. It could make it possible to accelerate/decelerate objects, lift them up, etc.

2.9 Benchmark

The following chart shows the performance of the different phases of the algorithm for a scene containing only spheres. The spheres were placed randomly close to each other, and then the simulation was run for 1000 frames. This was run on a **GeForce 8800 GTS 512 G92**

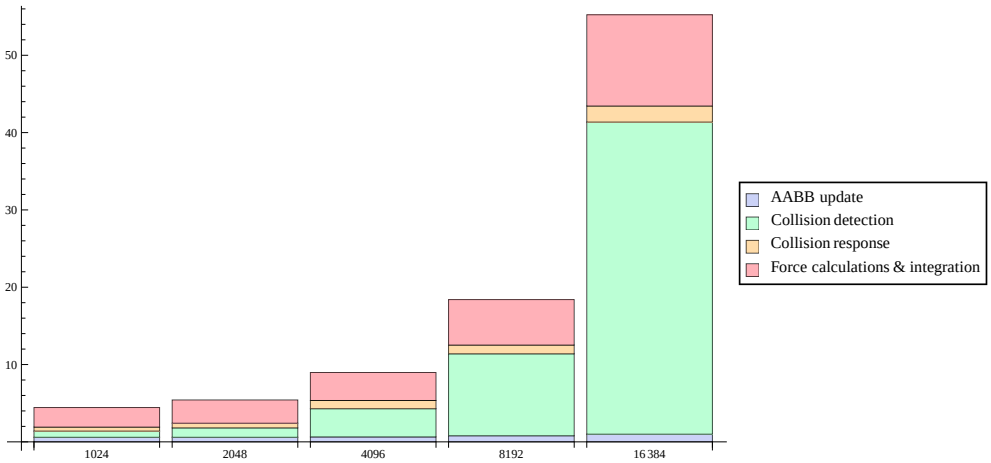


Figure 2.11: The time it takes for each phase to complete in milliseconds for different numbers of objects

The figure shows that all phases – except for the collision detection – scale approximately linearly with the number of objects (the lowest bar shows the length of the AABB update phase, the one above shows the length of the collision detection phase, etc.)

Appendix

2.10 Engine interface

The programming interface for the engine was created to be easy to use, and does not depend on CUDA-specific data types. It enables the user to add objects to the scene. The whole scene is handled through a single class object which has interface functions and directly accessible data fields.

Since the scene information is stored on the device, it is necessary to copy this data to the host to be usable outside the engine (i.e. to draw the elements of the scene). This is of course also supported.

Debugging device code is more difficult than debugging host code, since no debugging features are supported by default (stepping through code, watching variables, etc.). These features are only available in Visual Studio through NVIDIA©Parallel Nsight^{TM}, and an additional graphics card (or an additional computer) is required to debug device code with it. Therefore the only way do track the progress of device code execution is to write variables to device memory, and when the kernel has completed execution copy its contents to the host, and display them there. Unfortunately if kernel execution fails (i.e. gets into an infinite loop or a memory operation fails) then this copying is not possible.

Also, each phase of the physics simulation step can be (and must be) called separately.

2.10.1 Creation of new physical objects

Name: addSphere

Parameters: `float` x, `float` y, `float` z, `float` vx, `float` vy, `float` vz, `float` $angularX$, `float` $angularY$, `float` $angularZ$, `float` $mass$, `float` r, `float` $fixed$

Value returned: The index of the sphere that was added

Description: This adds a sphere at location $\{x, y, z\}$, with velocity $\{vx, vy, vz\}$, angular momentum $\{angularX, angularY, angularZ\}$, mass m and radius r. The last parameter is optional and specifies whether it should be a static or a moving object (a positive value for this parameter means that the object is static).

Name: addRod

Parameters: `int` A, `int` B, `float` r, `float` l

Value returned: The index of the rod that was added

Description: This adds a rod connecting spheres with indexes A and B, with radius r. The last parameter is optional and specifies whether the rod have a fixed length l. This can be used to simulate capsules for example.

Name: addPolygon

Parameters: `float` centerX, `float` centerY, `float` centerZ, `float` firstX, `float` firstY, `float` firstZ, `float` secondX, `float` secondY, `float` secondZ

Value returned: The index of the polygon that was added

Description: This adds a polygon (shaped like a parallelogram) specified by 3 vectors. The vector {centerX, centerY, centerZ} specifies the location of the corner of the polygon, {firstX, firstY, firstZ} and {secondX, secondY, secondZ} specify the 2 edges connected to the corner. **WARNING:** The polygon is facing at the direction of {firstX, firstY, firstZ}×{secondX, secondY, secondZ}

Name: addCapsule

Parameters: `float` x1, `float` y1, `float` z1, `float` x2, `float` y2, `float` z2, `float` vx, `float` vy, `float` vz, `float` m, `float` r

Value returned: The index of the capsule that was added

Description: This function adds a capsule. Its ends (the centers of the enclosing spheres) will be at {x1, y1, z1} and {x2, y2, z2}. The initial velocity will be {vx, vy, vz}. It will have a mass m and radius r

Name: addBox

Parameters: `float` cornerX, `float` cornerY, `float` cornerZ, `float` Ax, `float` Ay, `float` Az, `float` Bx, `float` By, `float` Bz, `float` Cx, `float` Cy, `float` Cz

Value returned: The index of the box that was added.

Description: This adds a box specified by 4 vectors. One of its corners will be at {cornerX, cornerY, cornerZ}, and the 3 edges connected to it will be specified by the vectors {Ax, Ay, Az}, {Bx, By, Bz} and {Cx, Cy, Cz}, **which must be orthogonal**.

Name: addSpring

Parameters: `int` A, `int` B, `float` k, `float` l, `float` d

Value returned: The index of the spring that was added.

Description: This adds a spring connecting spheres with indexes A and B. k is the spring constant, l is the length of the spring and d is the damping coefficient.

2.10.2 Copying scene information from device to host

Name: updateObjectsInHostMemory

Parameters: N/A

Value returned: N/A

Description: This function updates the data about the scene in host memory. The fields copied are the following: Position, radius and orientation of spheres and all data about rods (from this data only the sphere indexes are relevant), boxes and polygons. These are accessible through the following arrays in host memory: `h_pointmass.velocity`, `h_spheres.radius`, `h_spheres.orientationX`, `h_spheres.orientationY`, `h_spheres.orientationZ`, `h_rods.rod`, `h_planes.plane[0]`, `h_planes.plane[1]`, `h_planes.plane[2]`, `h_cuboids.cuboid[0]`, `h_cuboids.cuboid[1]`, `h_cuboids.cuboid[2]`, `h_cuboids.cuboid[3]` Note: internally rods are named cylinders, polygons are named planes and boxes are named cuboids.

2.10.3 Debug variables

Name: updateDebugVarsInHostMemory

Parameters: N/A

Value returned: N/A

Description: This function copies the memory area used to store auxiliary variables (mostly used for debugging) from device to host memory. This must be called before the values of these variables can be displayed

2.10.4 Physics simulation phases

The physics simulation requires the use of these phases in the sequence they are described. The collision phase can be omitted. The AABB update phase and the collision detection phase might also be omitted in some frames to conserve computation time.

Name: updateAABBs

Parameters: N/A

Value returned: N/A

Description: Updates the AABBs of objects (since their position has changed)

Name: broadPhaseCD

Parameters: N/A

Value returned: N/A

Description: Broad phase collision detection, which controls whether the AABBs of objects overlap. Further (i.e. narrow phase) collision detection is integrated into the collision handling and contact handling algorithms.

Name: handleCollisions

Parameters: N/A

Value returned: N/A

Description: Handles the collisions between objects. If omitted no collisions happen.

Name: updatePositions

Parameters: `time step`

Value returned: N/A

Description: Calculates force between objects, performs numerical integration and updates the positions and velocities of objects.

2.11 Scene file format

The front end is capable of reading object data from a file. It's a simple text file where each line specifies what object should be added. Each line begins with a (case sensitive) letter specifying the object type, which is followed by the parameters of the function call as described below. Lines beginning with unspecified letters are ignored (can be used as comments). The exact syntax of the functions referred to is described in the section above.

Syntax: `S x a b c d e f g h i j k l`

Corresponding function call:
`addSphere(a, b, c, d, e, f, g, h, i, j, k, l);`

Description: Adds a sphere to the scene by calling the `addSphere` function. The value of x is an integer between 0 and 1024 and is used to refer to the sphere that was added (to connect other objects to it). If the same reference is used multiple times the last one is valid.

Syntax: `R x y r`

Corresponding function call:
`addRod(indexStoredAsX, indexStoredAsY, r);`

Description: Adds a rod to the scene by calling the `addRod` function. The value of x and y is an integer between 0 and 1024 and are referring to the spheres that will be connected. r is the radius.

Syntax: `C a b c d e f g h i j k`

Corresponding function call:
`addCapsule(a, b, c, d, e, f, g, h, i, j, k);`

Description: Adds a capsule to the scene by calling the `addCapsule` function.

Syntax: `P a b c d e f g h i`

Corresponding function call:
`addPolygon(a, b, c, d, e, f, g, h, i);`

Description: Adds a polygon to the scene by calling the `addPolygon` function. **WARNING!** The sequence of vectors is important, they specify which way the polygon is facing. For details see the description of `addPolygon`.

Syntax: `K x y c d e`

Corresponding function call:
`addSpring(indexStoredAsX, indexStoredAsY, c, d ,e);`

Description: Adds a spring to the scene by calling the `addSpring` function. The value of x and y is an integer between 0 and 1024 and are referring to the spheres that will be connected.

Syntax: `B a b c d e f g h i j k l`

Corresponding function call:

```
addBox(a, b, c, d, e, f, g, h, i, j, k, l);
```

Description: Adds a box to the scene by calling the `addBox` function.

Example

The following is a valid scene description file:

```
S 1 -1 4 0 0 1 0 0 0 0 1 0.2 0
S 2 1 4 0 0 1 0 0 0 0 1 0.2 0
R 1 2 0.1
S 3 0 5 -1 0 1 0 0 0 0 1 0.2 0
S 4 0 5 1 0 1 0 0 0 0 1 0.2 0
R 3 4 0.1
S 5 0 1 0 0 1 0 0 0 0 1 0.2 0
S 6 0 3 0 0 1 0 0 0 0 1 0.2 0
R 5 6 0.1
K 5 6 100 2 1
this is a comment
```

The first 2 lines add a sphere to the scene with reference numbers 1 and 2. These numbers are used in the following line to connect the spheres with a rod. This process is repeated 2 more times, then at the end a spring is added between the last 2 spheres (using references 5 and 6). References can be reused, so the following file is (almost) equivalent to the previous one:

```
S 17 -1 4 0 0 1 0 0 0 0 1 0.2 0
S 23 1 4 0 0 1 0 0 0 0 1 0.2 0
R 17 23 0.1
S 17 0 5 -1 0 1 0 0 0 0 1 0.2 0
S 23 0 5 1 0 1 0 0 0 0 1 0.2 0
R 17 23 0.1
another comment
S 17 0 1 0 0 1 0 0 0 0 1 0.2 0
S 23 0 3 0 0 1 0 0 0 0 1 0.2 0
R 17 23 0.1
K 17 23 100 2 1
```

The only difference is that since references have been reused, the previously added spheres can no longer be referred to. Since the file is processed sequentially, it's also not possible to use indexes before they were created.

2.12 The contents of the CD

The CD contains the source code and executable version of the program including all dlls that are needed to run it. It also contains scene description files that can be read by the program. These showcase the capabilities of the engine. For further information on the contents of the CD see the included `CDcontents.txt` file at the root folder of the CD.

2.12.1 Program usage

The program accepts the following input:

F1 Shows a window displaying the controls available

Arrow keys Move camera around

Left click + move mouse Change camera direction

Right click + move mouse Change camera height

A Start/stop simulation

S Progress simulation 1 step forward

R Read input file called `input.txt`

K Turn on/off collisions.

T Turn on/off real time physics. If it's turned off, the fixed time step is 20 ms long.

D/C Increase/Decrease the contact force activation threshold.

F/V Increase/Decrease damping of contact

G/B Increase/Decrease contact force

H/N Increase/Decrease friction

J/M Increase/Decrease gravity

Z/X Increase/Decrease coefficient of restitution

Esc Quit

The input file must be placed in the same directory as the exe file.

Hardware requirements

This program requires CUDA compatible hardware to run. This means one of the following cards:

GeForce GTX 590, GeForce GTX 580, GeForce GTX 570, GeForce GTX 560 Ti, GeForce GTX 560, GeForce GTX 550 Ti, GeForce GTX 480, GeForce GTX 470, GeForce GTX 465, GeForce GTX 460, GeForce GTX 460 SE, GeForce GTS 450, GeForce GT 440, GeForce GT 430, GeForce GT 420, GeForce GTX 295, GeForce GTX 285, GeForce GTX 280, GeForce GTX 275, GeForce GTX 260, GeForce GTS 250, GeForce GTS 240, GeForce GT 240, GeForce GT 220, GeForce 210/G210, GeForce 9800 GX2, GeForce 9800 GTX+, GeForce 9800 GTX, GeForce 9800 GT, GeForce 9600 GSO, GeForce 9600 GT, GeForce 9500 GT, GeForce 9400 GT, GeForce 9400 mGPU, GeForce 9300 mGPU, GeForce 9100 mGPU, GeForce 8800 Ultra, GeForce 8800 GTX, GeForce 8800 GTS, GeForce 8800 GT, GeForce 8800 GS, GeForce 8600 GTS, GeForce 8600 GT, GeForce 8600 mGT, GeForce 8500 GT, GeForce 8400 GS, GeForce 8300 mGPU, GeForce 8200 mGPU, GeForce 8100 mGPU, GeForce GT 555M, GeForce GT 550M, GeForce GT 540M, GeForce GT 525M, GeForce GT 520M, GeForce GTX 480M, GeForce GTX 470M, GeForce GTX 460M, GeForce GT 445M, GeForce GT 435M, GeForce GT 425M, GeForce GT 420M, GeForce GT 415M, GeForce GTX 285M, GeForce GTX 280M, GeForce GTX 260M, GeForce GTS 360M, GeForce GTS 350M, GeForce GTS 260M, GeForce GTS 250M, GeForce GT 335M, GeForce GT 330M, GeForce GT 325M, GeForce GT 320M, GeForce 310M, GeForce GT 240M, GeForce GT 230M, GeForce GT 220M, GeForce G210M, GeForce GTS 160M, GeForce GTS 150M, GeForce GT 130M, GeForce GT 120M, GeForce G110M, GeForce G105M, GeForce G103M, GeForce G102M, GeForce G100, GeForce 9800M GTX, GeForce 9800M GTS, GeForce 9800M GT, GeForce 9800M GS, GeForce 9700M GTS, GeForce 9700M GT, GeForce 9650M GT, GeForce 9650M GS, GeForce 9600M GT, GeForce 9600M GS, GeForce 9500M GS, GeForce 9500M G, GeForce 9400M G, GeForce 9300M GS, GeForce 9300M G, GeForce 9200M GS, GeForce 9100M G, GeForce 8800M GTX, GeForce 8800M GTS, GeForce 8700M GT, GeForce 8600M GT, GeForce 8600M GS, GeForce 8400M GT, GeForce 8400M GS, GeForce 8400M G, GeForce 8200M G, Quadro 6000, Quadro 5000, Quadro 4000, Quadro 2000, Quadro 600, Quadro FX 5800, Quadro FX 5600, Quadro FX 4800, Quadro FX 4700 X2, Quadro FX 4600, Quadro FX 3800, Quadro FX 3700, Quadro FX 1800, Quadro FX 1700, Quadro FX 580, Quadro FX 570, Quadro FX 380, Quadro FX 370, Quadro NVS 450, Quadro NVS 420, Quadro NVS 295, Quadro NVS 290, Quadro Plex 1000 Model IV, Quadro Plex 1000 Model S4, Quadro 5000M, Quadro FX 3800M, Quadro FX 3700M, Quadro FX 3600M, Quadro FX 2800M, Quadro FX 2700M, Quadro FX 1800M, Quadro FX 1700M, Quadro FX 1600M, Quadro FX 880M, Quadro FX 770M, Quadro FX 570M, Quadro FX 380M, Quadro FX 370M, Quadro FX 360M, Quadro NVS 320M, Quadro NVS 160M, Quadro NVS 150M, Quadro NVS 140M, Quadro NVS 135M, Quadro NVS 130M, Tesla C2050/2070, Tesla M2050/M2070, Tesla S2050, Tesla S1070, Tesla M1060, Tesla C1060, Tesla C870, Tesla D870, Tesla S870

Other requirements are dependent on scene complexity. The memory requirements of the engine are relatively low, approximately 500 bytes per sphere, which means that even the largest scenes that can be handled only take up a few megabytes of device memory.

2.12.2 Using the engine

From a programmer's perspective the most important ways to interact with the engine are the interface functions that add objects and perform the different phases of the physics simulation step. These were already described above. It's worth noting that the internally used unit of time is the second, and the unit of length is arbitrary. Therefore when specifying the velocity of an object for example, it must be given in *units/second*. The constant of gravity is given in (units/second)/second, etc. The values of the parameters are not checked in any way, so it's the programmer's job to make sure not to use nonsensical data, like negative values for mass or radius.

Many vectors are represented internally using a `float4` type, with the 4th field unused. An exception is the `w` field of the position vector of the sphere, which is used to specify whether the object is static or not.

Bibliography

[1] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 223–232. ACM, 1989.

[2] Erwin Coumans et al. Bullet.

[3] P.E. Gill, W. Murray, and M.H. Wright. *Numerical linear algebra and optimization*, volume 1. Perseus Books, 1991.

[4] S. Green. Cuda particles. *NVIDIA Whitepaper, November*, 2007.

[5] C. Hecker. Physics, part 3: Collision response. *Game Developer*, pages 11–18, 1997.

[6] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 181–ff. ACM, 1995.

[7] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *ACM Siggraph Computer Graphics*, volume 22, pages 289–298. ACM, 1988.

[8] I. Newton. Philosophiae naturalis principia mathematica (mathematical principles of natural philosophy). *London, July*, 1687.

[9] C. Nvidia. PhysX.

[10] C. Nvidia. NVIDIA CUDA C Programming Guide version 4.0. *NVIDIA Corporation*, 2011.

[11] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. *GPU gems*, 3:677–695, 2007.

[12] J.C. Platt and A.H. Barr. Constraints methods for flexible models. *ACM SIGGRAPH Computer Graphics*, 22(4):279–288, 1988.

[13] Russell Smith. Open Dynamics Engine.

[14] K. Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 1994.