CUDA-optimised Physics Engine

Jonathan Frawley
B.A. (Mod.) Computer Science
Final Year Project, March 2010
Supervisor: Dr. Michael Manzke

Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____ April 7, 2010

Jonathan Frawley

Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_____ April 7, 2010

Jonathan Frawley

Acknowledgements

I would like to thank my parents for supporting me throughout the decisions I make in life. I would also like to thank Aoife, my girlfriend, who has helped me through every aspect of life while doing this project.

I would also like to thank my supervisor Dr. Michael Manzke, who allowed me the freedom to do a project of my own choosing and who gave me invaluable advice at every stage of the project.

I hear and I forget. I see and I remember. I do and I understand.

Confucius

Contents

1 Introduction **2**
1.1 Project Definition 2

2 Background **3**
2.1 Physics Engines 3
2.2 GPGPU Programming 4

3 Design **7**
3.1 Engine Design 7
3.2 Naming Conventions 7
3.2.1 Conventions 8
3.3 Interchangeability of components 8
3.4 Accessor Functions 9
3.5 Rule of Three 11
3.6 Floating Point Precision 12
3.7 Demo Programs and Tests 12
3.8 Existing Solutions Used 13

4 Implementation **14**
4.1 Basic Physics Background 14
4.1.1 Rigid Body Kinematics 14
4.1.2 Newton's Laws of Linear Motion 15
4.1.3 D'Alembert's Principle for Linear Motion 16
4.1.4 Torque 16
4.1.5 Moments of Inertia 16
4.1.6 D'Alembert's Principle for Rotation 17
4.2 Linear Algebra 17
4.2.1 Vectors 17
4.2.2 Quaternions 18
4.2.3 Matrices 18
4.3 Overview of the Ridid Body Physics Engine 21
4.3.1 Force Generators 22
4.3.2 The Rigid Body Simulator 22
4.3.3 The Collision Detection Subsystem 22
4.3.4 The Collision Resolver Subsystem 22

4.4 Building the x86 Physics Engine . . . 23
4.4.1 Linear Algebra . . . 23
4.4.2 Rigid Bodies . . . 41
4.4.3 Force Generators . . . 48
4.4.4 Collision Detection Pipeline . . . 48
4.4.5 Collision Detection . . . 49
4.4.6 Collision Response . . . 52
4.4.7 Collision Detection System Review . . . 55
4.5 CUDA-optimised Physics Engine . . . 55
4.5.1 Collision Detection using CUDA . . . 56
4.5.2 CUDA implementation of Sphere-Sphere Collisions . . . 58
4.5.3 CUDA Configuration . . . 58
4.5.4 CUDA Kernel and Device Functions . . . 60

5 Analysis **65**
5.1 Test Machine . . . 65
5.2 Collision Detection Benchmarks . . . 65
5.3 Limitations . . . 66
5.4 Hybrid Approach . . . 67

6 Conclusion **69**
6.1 Knowledge Acquired . . . 69
6.2 Future Work . . . 69
6.3 Summary . . . 70

A Including Code **73**

Listings

3.1 Getter Example 1 . . . 9
3.2 Getter Example 2 . . . 10
3.3 Getter Example 3 . . . 11
3.4 jfPrecision.h . . . 12
4.1 jfVector3_x86 definition . . . 23
4.2 jfVector3_x86 magnitude method . . . 25
4.3 jfVector3_x86 addScaledVector method . . . 25
4.4 jfVector3_x86 crossProduct method . . . 25
4.5 jfVector3_x86 dotProduct method . . . 25
4.6 jfQuaternion_x86 definition . . . 26
4.7 jfQuaternion_x86 normalize method . . . 26
4.8 jfQuaternion_x86 operator* . . . 27
4.9 jfQuaternion_x86 rotateByVector method . . . 27
4.10 jfQuaternion_x86 addScaledVector method . . . 27
4.11 jfMatrix3_x86 definition . . . 28
4.12 jfMatrix3_x86 add method . . . 29
4.13 jfMatrix3_x86 multiply matrix method . . . 29
4.14 jfMatrix3_x86 multiply vector method . . . 30
4.15 jfMatrix3_x86 setInverse method . . . 30
4.16 jfMatrix3_x86 setTranspose method . . . 31
4.17 jfMatrix3_x86 setOrientation method . . . 32
4.18 jfMatrix4_x86 definition . . . 32
4.19 jfMatrix4_x86 multiply matrix method . . . 33
4.20 jfMatrix4_x86 multiply vector method . . . 35
4.21 jfMatrix4_x86 transformDirection method . . . 35
4.22 jfMatrix4_x86 getDeterminant and setInverse methods . . . 36
4.23 jfMatrix4_x86 transformInverse method . . . 39
4.24 jfMatrix4_x86 transformInverseDirection method . . . 39
4.25 jfMatrix4_x86 setOrientationAndPos method . . . 40
4.26 jfMatrix4_x86 getAxisVector method . . . 40
4.27 Member variables from jfRigidBody.h . . . 41
4.28 jfRigidBody_x86 definition . . . 43
4.29 jfRigidBody_x86 calculateDerivedData method . . . 44
4.30 jfRigidBody_x86 addForce method . . . 44
4.31 jfRigidBody_x86 clearAccumulators method . . . 44

4.32 jfRigidBody_x86 integrate method 45
4.33 jfRigidBody_x86 getPointInLocalSpace method 46
4.34 jfRigidBody_x86 calculateTransformMatrix method 47
4.35 jfForceGenerator definition 48
4.36 jfCollisionDetector_x86 sphereAndSphere method start 50
4.37 jfCollisionDetector_x86 sphereAndSphere method collision check . 51
4.38 jfCollisionDetector_x86 sphereAndSphere method setting contact normal 51
4.39 jfCollisionDetector_x86 sphereAndSphere method setting contact point 51
4.40 jfCollisionDetector_x86 sphereAndSphere method setting penetration depth 51
4.41 jfCollisionDetector_x86 sphereAndSphere method setting body data 52
4.42 jfCollisionDetector_x86 sphereAndSphere method adding contact . 52
4.43 jfContactResolver_x86 prepareContacts method 53
4.44 Pseudocode for penetration resolution algorithm 53
4.45 Pseudocode for velocity resolution algorithm 54
4.46 jfCollisionDetector_x86 sphereAndSphereBatch method 57
4.47 jfCollisionSphereStruct definition 58
4.48 jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionTiled function - start 58
4.49 jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionTiled function - device allocation and memory copy 59
4.50 jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionTiled function - kernel configuration and running 60
4.51 jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionTiled function - retrieving results from device 60
4.52 jfCollisionDetectorKernel_cuda.cu calculateContacts function - start 61
4.53 jfCollisionDetectorKernel_cuda.cu calculateContacts function - loading and checking a triangular tile of spheres 61
4.54 jfCollisionDetectorKernel_cuda.cu calculateContacts function - loading and checking square tiles of spheres 62
4.55 jfCollisionDetectorKernel_cuda.cu calculateContacts function - saving results to global memory 62
4.56 jfCollisionDetectorKernel_cuda.cu sphereSphereCollision device function - Validity Check 63
4.57 jfCollisionDetectorKernel_cuda.cu sphereSphereCollision device function - Saving contact information 64

List of Figures

2.1 Popularity of physics Engines, December 2009 4
2.2 Demonstration of the open-source ODE engine with many rigid bodies 4
2.3 Bandwidth increase with GPU relative to CPU in recent years . . 5
2.4 How CUDA is organised on a hardware level 6

3.1 Initial Abstract Factory class hierarchy (All methods and classes not shown for brevity) 9
3.2 Modified Abstract Factory class hierarchy (All methods and classes not shown for brevity) 10

4.1 Rigid body interpenetration example 23
4.2 Physics Engine Collision Detection Pipeline 55

5.1 Rendering Disabled, Collision Detection times 66
5.2 Rendering Disabled, Collision Detection times (Small numbers of Spheres) 67
5.3 Rendering Enabled, Collision Detection times 68

List of Tables

Abstract

Physics engines have become prevalent in modern games. A core part of most of these engines is simulating the interaction and motion of rigid bodies. A lot of work has been done in optimising physics engines to take advantage of specialised hardware. NVIDIA's CUDA framework enables us to use the GPU as a highly-parallel co-processor. This thesis presents a rigid body physics engine, which has been optimised using NVIDIA's CUDA framework.

The aim of this project was to assess the potential for speedup in a physics engine using CUDA. A CPU-bound rigid-body physics engine was developed in C++ and each component was examined for potential parallelism. A scheme for detecting collisions between spheres processed on the GPU using CUDA is presented. Two versions of the engine resulted, an entirely cpu-bound version, and a CUDA-optimised version. A comparison between both versions of the engine is then made, and an assessment of the potential for using CUDA for rigid body simulation is given.

CD Contents

src/jfpx/CMakeLists.txt is the CMake build file for building the "jfpx" libraries and demo applications.

src/CMake contains CMake files to assist with compiling the CUDA sources(Not part of project, included for convenience).

src/jfpx contains C++ and CUDA source code for the "jfpx" physics engine.

src/jfpx/x86 contains C++ source code for the x86-only "jfpx" physics engine.

src/jfpx/cuda contains C++ and CUDA source code for the CUDA-optimised "jfpx" physics engine.

src/jfClient contains C++ source code for the demo applications.

src/jfClient/jfBallistic contains C++ source code for the ballistic demo application.

src/jfClient/jfFlight contains C++ source code for the flightsim demo application.

src/jfClient/jfBoxesAndBalls/x86 contains C++ source code for the x86-only benchmarking application.

src/jfClient/jfBoxesAndBalls/cuda contains C++ source code for the CUDA-optimised benchmarking application.

src/jfClient/jfBoxesAndBalls/cuda contains C++ source code for the CUDA-optimised benchmarking application.

src/jfCamera contains C++ source code for a first-person camera class used by the benchmarking application.

src/jfEvent contains C++ source code for event management for the demo applications.

src/jfGraphics contains C++ source code for window management and 3D graphics using OpenGL used by the demo applications.

src/jfLog contains C++ source code for simple logging.

src/jfShape contains C++ source code for shapes which have both graphical and physical properties, used by the demo applications.

src/jfSimulation contains C++ source code for simulation interface for the demo applications.

src/jfTimer contains C++ source code for a millisecond-accurate timer and a microsecond-accurate timer.

bin/unix contains demo program executables compiled for x86_64 linux 2.6.32.

bin/unix contains demo program executables compiled for Windows XP 64-bit.

test contains UnitTest++ unit tests for certain parts of the engine.

lib contains 3rd party libraries used by the project.

doc contains the report files, as well as result files for the benchmarks run.

media contains textures used in the demo programs.

build contains legacy codeblocks IDE build files. Use Cmake method instead for compilation.

Chapter 1

Introduction

This project attempts to find a basis for integrating the CUDA framework into a physics engine. The project was chosen due to the author's personal interest in the subject of computer game design, and the design of real-time physics for modern day computer games. The report is organised as follows:

- 1 is a general introduction to the project.
- 2 give a background to the area, including developments in games physics and in the realm of GPGPU programming.
- 3 is a general look at the design of the system and what decisions had to be made in the design process.
- 4 looks in detail at how the x86-only and CUDA optimised engines were constructed
- 5 looks at the results from testing the two versions of the engine and performing a comparatve study.
- 6 gives the conclusions which were arrived at from the project.

1.1 Project Definition

The project itself looks at the problem of simulating real world physics using NVIDIA's CUDA framework. It looks at the difference between CPU and GPU-optimised versions of a particular aspect of the engine. The report details the integration of the CUDA-optimised system into the x86-only system and the problems involved in doing so. It also reaches some conclusions based on benchmarking as to the benefits of each system and in what situation a CUDA-optimised version is more optimal than a CPU-version. Finally speculation is made about how other parts of the engine might be optimised using CUDA how performance could be improved.

Chapter 2

Background

2.1 Physics Engines

As games become more and more popular, the expectations for realism in the environments increases. Water, planes, ballistics and everyday objects are expected by the gamer to look and act as they do in the real world. However, attempts to make more and more realistic-looking scenes leads to an increase in demand for processing power. Physical simulation refers to the simulation of physical objects using the using laws of motion. Physics engines which simulate the motion of rigid bodies using rigid body kinematics are known as rigid body physics engines. They are used in several fields, including engineering, medical science, robotics and computer game design. In most of the fields, we are not performing the calculations in real-time, and the accuracy of the simulation is what is paramount. However in other fields, we are performing the calculations in real-time, so tradeoffs have to be taken to be able to process all of the physics needed inside the game loop. One of these is the area of computer game physical simulation.

Game physics engines provide an approximate method simulating physics, based on a tradeoff between what looks and acts reasonably realistically and what impacts performance heavily. These simulations have to be fast enough so as not to affect the framerate of the game unfavorably. The first modern physics engine designed was a product called *MathEngine* which was designed in 1997 by MathEngine PLC. [13, p. 4] [12] Since then, many commercial and open-source physics engines have been developed. One that has gained a certain amount of noteriety is the Havok engine, a company founded by Stephen Collins. It appers however that after many years of Havok being the dominant physics engine on the market, NVIDIA's *PhysX* engine has now become the most popular for games released in 2009 according to [24] as shown in 2.1. The main open-source engine for now and for many years has been ODE [5] although Bullet [2] has recently also become very popular. Recently there has been increased focus on hardware-optimised physics simulations. Ageia released the first Physics Processing Unit(PPU) to muted reception. It was released in conjunction with their PhysX engine, which offloaded some of the computations to the PPU for

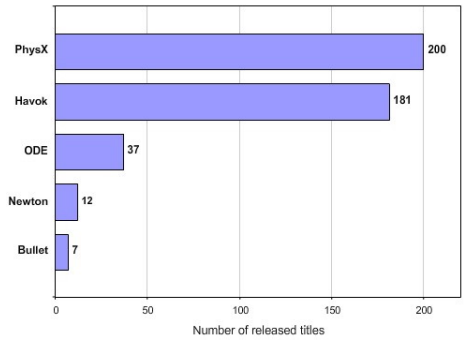


Figure 2.1: Popularity of physics Engines, December 2009

processing. In February 2008 NVIDIA bought Ageia technologies and now the PhysX engine uses the GPU for optimisations rather than the PPU. Other engines are beginning to look at hardware optimisations using the GPU. Havok FX saw Havok's engine use the GPU to offload some of the processing of their engine to the GPU. Bullet have recently added CUDA support for their broadphase collision detection as seen at [3].

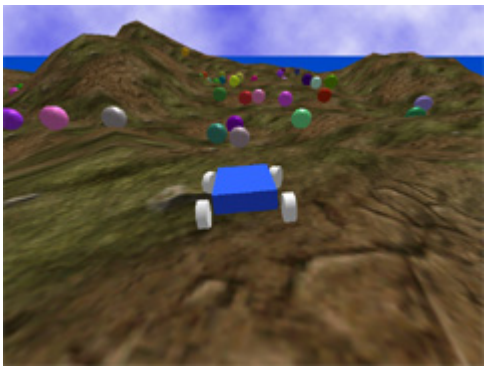


Figure 2.2: Demonstration of the open-source ODE engine with many rigid bodies

2.2 GPGPU Programming

This is an emerging field which is currently being led by NVIDIA[11] and ATI[6]. An example of the potential of the GPU in performance is shown in 2.3.

GPGPU programming was in the past, quite a difficult, low-level task which could not be scaled easily. For many years, programmers utlised the power offered by the GPU using programs written in "Shader" languages. These languages were designed to provide graphical effects and in order to perform general pur-

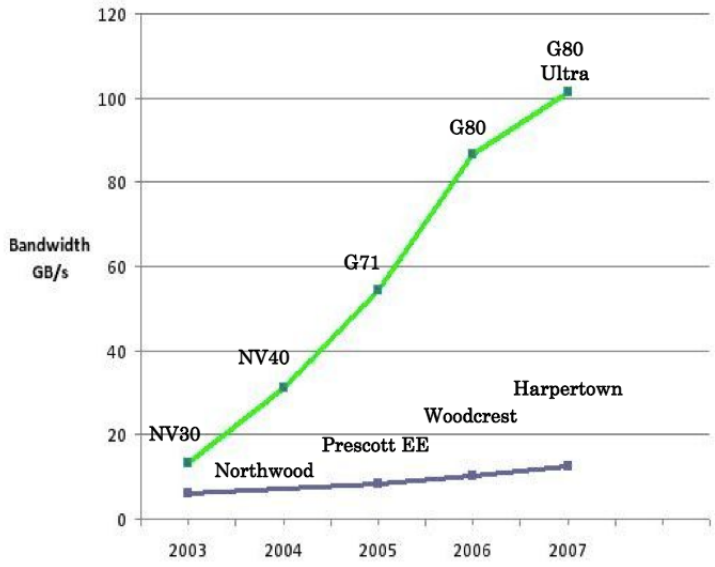


Figure 2.3: Bandwidth increase with GPU relative to CPU in recent years

pose computation with them, different tricks have to be used. With the advent of CUDA by NVIDIA, GPGPU programming can be used in an easily scalable fashion from the C programming language. CUDA contains both a low-level and higher level API, with the lower level API giving more fine-grained control over the system. [10] CUDA is a highly-distributed model for computation. [10, p. 77] describes CUDA as a *SIMT* architecture, which allows hundreds of threads to be run in parallel, each with unique flow control. It therefore is not suitable for the all of the same operations as vector SIMD architectures such as Intel's SSE instruction set [8] which only allows the same instruction to operate on multiple data sets. OpenCL[22] is a cross-vendor framework for writing GPGPU programs which is becoming increasingly popular. While CUDA programs can only run on NVIDIA GPUs, OpenCL programs can be run on either NVIDIA or ATI GPUs, giving more flexibility in terms of hardware. CUDA is organised as pictured in 2.4. With the limits of instruction level parallelism limiting the amount of instructions which can be executed concurrently, moves are being made industry-wide to a more parallel approach. GPUs allow programmers to utilise hundreds of cores and take advantage of the parallelism available in many algorithms. Programmers have fine-grained control over synchronisation and have numerous choices for memory access patterns. This gives the flexibility needed to be very useful in scalable, highly parallel systems. It gives such an increase in performance, it is being used to process the large amounts of data being output by the large hadron collider [1]
Intel has announced a new platform for GPGPU programming, similar to CUDA,

called Larabee [9]. It seems that the industry is heading towards a unified aim : to exploit the huge potential in processing power afforded by graphics cards in a scalable fashion.

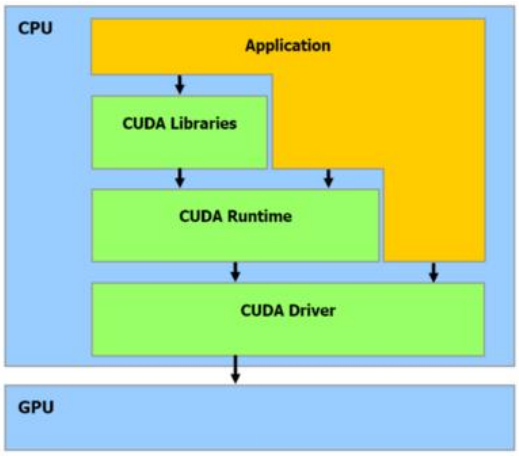


Figure 2.4: How CUDA is organised on a hardware level

Chapter 3

Design

This chapter presents the design of both the CPU-bound and CUDA-optimised components of the physics engine. It is also details the general design of the demo applications which were used to test the engine and are available on the CD. A number of design criteria were decided upon early on in the project :

- The CUDA and x86-only components of the engine need to be interchangeable.
- The design must be free of memory leaks.
- The design should be based on an existing engine.

3.1 Engine Design

The design of the CPU-bound engine is based on the Cyclone engine [20]. The decision to use this engine was made due to the completeness of the documentation for the engine and its simple, and extendable design. Other engines looked at were ODE and Bullet, but both presented very complex codebases and it was difficult to see how one could integrate in components into their designs.

3.2 Naming Conventions

The physics engine designed for this project was named "jfpx", after the author. It is for this reason that all type names in the engine are prefixed with "jf". This is used to avoid problems where two types from different frameworks define different types with the same name. An example would be the "real" data type employed by many frameworks such as "Octave"[4]. This problem was encountered during the early stages of the project when the "jfReal" member of "jfpx" was named "real." As unit tests which used "Octave" were written, there was a conflict as "Octave" defines a "real" type as well. This naming convention removes ambiguity with type names.
Another naming convention used is to specify the architecture with which the

class is implemented for. Classes which are implemented for the CPU only are suffixed with "_x86," referring to the x86 CPU architecture. Classes which have been optimised with CUDA are suffixed with "_cuda." Classes which have neither suffix are either abstract or have no specific requirements for architecture.

3.2.1 Conventions

Class members are all prefixed with "m_" to signify that they belong to the class and will be deallocated when the class is deleted. Header files act as a form of documentation for the engine. They are split up into relevant sections using comments, seperating out the "getters and setters" section from the methods which have been inherited and the unique elements of the class. This kind of documentation helps the user get a feel for how the engine is to be used without needing to look at the implementation files.

3.3 Interchangeability of components

Cyclone only uses only the CPU for computation, and uses a flat hierarchy for classes. For the purposes of this project, the framework needed to be designed so that CUDA and CPU components could be used interchangeably. This was so that when it came to evaluation, the components could easily be tested against each other. It also gives a high degree of flexibility, so that CUDA and CPU components can be combined dynamically at runtime.

The object-oriented Abstract Factory pattern [15, p. 87] was used to satisy this design criteria. This pattern allows for different families of objects to be created, members of which satisfy a common interface. In the case of the "jfpx" engine, the two families of objects to be created are x86-only and CUDA-optimised components. An abstract factory class specifies the construction of objects of the physics engine by clients. This greatly simplifies the use of either x86 or CUDA versions of the engine by clients, they only need to change their instantiation of the factory class.

My initial design of the abstract factory class hierarchy looked similar to 3.1. Italics indicates that the class or method is abstract as in [14, p. 70].

It became apparant throughout the course of the project that some of the classes would not benefit from being optimised by CUDA. To reduce the effort of duplicating code for the CUDA version of the engine, I modified the hierarchy so that the CUDA factory and CUDA products inherit from the x86 versions.

As can be seen from 3.2, the cuda factory in fact inherits from the x86 factory. This enables us to only implement classes we want to optimise and defer all of the rest to the x86 versions of the classes. Similarly the "jfCollisionDetector_cuda" class inherits from the x86 version "jfCollisionDetector_x86", and only overrides the functions it wishes to optimise. Since all of the functions in these classes are virtual, the appropriate method will be called at runtime.
For example, if the client calls a method "boxBoxCollision' on a "jfCollisionDetector_cuda"' object, which is implemented in "jfCollisionDetector_x86" but not in "jfCollisionDetector_cuda" (which is the case in the jfpx engine), the x86's

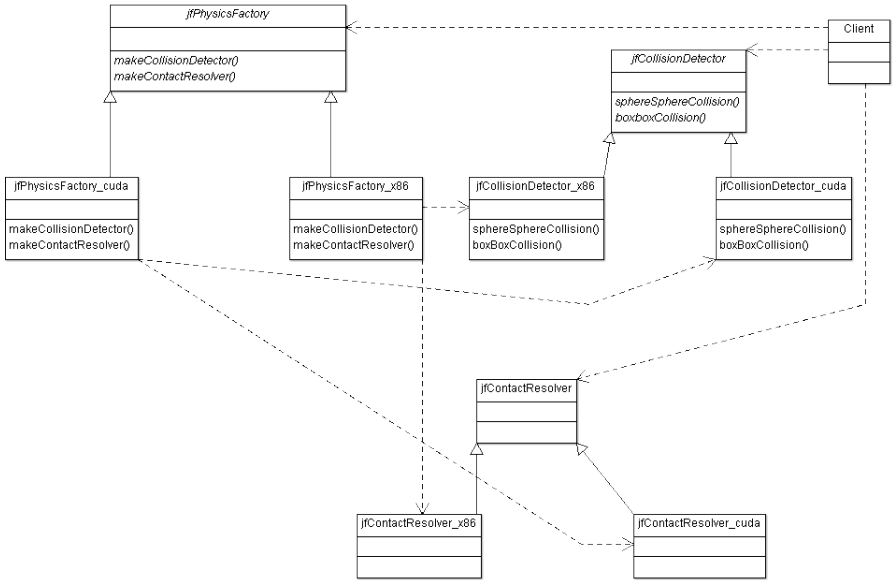


Figure 3.1: Initial Abstract Factory class hierarchy (All methods and classes not shown for brevity)

version of the function will be called.
Furthermore, if the client calls the function "sphereSphereCollision" on the same "jfCollisionDetector_cuda" object, the cuda-optimised method will be called. While using virtual methods imposes a small cost at runtime, the benefits of such a flexible and extendible design outweigh the potential performance costs. This framework lends itself to being highly extendable. For instance, it would be easy to integrate another family of objects, for instance a CELL processor-optimised version of the engine. An engine composed of various architectures could then be formed.

3.4 Accessor Functions

A word should be said about the accessor functions used for the framework.
Accessor functions in this context are functions which are prefixed with "get" and give access to the protected/private members of a class.
I will refer to such functions as "getter" functions from now on.
Traditionally, getter functions are of the form :

Listing 3.1: Getter Example 1

```
1std::string myObject::getName() const
```

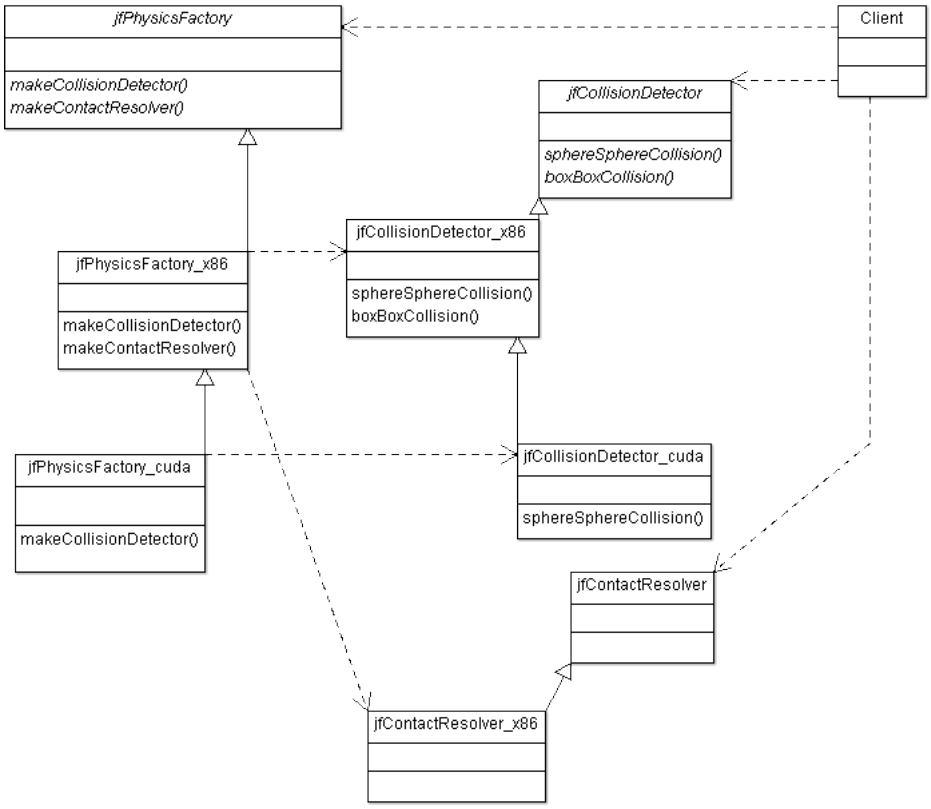


Figure 3.2: Modified Abstract Factory class hierarchy (All methods and classes not shown for brevity)

```
2{
3    return m_Name;
4}
```

Which is acceptable for return types which are not abstract. However in C++, we cannot define functions which return an abstract type. We can however return a pointer to an abstract type. The solution then becomes:

Listing 3.2: Getter Example 2

```
1private:
2    myAbstractClass* m_AbstractObject;
3...
4myAbstractClass* myClass::getAbstractObject() const
5{
6    return m_AbstractObject;
```

```
7}
```

The problem then is that you are giving the caller access to the private member of the class. If the caller decides to delete the memory, thinking that the memory was newly allocated by the function, it will delete the private member of "myClass". Likewise, if the caller suspects that the memory hasn't been newly allocated by the getter function, and it does not delete the object, we get a memory leak.
Also, the caller could easily change the private value of "myClass" without "myClass" knowing about it.
The solution is to assume that the caller will allocate the memory for us prior to calling the function. It passes in a pointer to the object it allocated and this will then be set to the same value of the private member. The code then becomes:

Listing 3.3: Getter Example 3

```
1private:
2    myAbstractClass* m_AbstractObject;
3...
4void myClass::getAbstractObject(myAbstractClass* result) const
5{
6    (*result) = (*m_AbstractObject);
7}
```

This requires that the "operator=" function is defined for the abstract class. This removes any ambiguity over who allocated what memory - the caller is responsible for allocation and deallocation of memory. As a convention, this parameter is usually called "result," unless a more suitable name is available.

3.5 Rule of Three

The "Rule of three" to which I refer to in the prototype for the class is a general rule of thumb for designing C++ classes which states that if you define any one of:

- A destructor
- A copy constructor
- "operator=" method

then you should define them all. [19] This rule was applied where it was deemed appropriate for some core classes such as the "jfVector3" class which is used very often throughout the engine.

3.6 Floating Point Precision

As suggested in [20], it was decided that a new type "jfReal" would be added so that it would be easy to change between single and double precision floating point numbers. This also involved giving unique names for standard math functions for calculating the square root of a number, the power of a number and the absolute value of a number so that they could be changed to single or double precision at will. The maximum value for a number of the "jfReal" type is also given.
It is defined in "jfPrecision.h":

Listing 3.4: jfPrecision.h

```
1#ifndef JFPRECISION_H
2#define JFPRECISION_H
3
4#include <math.h>
5#include <float.h>
6
7//Double or single precision
8typedef float jfReal;
9#define jfRealSqrt(x) sqrtf(x)
10#define jfRealPow(x,y) powf(x,y)
11#define jfRealAbs(x) fabs(x)
12
13const float JF_REAL_MAX = FLT_MAX;
14
15#endif
```

3.7 Demo Programs and Tests

Demo programs were created at various stages of the project and show various features of the engine. The first two demos are modified from [20] but the last demo was created from scratch. The "jfBallistic" demo is a simple demonstration of how integration of a simple particle can be used to generate physical effects. The "jfFlight" demo demonstrates force generators in action The "jfBoxesAndBalls" demo shows almost all of the features of the engine in action. A first-person view allows the user to navigate using the "W", "S", "A" and "D" keys and the mouse. Arbitrary amounts of boxes and balls can be added to the program by editing the "jfBoxesAndBallsSimulation_$ARCH.cpp" where $ARCH is either x86 or cuda.

The whole codebase was made to be as cross-platform as possible but has only been tested on Windows XP 64-bit and Arch-linux 2010 64-bit. SDL was used for window management and OpenGL was used for 3D rendering. CMake was used for compilation of the demo files in a cross platform fashion. In fact, this compiles the "jfpx" engine into a shared library and compiles the example programs. On windows it will generate Visual Studio files, on unix it will generate Makefiles, etc. See [17] for more details. Codeblocks was used earlier on in the project and codeblocks projects are available on the CD. Unit tests were designed for a subset of the core classes of the engine to verify functionality. These were written using the Unittest++ framework. [25]

3.8 Existing Solutions Used

The engine is designed from the ground up based on the method put forth by [20] and has been modified from the Cyclone engine by the same author. The CUDA-optimised portion of the engine was designed based mostly on work from [23] and [16]. The implementation of narrow-phase collision detection using CUDA is unique from what I have researched. The implementation presented in this report can also be adapted to other areas as will be described in 6.

Chapter 4

Implementation

This chapter will describe the background needed to understand how the physics engine works. It will then go on to describe how the core math classes were implemented. Then we will see how the collision detection pipeline works and finally the CUDA optimisations are explained in detail.

4.1 Basic Physics Background

In this section, we will look at some basic concepts from physics that are relevant to the analysis of motion and interaction of rigid bodies for the construction of a rigid body physics engine. A rigid body is classified a type of region which contains a mass. Rigid bodies differ in how that mass is distributed.

4.1.1 Rigid Body Kinematics

Kinematics is the study of the motion of objects without taking into account the effects of external forces. Let us first consider the motion of particles in 3 dimensions (spatial coordinates).
The position of the particle at time t r(t) according to [13] is given by:

$$r(t) = x(t)i + y(t)j + z(t)k \tag{4.1}$$

where i = (1,0,0), j = (0,1,0) and k = (0,0,1).
The velocity of a particle at time t v(t) is given according to [13] by:

$$v(t) = \dot{r} \tag{4.2}$$

where the dot symbol refers to differentiation with respect to t.
The acceleration of the particle at time t, at is given according to [13] by:

$$a(t) = \dot{v} \tag{4.3}$$

It should be noted that this shows that we can convert between either position, velocity and acceleration using only differentiation or integration.

4.1.2 Newton's Laws of Linear Motion

Now we consider objects which have external forces acting upon them. Getting to know Newton's laws of linear motion are core to the understanding of basic mechanics - vital to a physics engine.
The following definitions have been consulted from [13]. One key concept is the amount of *inertia* of an object. Another is an object's *mass*.

Definition Inertia is the tendancy of an object to stay at rest.

Definition Mass is the measure of the inertia of an object. Its unit of force is the *kilogram*.

Force is the main concept Newton uses in his equations.

Definition Force is the changing of the mechanical state of an object.

Definition Newton's Second Law:
For an object of constant mass, its acceleration a is proportional to the force F and inversely proportianl to the mass m of the object. Force can be described by the relationship:

$$F = ma \tag{4.4}$$

where :
F is the force in Newtons
m is the mass of the object in kilogrammes
a is the acceleration of the object in ms^2

Although the above form is correct, for our purposes it is more convenient to think of force in another form:

$$F = \frac{d}{dt}(mv) = ma + \frac{dm}{dt}v \tag{4.5}$$

where:
v is the velocity of the object.

Now a definition of momentum is in order :

Definition The linear momentum of an object is described by the mass of the object multiplied by its velocity.

$$p = mv \tag{4.6}$$

where:
p is the linear momentum of the object.

Thus, we can look at Newton's second law in another way, as causing a change in an object's momentum over time.

Definition Newton's third law:
When a force is exerted on an object, an opposing force of equal magnitude and opposite direction is exerted on some other body which interacts with it.

4.1.3 D'Alembert's Principle for Linear Motion

D'Alembert's Principle states that if we have a set of forces acting on a body, we can replace all of the forces with a single force, defined as:

$$F = \sum_{i=0}^{n} f_i \tag{4.7}$$

Where F is the resultant force on the body. [20, p. 70] Using this, we can accumulate all of the forces acting on a body, and apply them all at once. This is what is done in the physics engine, with the application of the forces being done during the integration stage of the physics loop.

4.1.4 Torque

The turning force of a body is known as "torque." This is also known as "moment of force." [20] The torque for a point on a body, τ, can be expressed as :

$$\tau = r \times f \tag{4.8}$$

where:
r is the position of the point relative to the origin of the object.
f is the force applied at the point.

We can see this law in action in everyday life, for example when opening a door, it is much easier to open the door at the edge furthest away from the hinge. This is due to the torque being proportional to the distance from the origin of the object, in this case the hinge of the door.
Every force applied to a body will generate a corresponding torque, which will in turn rotate the object.
In the engine, torques are given in a scaled-axis representation similar to in [20, p. 197]:

$$\tau = a \cdot \hat{d} \tag{4.9}$$

4.1.5 Moments of Inertia

Just as mass is the measure of the body's resistance to linear motion, the *moment of inertia* of an object resists angular motion.

Definition The moment of inertia of an object is the measure of how difficult it is to change an object's rotational speed.

In the physics engine presented, the moment of inertia is represented by a 3×3 matrix known as the *inertia tensor* as in [20, p. 199]. On the diagonal of the matrix, I, the moments of inertia for each axis is represented:

$$\begin{bmatrix} I_x & & \\ & I_y & \\ & & I_z \end{bmatrix}$$

where I_x, I_y and I_z are the moments of inertia about the x,y and z axes through the centre of mass of the object.

The inertia tensor can now be used to come up with a rotational version of Newton's second law of motion [20, p. 200] :

$$\tau = I^{-1}\ddot{\theta} \tag{4.10}$$

or written in another way, to get the angular acceleration:

$$\ddot{\theta} = I^{-1}\tau \tag{4.11}$$

For this equation, we store the inertia tensor as its inverse in the *jfpx* engine.

4.1.6 D'Alembert's Principle for Rotation

The same principle which allowed us to compress all of the forces acting on a body into a single force acting on the body, allows us to combine all of the torques into a single torque acting on the body. The relevant formula from [20, p. 205]:

$$\tau = \sum_{i=0}^{n} \tau_i \tag{4.12}$$

where τ_i is the i^{th} torque and where n is the total number of torques appied to the body.

4.2 Linear Algebra

This section explains the theory for basic linear algebra, core to the operation of the physics engine.

4.2.1 Vectors

A vector is a quantity which has both magnitude and direction. A 3-dimensional vector has direction in 3 dimensions, x, y and z. The magnitude of a vector is its scaler length. The *dot product* is defined as :

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z \tag{4.13}$$

and it is mostly used in the physics engine when finding out the magnitude of one vector in the direction of another. [20]
The *cross product* is defined as :

$$a \times b = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix} \tag{4.14}$$

and it is useful in the calculation of mutually orthogonal vectors. [20]

4.2.2 Quaternions

Quaternions are numbers of the form (w + xi + yj + yk) where w, x, y and z are real numbers.[13]

Normalization

Rotations of unit-length are related to rotations of unit length. As our main use for quaternions in game physics is to rotate bodies, getting unit-length quaternions is important. Normalization involves ensuring that the quaternion is of unit length. In other words, that it satisfies the following formula from [20, p. 157] :

$$\sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{4.15}$$

Multipication

Multipication of two quaternions is defined as in [13, p. 512] :

$$\begin{aligned}(w_0 + x_0 i + y_0 j + z_0 k) \times (w_1 + x_1 i + y_1 j + z_1 k) = \\ (w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) + \\ (w_0 x_1 + x_0 w_1 - y_0 z_1 - z_0 y_1) i + \\ (w_0 y_1 - w_1 y_0 + z_0 x_1 - x_0 z_1) j + \\ (w_0 z_1 + w_1 z_0 + x_0 y_1 - y_0 x_1) k\end{aligned} \tag{4.16}$$

4.2.3 Matrices

Matrices are very important in computer graphics, and equally important in physical simulation. Matrices are used in the engine for a variety of purposes. Some of these are highlighted below:

- For transforming vectors between coordinate systems
- For holding the inertia tensor of a rigid body

Transformation Matrices

A set of 3 axes is called a "basis" or a set of axes. Take a matrix A :

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{4.17}$$

as being made up of three vectors:

$$\begin{bmatrix} a \\ d \\ g \end{bmatrix}, \begin{bmatrix} b \\ e \\ h \end{bmatrix} and \begin{bmatrix} c \\ f \\ i \end{bmatrix} \tag{4.18}$$

Then if each vector represents an axis, all three vectors make up a basis. Now, if we multiply a vector by this matrix, the result is the vector transformed into new

basis. [20]

Something which is often needed in physics engines, is to transform a point from one coordinate to another, and then get that point relative to a new origin. We could do this manually, by applying a 3x3 transform matrix to the point and then adding the new origin to the resultant point. However if the transform matrix is represented as a 4x4 matrix:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

where: $\begin{bmatrix} d \\ h \\ l \end{bmatrix}$ is the origin of the new coordinate system. Using this, we can perform the transform-and-add operation using standard matrix multipication:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix} \tag{4.20}$$

This is of use as there is the concept of object coordinates and world coordinates in the engine. Object coordinates are relative to the origin of the rigid body and world coordinates are relative to the origin of the world. It will be necessary to convert into and out of object coordinates throughout the engine. To convert in the opposite direction, what is needed is to multiply the vector by the inverse of the transform matrix:

$$A^{-1} v_{body} = v_{world} \tag{4.21}$$

where:
v_{body} is the vector in body coordinates.
v_{world} is the vector in world coordinates.
A^{-1} is the inverse of the transform matrix described above.

Matrix Addition

Addition of matrices is simple, and can only be done with matrices of equal size. We simply add the components of the two $(M \times N)$ matrix together:

$$\begin{bmatrix} a_{11} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & a_{mn} + bmn \end{bmatrix} \tag{4.22}$$

Matrix Multipication

Since our engine will only use 3×3 and 4×4 matrices matrices, only these matrices will be considered.

To multiply a 3×3 matrix with another 3×3 matrix, the following formula suffices:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} (a_{11} * b_{11}) + (a_{12} * b_{21}) + (a_{13} * b_{31}) & (a_{11} * b_{12}) + (a_{12} * b_{22}) + (a_{13} * b_{32}) & (a_{11} * b_{13}) + (a_{12} * b_{23}) + (a_{13} * b_{33}) \\ (a_{21} * b_{11}) + (a_{22} * b_{21}) + (a_{23} * b_{31}) & (a_{21} * b_{12}) + (a_{22} * b_{22}) + (a_{13} * b_{32}) & (a_{21} * b_{13}) + (a_{22} * b_{23}) + (a_{23} * b_{33}) \\ (a_{31} * b_{11}) + (a_{32} * b_{21}) + (a_{33} * b_{31}) & (a_{31} * b_{12}) + (a_{32} * b_{22}) + (a_{13} * b_{32}) & (a_{31} * b_{13}) + (a_{32} * b_{23}) + (a_{33} * b_{33}) \end{bmatrix} \tag{4.23}$$

This can be extended to the multipication of a 4×4 matrix easily:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} (a_{11} * b_{11}) + (a_{12} * b_{21}) + (a_{13} * b_{31}) + (a_{14} * b_{41}) & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \cdots & \cdots & (a_{31} * b_{13}) + (a_{32} * b_{23}) + (a_{33} * b_{33} + (a_{44} * b_{44}) \end{bmatrix} \tag{4.24}$$

Determinant Of A Matrix

The determinant of a matrix is a scalar quantity, which is used in the calculations of inverses of matrices.
For example, the determinant of a 3×3 matrix is as defined by [13] :

$$det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33} - a_{11}a_{23}a_{32} \tag{4.25}$$

The determinant of a 4×4 matrix is similarly defined as done in [20, p. 175].

Inverse Of A Matrix

Definition A matrix A^{-1} is said to be the *inverse* of the matrix A if:
$A \cdot A^{-1} = I$
where I is the identity matrix.

The calculation of inverses is very important for real time simulations. As said previously, each rigid body has a transformation matrix to convert vectors from local coordinates to world coordinates. Sometimes it is useful to do the reverse transformation, convert a vector from world to local coordinates. In this case we can us the inverse of the matrix in place of the original matrix to perform the transformation.
Also, the inertia tensor is usually stored as its inverse, so this has to be calculated as well.

The inverse of A, a 3×3 matrix, as described in [13, p. 637] and [20, p. 173] is:

$$A^{-1} = 1/(det(A)) \begin{bmatrix} a_{22}a_{33} - a_{23}a_{32} & a_{13}a_{32} - a_{12}a_{33} & a_{12}a_{23} - a_{13}a_{22} \\ a_{23}a_{31} - a_{21}a_{33} & a_{11}a_{33} - a_{13}a_{31} & a_{13}a_{21} - a_{11}a_{23} \\ a_{21}a_{32} - a_{22}a_{31} & a_{12}a_{31} - a_{11}a_{32} & a_{11}a_{22} - a_{12}a_{21} \end{bmatrix} \quad (4.26)$$

The inverse of a 4×4 matrix can be gotten using Cramer's rule as in [7].

Matrix Transpose

As was seen from 4.26, inverting a matrix is a complex task. In certain situations, we can perform another operation, known as the matrix transpose instead.

Definition The *transpose* of a matrix A, is the result of swapping the rows and column numbers for each element of a matrix.
That is:

$$A_{i,j} = A_{j,i}$$

for each element in the matrix, where i and j are the row and column number respectively of the element.

When a matrix represents a transformation, and that transformation represents only a rotation with no translation, the transpose of a matrix is equal to the inverse of the matrix.

$$M^T = M^{-1} \quad (4.27)$$

, only when M represents a transformation with rotation and no translation.

4.3 Overview of the Ridid Body Physics Engine

The physics engine developed can detect collisions between rigid bodies, resolve those collisions and simulate the motion of those bodies using the laws of motion. There are four main parts to the engine:

- The force generators
- The rigid body simulator
- The collision detection subsystem

- The collision resolver subsystem

The flow of the physics engine in a game loop can be described as:

1. The force and torque generators apply specific forces and torques to the body, based on the state of the world.
2. The rigid body simulator integrates the force and torque to get a position and velocity for the body.
3. The collision detection subsystem generates contacts if the body collides with other bodies or planes.
4. The collision resolution system resolves these contacts and writes a new position and velocity to the body based on the contacts.

Each part of the engine will now be described in detail.

4.3.1 Force Generators

Force generators apply forces and torques to the bodies of the world based on the current state of the game. For example, gravity uniformly applies an acceleration of 9.81N in the negative Y-axis. Another example is the aero control force generator developed as part of the flight simulation demo. This provides a lifting force on the wings of the aircraft when we want to turn.

4.3.2 The Rigid Body Simulator

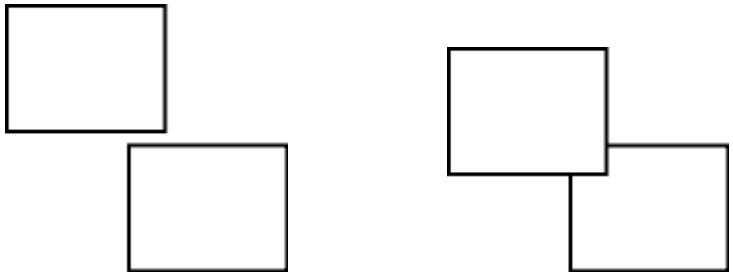
The rigid body simulator's responsibility is to integrate the forces and torques applied to the body based on the laws of motion discussed in 4.1.1 and 4.1.2. As we saw in 4.1.1, from an acceleration we can get a position by integration. And as we saw in 4.1.2, we can get the acceleration of a body from the forces acting on it. We use the forces acting on the body to get the current position of the body with respect to the simulation.

4.3.3 The Collision Detection Subsystem

The responsibility of the collision detection subsystem is to detect whether objects are colliding. If bodies are colliding, we need to generate contacts to encapsulate the information about the collision so that the collision resolver subsystem can resolve the collision appropriately.

4.3.4 The Collision Resolver Subsystem

The collision resolver subsystem is responsible for resolving the contacts generated by the collision detection subsystem. Potentially, the contacts generated by the collision detection subsystem will represent interpenetrations as well as collisions. This is due to the fact that we are looking at the positions of bodies at discrete moments in time. If bodies are not colliding at time t as in 4.1, they may be interpenetrating at time $(t + \alpha)$.



(a) Rigid bodies before collision

(b) Rigid bodies after collision with interpenetration

Figure 4.1: Rigid body interpenetration example

4.4 Building the x86 Physics Engine

This section describes the construction of the full x86 physics engine from scratch. It leads into the next section where the optimisations are made to the engine using CUDA.

4.4.1 Linear Algebra

For the physics engine, I implemented classes for linear algebra including a 3-Dimensional Vector class, a 3x3 matrix class and a 4x4 matrix class.

These classes are core to the functionality of the engine, and by implementing them by hand I was able to see if potential optimisations were available.
Unit tests were written for these classes and are available on the CD. These were made to ensure correct operation of the core math functions on which the engine is based.

3-Dimensional Vector class

The 3D vector class is based on an existing implementation [20] and made to be more object-oriented and to inherit from a common interface as described in 3.1. The class definition looks like:

Listing 4.1: jfVector3_x86 definition

```
1class jfVector3_x86 : public jfVector3
2{
3    public:
4        jfVector3_x86();
5
6        jfVector3_x86(const jfReal x, const jfReal y, const jfReal z)
             ;
7
8        jfVector3_x86(const jfVector3& other);
9
10       jfVector3_x86(const jfVector3Struct& other)
```

```
11          {
12              m_X = other.m_X;
13              m_Y = other.m_Y;
14              m_Z = other.m_Z;
15          }
16
17          jfVector3_x86& operator=(const jfVector3& other);
18
19          virtual jfVector3_x86* clone() const
20          {
21              return new jfVector3_x86(*this);
22          }
23
24          virtual ~jfVector3_x86();
25
26          virtual void invert();
27
28          virtual jfReal magnitude() const;
29
30          virtual jfReal squareMagnitude() const;
31
32          virtual void normalize();
33
34          virtual void operator*=(jfReal value);
35
36          virtual void operator+=(const jfVector3& v);
37
38          virtual void operator-=(const jfVector3& v);
39
40          virtual void addScaledVector(const jfVector3& v, jfReal scale
                );
41
42          virtual void componentProductUpdate(const jfVector3& v);
43
44          virtual jfReal dotProduct(const jfVector3& v) const;
45
46          virtual void multiply(jfReal val, jfVector3* result) const;
47
48          virtual void add(const jfVector3& val, jfVector3* result)
                const;
49
50          virtual void subtract(const jfVector3& val, jfVector3* result
                ) const;
51
52          virtual void subtract(jfReal val, jfVector3* result) const;
53
54          virtual void componentProduct(const jfVector3& val, jfVector3
                * result) const;
55
56          virtual void crossProduct(const jfVector3& vec, jfVector3*
                result) const;
57
58      private:
59  };
```

The magnitude of a vector is implemented as the square root of the addition of the squares of all of the vector's components :

Listing 4.2: jfVector3_x86 magnitude method

```
1jfReal jfVector3_x86::magnitude() const
2{
3    return jfRealSqrt((m_X*m_X) + (m_Y*m_Y) + (m_Z*m_Z));
4}
```

The "addScaledVector" function is used during integration of rigid bodies and thus is of vital importance to the engine.

Listing 4.3: jfVector3_x86 addScaledVector method

```
1void jfVector3_x86::addScaledVector(const jfVector3& v, jfReal scale)
2{
3    m_X += (v.getX() * scale);
4    m_Y += (v.getY() * scale);
5    m_Z += (v.getZ() * scale);
6}
```

Cross product implementation is as described in 4.14:

Listing 4.4: jfVector3_x86 crossProduct method

```
1void jfVector3_x86::crossProduct(const jfVector3& vec, jfVector3*
     result) const
2{
3    jfVector3_x86 tempResult;
4    tempResult.setX((m_Y*vec.getZ()) - (m_Z*vec.getY()));
5    tempResult.setY((m_Z*vec.getX()) - (m_X*vec.getZ()));
6    tempResult.setZ((m_X*vec.getY()) - (m_Y*vec.getX()));
7    (*result) = tempResult;
8}
```

The dot product implementation is as described in 4.13:

Listing 4.5: jfVector3_x86 dotProduct method

```
1jfReal jfVector3_x86::dotProduct(const jfVector3& v) const
2{
3    return ( (v.getX()*m_X) + (v.getY()*m_Y) + (v.getZ()*m_Z) );
4}
```

Quaternions

The Quaternion class in the engine is used for holding the orientation of a rigid body. The definition of the Quaternion class follows:

Listing 4.6: jfQuaternion_x86 definition

```
1class jfQuaternion_x86 : public jfQuaternion
2{
3    public:
4        jfQuaternion_x86();
5
6        jfQuaternion_x86(jfReal r
7                         ,jfReal i
8                         ,jfReal j
9                         ,jfReal k);
10
11       jfQuaternion_x86( const jfQuaternion& other );
12
13       virtual ~jfQuaternion_x86();
14
15       void normalize();
16
17       void operator*=(const jfQuaternion& other);
18
19       void rotateByVector(const jfVector3& vec);
20
21       void addScaledVector(const jfVector3& vec, jfReal scale) ;
22   protected:
23};
```

The normalize method sets the quaternion to be of unit length [20, p. 187]. The theory behind this function was described in 4.2.2. It is similar to the "jfVector3_x86" method of the same name :

Listing 4.7: jfQuaternion_x86 normalize method

```
1void jfQuaternion_x86::normalize()
2{
3        jfReal d = (m_R*m_R) + (m_I*m_I) + (m_J*m_J) + (m_K*m_K);
4
5        //Zero length quaternion, so give no-rotation quaternion.
6        if(d==0)
7        {
8            m_R = 1;
9            return;
10       }
11
12       d = ((jfReal)1.0/jfRealSqrt(d));
13       m_R *= d;
14       m_I *= d;
15       m_J *= d;
16       m_K *= d;
17}
```

Multipication of a quaternion, described in 4.2.2. Its rotational effect is to rotate this quaternion by the passed in quaternion. It is implemented by the "operator*=" method:

Listing 4.8: jfQuaternion_x86 operator*

```
1void jfQuaternion_x86::operator*=(const jfQuaternion& other)
2{
3    //Millington p.188
4    jfReal oldR = m_R;
5    jfReal oldI = m_I;
6    jfReal oldJ = m_J;
7    jfReal oldK = m_K;
8    m_R = (oldR*other.getR()) - (oldI*other.getI()) - (oldJ*other.
         getJ()) - (oldK*other.getK());
9    m_I = (oldR*other.getI()) + (oldI*other.getR()) + (oldJ*other.
         getK()) - (oldK*other.getJ());
10   m_J = (oldR*other.getJ()) + (oldJ*other.getR()) + (oldK*other.
         getI()) - (oldI*other.getK());
11   m_K = (oldR*other.getK()) + (oldK*other.getR()) + (oldI*other.
         getJ()) - (oldJ*other.getI());
12}
```

This method just creates a quaternion from the passed in vector and multiplies it by this quaternion, thus rotating it. [20, p. 189]

Listing 4.9: jfQuaternion_x86 rotateByVector method

```
1void jfQuaternion_x86::rotateByVector(const jfVector3& vec)
2{
3    jfQuaternion_x86 q(0, vec.getX(), vec.getY(), vec.getZ());
4    (*this) *= q;
5}
```

The final method that we need for quaternions is "addScaledVector". This allows us to update the quaternion by the angular velocity of the rigid body for the duration of the game physics loop. [20, p. 190]

Listing 4.10: jfQuaternion_x86 addScaledVector method

```
1void jfQuaternion_x86::addScaledVector(const jfVector3& vec, jfReal
     scale)
2{
3    //Millington p.190
4    jfQuaternion_x86 q(0,
5                       vec.getX() * scale,
6                       vec.getY() * scale,
7                       vec.getZ() * scale);
8    q *= *this;
9    m_R += q.getR() * ((jfReal)0.5);
10   m_I += q.getI() * ((jfReal)0.5);
11   m_J += q.getJ() * ((jfReal)0.5);
12   m_K += q.getK() * ((jfReal)0.5);
13}
```

Matrices

One might wonder as to the reason behind implementing fixed-size matrix classes. The simple answer is, efficiency. To handle matrices of arbitrary sizes would take a lot of effort and a lot of processing power. Furthermore, in the physics engine designed, the only necessary sizes for matrices are 3×3 and 4×4. Thus, we can design two highly-optimised matrix classes for use with the engine.

3×3 Matrix class

The 3×3 matrix is most often used in the engine to hold the inertia tensor of a body. The definition of this class follows:

Listing 4.11: jfMatrix3_x86 definition

```
1class jfMatrix3_x86 : public jfMatrix3
2{
3    public:
4        jfMatrix3_x86();
5        jfMatrix3_x86(jfReal e0, jfReal e1, jfReal e2,
6                      jfReal e3, jfReal e4, jfReal e5,
7                      jfReal e6, jfReal e7, jfReal e8);
8        jfMatrix3_x86(const jfMatrix3& other);
9        virtual ~jfMatrix3_x86();
10
11       virtual void multiply(const jfMatrix3& other, jfMatrix3*
             result) const;
12
13       virtual void multiply(const jfVector3& vec, jfVector3* result
             ) const;
14
15       virtual void add(const jfMatrix3& other, jfMatrix3* result)
             const;
16
17       virtual jfMatrix3_x86& operator*=(jfReal val);
18
19       virtual jfMatrix3_x86& operator*=(const jfMatrix3& val);
20
21       virtual jfMatrix3_x86& operator+=(const jfMatrix3& val);
22
23       virtual void setInverse(const jfMatrix3& other);
24
25       virtual void getInverse(jfMatrix3* result) const;
26
27       virtual void invert();
28
29       virtual void setTranspose(const jfMatrix3& other);
30
31       virtual void getTranspose(jfMatrix3* result) const;
32
33       virtual void setOrientation(const jfQuaternion& q);
34
35       virtual void transform(const jfVector3& vec, jfVector3*
             result) const;
36
37       virtual void transformTranspose(const jfVector3& vec,
```

```
38                                          jfVector3* result) const;
39
40        virtual void linearInterpolate(const jfMatrix3& a,
41                                                 const jfMatrix3& b,
42                                                 jfReal prop,
43                                                 jfMatrix3* result)
                                                     const;
44
45        virtual void setBlockInertiaTensor(const jfVector3& halfSizes
              , jfReal mass);
46
47    protected:
48    private:
```

Addition of matrices follows the formula presented in **??**

Listing 4.12: jfMatrix3_x86 add method

```
1void jfMatrix3_x86::add(const jfMatrix3& other, jfMatrix3* result)
      const
2{
3    result->setElem(0,m_Elems[0] + other.getElem(0));
4    result->setElem(1,m_Elems[1] + other.getElem(1));
5    result->setElem(2,m_Elems[2] + other.getElem(2));
6    result->setElem(3,m_Elems[3] + other.getElem(3));
7    result->setElem(4,m_Elems[4] + other.getElem(4));
8    result->setElem(5,m_Elems[5] + other.getElem(5));
9    result->setElem(6,m_Elems[6] + other.getElem(6));
10   result->setElem(7,m_Elems[7] + other.getElem(7));
11   result->setElem(8,m_Elems[8] + other.getElem(8));
12}
```

Multipication of matrices is done according to the formula presented in 4.23.

Listing 4.13: jfMatrix3_x86 multiply matrix method

```
1void jfMatrix3_x86::multiply(const jfMatrix3& other, jfMatrix3*
      result) const
2{
3    jfMatrix3_x86 tempResult;
4    tempResult.setElem(0, (m_Elems[0] * other.getElem(0)) +
5                           (m_Elems[1] * other.getElem(3)) +
6                           (m_Elems[2] * other.getElem(6)));
7    tempResult.setElem(1, (m_Elems[0] * other.getElem(1)) +
8                           (m_Elems[1] * other.getElem(4)) +
9                           (m_Elems[2] * other.getElem(7)));
10   tempResult.setElem(2, (m_Elems[0] * other.getElem(2)) +
11                          (m_Elems[1] * other.getElem(5)) +
12                          (m_Elems[2] * other.getElem(8)));
13   tempResult.setElem(3, (m_Elems[3] * other.getElem(0)) +
14                          (m_Elems[4] * other.getElem(3)) +
15                          (m_Elems[5] * other.getElem(6)));
16   tempResult.setElem(4, (m_Elems[3] * other.getElem(1)) +
```

```
17                          (m_Elems[4] * other.getElem(4)) +
18                          (m_Elems[5] * other.getElem(7)));
19    tempResult.setElem(5, (m_Elems[3] * other.getElem(2)) +
20                          (m_Elems[4] * other.getElem(5)) +
21                          (m_Elems[5] * other.getElem(8)));
22    tempResult.setElem(6, (m_Elems[6] * other.getElem(0)) +
23                          (m_Elems[7] * other.getElem(3)) +
24                          (m_Elems[8] * other.getElem(6)));
25    tempResult.setElem(7, (m_Elems[6] * other.getElem(1)) +
26                          (m_Elems[7] * other.getElem(4)) +
27                          (m_Elems[8] * other.getElem(7)));
28    tempResult.setElem(8, (m_Elems[6] * other.getElem(2)) +
29                          (m_Elems[7] * other.getElem(5)) +
30                          (m_Elems[8] * other.getElem(8)));
31    (*result) = tempResult;
32}
```

Mutliplying the matrix by a vector is done simply as if the vector were a 3×1 matrix, and the components of the vector are multiplied with the columns of the matrix :

Listing 4.14: jfMatrix3_x86 multiply vector method

```
1void jfMatrix3_x86::multiply(const jfVector3& vec, jfVector3* result)
      const
2{
3    jfVector3_x86 tempResult;
4    tempResult.setX((vec.getX()*m_Elems[0]) +
5                    (vec.getY()*m_Elems[1]) +
6                    (vec.getZ()*m_Elems[2]));
7    tempResult.setY((vec.getX()*m_Elems[3]) +
8                    (vec.getY()*m_Elems[4]) +
9                    (vec.getZ()*m_Elems[5]));
10   tempResult.setZ((vec.getX()*m_Elems[6]) +
11                   (vec.getY()*m_Elems[7]) +
12                   (vec.getZ()*m_Elems[8]));
13   (*result) = tempResult;
14}
```

This, in effect transforms the vector by the matrix as discussed in 4.2.3. This is why the "transform" method is simply a synonym for this method.

The inverse of the matrix is as described in 4.26

Listing 4.15: jfMatrix3_x86 setInverse method

```
1void jfMatrix3_x86::setInverse(const jfMatrix3& other)
2{
3    jfReal a11 = other.getElem(0);
4    jfReal a12 = other.getElem(1);
5    jfReal a13 = other.getElem(2);
6    jfReal a21 = other.getElem(3);
7    jfReal a22 = other.getElem(4);
8    jfReal a23 = other.getElem(5);
```

```
 9    jfReal a31 = other.getElem(6);
10    jfReal a32 = other.getElem(7);
11    jfReal a33 = other.getElem(8);
12
13    jfReal det = ( (a11*a22*a33) + (a12*a23*a31) + (a13*a21*a32) - (
          a13*a22*a31) - (a12*a21*a33) - (a11*a23*a32) );
14    if (det == jfReal(0.0))
15    {
16        //undefined division
17        return;
18    }
19    jfReal detInv = jfReal(1.0)/det;
20
21    m_Elems[0] = (detInv * ((a22*a33) - (a23*a32)));
22    m_Elems[1] = (detInv * ((a13*a32) - (a12*a33)));
23    m_Elems[2] = (detInv * ((a12*a23) - (a13*a22)));
24    m_Elems[3] = (detInv * ((a23*a31) - (a21*a33)));
25    m_Elems[4] = (detInv * ((a11*a33) - (a13*a31)));
26    m_Elems[5] = (detInv * ((a13*a21) - (a11*a23)));
27    m_Elems[6] = (detInv * ((a21*a32) - (a22*a31)));
28    m_Elems[7] = (detInv * ((a12*a31) - (a11*a32)));
29    m_Elems[8] = (detInv * ((a11*a22) - (a12*a21)));
30}
```

The transpose of a matrix is faster than the inverse calculation and gives the same result if the matrix represents only a rotation.
A full explanation of the transpose of a matrix is given in 4.2.3.

Listing 4.16: jfMatrix3_x86 setTranspose method

```
 1void jfMatrix3_x86::setTranspose(const jfMatrix3& other)
 2{
 3    jfReal a11 = other.getElem(0);
 4    jfReal a12 = other.getElem(1);
 5    jfReal a13 = other.getElem(2);
 6    jfReal a21 = other.getElem(3);
 7    jfReal a22 = other.getElem(4);
 8    jfReal a23 = other.getElem(5);
 9    jfReal a31 = other.getElem(6);
10    jfReal a32 = other.getElem(7);
11    jfReal a33 = other.getElem(8);
12
13    m_Elems[0] = a11;
14    m_Elems[1] = a21;
15    m_Elems[2] = a31;
16    m_Elems[3] = a12;
17    m_Elems[4] = a22;
18    m_Elems[5] = a32;
19    m_Elems[6] = a13;
20    m_Elems[7] = a23;
21    m_Elems[8] = a33;
22}
```

The following function converts a quaternion to a matrix. This is necessary as graphics libraries such as OpenGL need the rotation to be experssed in matrix

form. The implementation is taken from [20, p. 179].

Listing 4.17: jfMatrix3_x86 setOrientation method

```
1void jfMatrix3_x86::setOrientation(const jfQuaternion& q)
2{
3    //Millington p.179
4    m_Elems[0] = 1.0 - ((2*q.getJ()*q.getJ()) + (2*q.getK()*q.getK())
         );
5    m_Elems[1] = (2.0*q.getI()*q.getJ()) + (2.0*q.getK()*q.getR());
6    m_Elems[2] = (2.0*q.getI()*q.getK()) - (2.0*q.getJ()*q.getR());
7    m_Elems[3] = (2.0*q.getI()*q.getJ()) - (2.0*q.getK()*q.getR());
8    m_Elems[4] = 1.0 - ((2.0*q.getI()*q.getI()) + (2.0*q.getK()*q.
         getK()));
9    m_Elems[5] = (2.0*q.getJ()*q.getK()) + (2.0*q.getI()*q.getR());
10   m_Elems[6] = (2.0*q.getI()*q.getK()) + (2.0*q.getJ()*q.getR());
11   m_Elems[7] = (2.0*q.getJ()*q.getK()) - (2.0*q.getI()*q.getR());
12   m_Elems[8] = 1.0 - ((2.0*q.getI()*q.getI()) + (2.0*q.getJ()*q.
         getJ()));
13}
```

$4x4$ Matrix class

As described in matrix:transform, 4×4 matrices can be used to transform a vector. The matrix represents a rotation and add operation, which can transform vectors from one coordinate system to another, and then get the new point relative to another. For the purposes of this engine, the point is the center of gravity of the body, or the body's position coordinate.

The definition of the class looks like the following:

Listing 4.18: jfMatrix4_x86 definition

```
1class jfMatrix4_x86 : public jfMatrix4
2{
3    public:
4        jfMatrix4_x86();
5
6        jfMatrix4_x86(const jfMatrix4& other);
7
8        virtual ~jfMatrix4_x86();
9
10       virtual void multiply(const jfVector3& vec, jfVector3* result
             ) const;
11
12       virtual void multiply(const jfMatrix4& other, jfMatrix4*
             result) const;
13
14       virtual jfReal getDeterminant() const;
15
16       virtual void setInverse(const jfMatrix4& other);
17
18       virtual void getInverse(jfMatrix4* result) const;
```

```
19
20          virtual void invert();
21
22          virtual void setOrientationAndPos(const jfQuaternion& q,
                const jfVector3& pos);
23
24          virtual void transform(const jfVector3& vec, jfVector3*
                result) const;
25
26          virtual void transformInverse(const jfVector3& vec, jfVector3
                * result) const;
27
28          virtual void transformDirection(const jfVector3& vec,
                jfVector3* result) const;
29
30          virtual void transformInverseDirection(const jfVector3& vec,
                jfVector3* result) const;
31
32          virtual void fillColumnMajorArray(float array[16]) const;
33
34          virtual void getAxisVector(unsigned index, jfVector3* result)
                 const;
35      protected:
36      private:
37  };
```

The theory behind the multipication of two matrices was covered in 4.2.3. Essentially it is the same as for a 3×3 matrix, with an extra column and row to deal with:

Listing 4.19: jfMatrix4_x86 multiply matrix method

```
 1void jfMatrix4_x86::multiply(const jfMatrix4& other,
 2                              jfMatrix4* result) const
 3{
 4    jfMatrix4_x86 tempResult;
 5    //Row by columns
 6    tempResult.setElem(0,
 7                            (m_Elems[0] * other.getElem(0)) +
 8                            (m_Elems[1] * other.getElem(4)) +
 9                            (m_Elems[2] * other.getElem(8)) +
10                            (m_Elems[3] * other.getElem(12)));
11    tempResult.setElem(1,
12                            (m_Elems[0] * other.getElem(1)) +
13                            (m_Elems[1] * other.getElem(5)) +
14                            (m_Elems[2] * other.getElem(9)) +
15                            (m_Elems[3] * other.getElem(12)));
16    tempResult.setElem(2,
17                            (m_Elems[0] * other.getElem(2)) +
18                            (m_Elems[1] * other.getElem(6)) +
19                            (m_Elems[2] * other.getElem(10)) +
20                            (m_Elems[3] * other.getElem(14)));
21    tempResult.setElem(3,
22                            (m_Elems[0] * other.getElem(3)) +
23                            (m_Elems[1] * other.getElem(7)) +
24                            (m_Elems[2] * other.getElem(11)) +
```

```
25                          (m_Elems[3] * other.getElem(15)));
26    tempResult.setElem(4,
27                          (m_Elems[4] * other.getElem(0)) +
28                          (m_Elems[5] * other.getElem(4)) +
29                          (m_Elems[6] * other.getElem(8)) +
30                          (m_Elems[7] * other.getElem(12)));
31    tempResult.setElem(5,
32                          (m_Elems[4] * other.getElem(1)) +
33                          (m_Elems[5] * other.getElem(5)) +
34                          (m_Elems[6] * other.getElem(9)) +
35                          (m_Elems[7] * other.getElem(13)));
36    tempResult.setElem(6,
37                          (m_Elems[4] * other.getElem(2)) +
38                          (m_Elems[5] * other.getElem(6)) +
39                          (m_Elems[6] * other.getElem(10)) +
40                          (m_Elems[7] * other.getElem(14)));
41    tempResult.setElem(7,
42                          (m_Elems[4] * other.getElem(3)) +
43                          (m_Elems[5] * other.getElem(7)) +
44                          (m_Elems[6] * other.getElem(11)) +
45                          (m_Elems[7] * other.getElem(15)));
46    tempResult.setElem(8,
47                          (m_Elems[8] * other.getElem(0)) +
48                          (m_Elems[9] * other.getElem(4)) +
49                          (m_Elems[10] * other.getElem(8)) +
50                          (m_Elems[11] * other.getElem(12)));
51    tempResult.setElem(9,
52                          (m_Elems[8] * other.getElem(1)) +
53                          (m_Elems[9] * other.getElem(5)) +
54                          (m_Elems[10] * other.getElem(9)) +
55                          (m_Elems[11] * other.getElem(13)));
56    tempResult.setElem(10,
57                          (m_Elems[8] * other.getElem(2)) +
58                          (m_Elems[9] * other.getElem(6)) +
59                          (m_Elems[10] * other.getElem(10)) +
60                          (m_Elems[11] * other.getElem(14)));
61    tempResult.setElem(11,
62                          (m_Elems[8] * other.getElem(3)) +
63                          (m_Elems[9] * other.getElem(7)) +
64                          (m_Elems[10] * other.getElem(11)) +
65                          (m_Elems[11] * other.getElem(15)));
66    tempResult.setElem(12,
67                          (m_Elems[12] * other.getElem(0)) +
68                          (m_Elems[13] * other.getElem(4)) +
69                          (m_Elems[14] * other.getElem(8)) +
70                          (m_Elems[15] * other.getElem(12)));
71    tempResult.setElem(13,
72                          (m_Elems[12] * other.getElem(1)) +
73                          (m_Elems[13] * other.getElem(5)) +
74                          (m_Elems[14] * other.getElem(9)) +
75                          (m_Elems[15] * other.getElem(13)));
76    tempResult.setElem(14,
77                          (m_Elems[12] * other.getElem(2)) +
78                          (m_Elems[13] * other.getElem(6)) +
79                          (m_Elems[14] * other.getElem(10)) +
80                          (m_Elems[15] * other.getElem(14)));
81    tempResult.setElem(15,
82                          (m_Elems[12] * other.getElem(3)) +
```

```
83                          (m_Elems[13] * other.getElem(7)) +
84                          (m_Elems[14] * other.getElem(11)) +
85                          (m_Elems[15] * other.getElem(15)));
86    (*result) = tempResult;
87}
```

Multiplying a vector by the matrix is similar to for a 3×3 matrix. The "transform" method is a synonym for this method.
However, we also have to take into account the origin of the new coordinate system as discussed in 4.2.3:

Listing 4.20: jfMatrix4_x86 multiply vector method

```
1void jfMatrix4_x86::multiply(const jfVector3& vec,
2                             jfVector3* result) const
3{
4    jfVector3_x86 tempResult;
5    tempResult.setX((vec.getX() * m_Elems[0]) +
6                    (vec.getY() * m_Elems[1]) +
7                    (vec.getZ() * m_Elems[2]) + m_Elems[3]);
8    tempResult.setY((vec.getX() * m_Elems[4]) +
9                    (vec.getY() * m_Elems[5]) +
10                   (vec.getZ() * m_Elems[6]) + m_Elems[7]);
11   tempResult.setZ((vec.getX() * m_Elems[8]) +
12                   (vec.getY() * m_Elems[9]) +
13                   (vec.getZ() * m_Elems[10]) + m_Elems[11]);
14   (*result) = tempResult;
15}
```

If the vector we have represents a direction instead of a point, it is only necessary to transform its orientation and not to get the vector relative to the body's origin. The corresponding method for this follows:

Listing 4.21: jfMatrix4_x86 transformDirection method

```
1void jfMatrix4_x86::transformDirection(const jfVector3& vec,
2                                       jfVector3* result) const
3{
4    jfVector3_x86 tempResult;
5    tempResult.setX((vec.getX() * m_Elems[0]) +
6                    (vec.getY() * m_Elems[1]) +
7                    (vec.getZ() * m_Elems[2]));
8    tempResult.setY((vec.getX() * m_Elems[4]) +
9                    (vec.getY() * m_Elems[5]) +
10                   (vec.getZ() * m_Elems[6]));
11   tempResult.setZ((vec.getX() * m_Elems[8]) +
12                   (vec.getY() * m_Elems[9]) +
13                   (vec.getZ() * m_Elems[10]));
14   (*result) = tempResult;
15}
```

Next comes the calculation of determinant and inverse which is a modified version of the one presented in [20, p. 175], but their method doesn't take into account the bottom elements of the matrix for efficiency. For a more portable design, it was decided to take into account all of the elements of the matrix for the determinant and inverse functions :

Listing 4.22: jfMatrix4_x86 getDeterminant and setInverse methods

```
jfReal jfMatrix4_x86::getDeterminant() const
{
    //Make copy so we don't overwrite our elements
    jfReal a11 = getElem(0);
    jfReal a12 = getElem(1);
    jfReal a13 = getElem(2);
    jfReal a14 = getElem(3);
    jfReal a21 = getElem(4);
    jfReal a22 = getElem(5);
    jfReal a23 = getElem(6);
    jfReal a24 = getElem(7);
    jfReal a31 = getElem(8);
    jfReal a32 = getElem(9);
    jfReal a33 = getElem(10);
    jfReal a34 = getElem(11);
    jfReal a41 = getElem(12);
    jfReal a42 = getElem(13);
    jfReal a43 = getElem(14);
    jfReal a44 = getElem(15);
    return ( ( a11 * a22 * a33 * a44 )
            + ( a11 * a23 * a34 * a42 )
            + ( a11 * a24 * a32 * a43 )

            + ( a12 * a21 * a34 * a43 )
            + ( a12 * a23 * a31 * a44 )
            + ( a12 * a24 * a33 * a41 )

            + ( a13 * a21 * a32 * a44 )
            + ( a13 * a22 * a34 * a41 )
            + ( a13 * a24 * a31 * a42 )

            + ( a14 * a21 * a33 * a42 )
            + ( a14 * a22 * a31 * a43 )
            + ( a14 * a23 * a32 * a41 )

            - ( a11 * a22 * a34 * a43 )
            - ( a11 * a23 * a32 * a44 )
            - ( a11 * a24 * a33 * a42 )

            - ( a12 * a21 * a33 * a44 )
            - ( a12 * a23 * a34 * a41 )
            - ( a12 * a24 * a31 * a43 )

            - ( a13 * a21 * a34 * a42 )
            - ( a13 * a22 * a31 * a44 )
            - ( a13 * a24 * a32 * a41 )

            - ( a14 * a21 * a32 * a43 )
            - ( a14 * a22 * a33 * a41 )
```

```
50            - ( a14 * a23 * a31 * a42 )
51            );
52}
53
54void jfMatrix4_x86::setInverse(const jfMatrix4& other)
55{
56    //Make copy so we don't overwrite our elements
57    jfReal a11 = other.getElem(0);
58    jfReal a12 = other.getElem(1);
59    jfReal a13 = other.getElem(2);
60    jfReal a14 = other.getElem(3);
61    jfReal a21 = other.getElem(4);
62    jfReal a22 = other.getElem(5);
63    jfReal a23 = other.getElem(6);
64    jfReal a24 = other.getElem(7);
65    jfReal a31 = other.getElem(8);
66    jfReal a32 = other.getElem(9);
67    jfReal a33 = other.getElem(10);
68    jfReal a34 = other.getElem(11);
69    jfReal a41 = other.getElem(12);
70    jfReal a42 = other.getElem(13);
71    jfReal a43 = other.getElem(14);
72    jfReal a44 = other.getElem(15);
73    jfReal det = other.getDeterminant();
74
75    if ((jfReal)det == 0)
76    {
77        return;
78    }
79    jfReal detInv = ((jfReal)1.0) / det;
80
81    m_Elems[0] = ( detInv * ( (a22*a33*a44)
82                              + (a23*a34*a42)
83                              + (a24*a32*a43)
84                              - (a22*a34*a43)
85                              - (a23*a32*a44)
86                              - (a24*a33*a42)
87                ));
88    m_Elems[1] = ( detInv * ( (a12*a34*a43)
89                              + (a13*a32*a44)
90                              + (a14*a33*a42)
91                              - (a12*a33*a44)
92                              - (a13*a34*a42)
93                              - (a14*a32*a43)
94                ));
95    m_Elems[2] = ( detInv * ( (a12*a23*a44)
96                              + (a13*a24*a42)
97                              + (a14*a22*a43)
98                              - (a12*a24*a43)
99                              - (a13*a22*a44)
100                             - (a14*a23*a42)
101               ));
102   m_Elems[3] = ( detInv * ( (a12*a24*a33)
103                             + (a13*a22*a34)
104                             + (a14*a23*a32)
105                             - (a12*a23*a34)
106                             - (a13*a24*a32)
107                             - (a14*a22*a33)
```

```
108                 ));
109     m_Elems[4] = ( detInv * ( (a21*a34*a43)
110                                 + (a23*a31*a44)
111                                 + (a24*a33*a41)
112                                 - (a21*a33*a44)
113                                 - (a23*a34*a41)
114                                 - (a24*a31*a43)
115                 ));
116     m_Elems[5] = ( detInv * ( (a11*a33*a44)
117                                 + (a13*a34*a41)
118                                 + (a14*a31*a43)
119                                 - (a11*a34*a43)
120                                 - (a13*a31*a44)
121                                 - (a14*a33*a41)
122                 ));
123     m_Elems[6] = ( detInv * ( (a11*a24*a43)
124                                 + (a13*a21*a44)
125                                 + (a14*a23*a41)
126                                 - (a11*a23*a44)
127                                 - (a13*a24*a41)
128                                 - (a14*a21*a43)
129                 ));
130     m_Elems[7] = ( detInv * ( (a11*a23*a34)
131                                 + (a13*a24*a31)
132                                 + (a14*a21*a33)
133                                 - (a11*a24*a33)
134                                 - (a13*a21*a34)
135                                 - (a14*a23*a31)
136                 ));
137     m_Elems[8] = ( detInv * ( (a21*a32*a44)
138                                 + (a22*a34*a41)
139                                 + (a24*a31*a42)
140                                 - (a21*a34*a42)
141                                 - (a22*a31*a44)
142                                 - (a24*a32*a41)
143                 ));
144     m_Elems[9] = ( detInv * ( (a11*a34*a42)
145                                 + (a12*a31*a44)
146                                 + (a14*a32*a41)
147                                 - (a11*a32*a44)
148                                 - (a12*a34*a41)
149                                 - (a14*a31*a42)
150                 ));
151     m_Elems[10] = ( detInv * ( (a11*a22*a44)
152                                 + (a12*a24*a41)
153                                 + (a14*a21*a42)
154                                 - (a11*a24*a42)
155                                 - (a12*a21*a44)
156                                 - (a14*a22*a41)
157                 ));
158     m_Elems[11] = ( detInv * ( (a11*a24*a32)
159                                 + (a12*a21*a34)
160                                 + (a14*a22*a31)
161                                 - (a11*a22*a34)
162                                 - (a12*a24*a31)
163                                 - (a14*a21*a32)
164                 ));
165     m_Elems[12] = ( detInv * ( (a21*a33*a42)
```

```
166                                 + (a22*a31*a43)
167                                 + (a23*a32*a41)
168                                 - (a21*a32*a43)
169                                 - (a22*a33*a41)
170                                 - (a23*a31*a42)
171                 ));
172     m_Elems[13] = ( detInv * ( (a11*a32*a43)
173                                 + (a12*a33*a41)
174                                 + (a13*a31*a42)
175                                 - (a11*a33*a42)
176                                 - (a12*a31*a43)
177                                 - (a13*a32*a41)
178                 ));
179     m_Elems[14] = ( detInv * ( (a11*a23*a42)
180                                 + (a12*a21*a43)
181                                 + (a13*a22*a41)
182                                 - (a11*a22*a43)
183                                 - (a12*a23*a41)
184                                 - (a13*a31*a42)
185                 ));
186     m_Elems[15] = ( detInv * ( (a11*a22*a33)
187                                 + (a12*a23*a31)
188                                 + (a13*a21*a32)
189                                 - (a11*a23*a32)
190                                 - (a12*a21*a33)
191                                 - (a13*a22*a31)
192                 ));
193 }
```

Now for the method which performs the reverse transformation of the matrix, by first inverting it and then transforming the vector:

Listing 4.23: jfMatrix4_x86 transformInverse method

```
1 void jfMatrix4_x86::transformInverse(const jfVector3& vec,
2                                       jfVector3* result) const
3 {
4     jfMatrix4_x86 tempResult;
5     getInverse(&tempResult);
6     tempResult.multiply(vec,result);
7 }
```

And now the corresponding transform for directions, which uses the property that the inverse is the transpose of the matrix when we are only taking into account orientation. Taken from [20, p. 183]

Listing 4.24: jfMatrix4_x86 transformInverseDirection method

```
1 void jfMatrix4_x86::transformInverseDirection(const jfVector3& vec,
2                                                 jfVector3* result)
                                                     const
3 {
4     jfVector3_x86 tempResult;
```

```
 5    tempResult.setX((vec.getX() * m_Elems[0]) +
 6                    (vec.getY() * m_Elems[4]) +
 7                    (vec.getZ() * m_Elems[8]));
 8    tempResult.setY((vec.getX() * m_Elems[1]) +
 9                    (vec.getY() * m_Elems[5]) +
10                    (vec.getZ() * m_Elems[9]));
11    tempResult.setZ((vec.getX() * m_Elems[2]) +
12                    (vec.getY() * m_Elems[6]) +
13                    (vec.getZ() * m_Elems[10]));
14    (*result) = tempResult;
15}
```

Next, is the function which sets the matrix to be a transform matrix for the given orientation quaternion and origin. Theory behind this is covered in 4.2.3. Implementation is taken from [20, p. 179].

Listing 4.25: jfMatrix4_x86 setOrientationAndPos method

```
 1void jfMatrix4_x86::setOrientationAndPos(const jfQuaternion& q,
 2                                         const jfVector3& pos)
 3{
 4    m_Elems[0] = 1.0 - ((2.0*q.getJ()*q.getJ()) + (2.0*q.getK()*q.
          getK()));
 5    m_Elems[1] = (2.0*q.getI()*q.getJ()) + (2.0*q.getK()*q.getR());
 6    m_Elems[2] = (2.0*q.getI()*q.getK()) - (2.0*q.getJ()*q.getR());
 7    m_Elems[3] = pos.getX();
 8    m_Elems[4] = (2.0*q.getI()*q.getJ()) - (2.0*q.getK()*q.getR());
 9    m_Elems[5] = 1.0 - ((2.0*q.getI()*q.getI()) + (2.0*q.getK()*q.
          getK()));
10    m_Elems[6] = (2.0*q.getJ()*q.getK()) + (2.0*q.getI()*q.getR());
11    m_Elems[7] = pos.getY();
12    m_Elems[8] = (2.0*q.getI()*q.getK()) + (2.0*q.getJ()*q.getR());
13    m_Elems[9] = (2.0*q.getJ()*q.getK()) - (2.0*q.getI()*q.getR());
14    m_Elems[10] = 1.0 - ((2.0*q.getI()*q.getI()) + (2.0*q.getJ()*q.
          getJ()));
15    m_Elems[11] = pos.getZ();
16}
```

The following is a utitlity function which gives a vector from one of the axes represented by the matrix:

Listing 4.26: jfMatrix4_x86 getAxisVector method

```
 1void jfMatrix4_x86::getAxisVector(unsigned index, jfVector3* result)
       const
 2{
 3    result->setX(m_Elems[index]);
 4    result->setY(m_Elems[index+4]);
 5    result->setZ(m_Elems[index+8]);
 6}
```

4.4.2 Rigid Bodies

The rigid body uses all of the preceding classes in its operations. This section will look at how the rigid body class, "jfRigidBody_x86," uses these mathematical classes to simulate the motion of a rigid body.

Firstly, it would be instructive to look at the member variables of the "jfRigidBody" class, which the "jfRigidBody_x86" class inherits.

Listing 4.27: Member variables from jfRigidBody.h

```
1        jfReal m_InverseMass;
2        jfVector3* m_Pos;
3        jfQuaternion* m_Orientation;
4        jfVector3* m_Velocity;
5        jfVector3* m_Rotation;
6        /**
7         * Cached val, changed by Orientation and Position
8         **/
9        jfMatrix4* m_TransformMatrix;
10       /**
11        * Inverse of the inertia tensor of the rigidbody.
12        **/
13       jfMatrix3* m_InverseInertiaTensor;
14       /**
15        * m_InverseInertiaTensor in world coords
16        **/
17       jfMatrix3* m_InverseInertiaTensorWorld;
18       /**
19        * Accumulators for linear forces and torque.
20        **/
21       jfVector3* m_ForceAccum;
22       jfVector3* m_TorqueAccum;
23       /**
24        * Damping is applied to angular motion to remove energy
25        * added through numerical instability (Millington p.211)
26        **/
27       jfReal m_LinearDamping;
28       jfReal m_AngularDamping;
29       /**
30        * Used to set constant acceleration such as gravity.
31        **/
32       jfVector3* m_Accel;
33       jfVector3* m_LastFrameAccel;
34       /**
35        * To deal with sleep state
36        */
37       jfReal m_Motion; //Recency weighted mean of motion
38       bool m_IsAwake;
39       bool m_CanSleep; //Some bodies should not be allowed to sleep
```

The function of each member will now be explained:

The "m_InverseMass" member holds the value of the mass of the body inverted. It is calculated by the equation:

$$invMass = 1/(Mass) \tag{4.28}$$

Inverse mass is more useful than mass for calculations in the engine simulation, so this is the preferred form.

- The "m_Pos" variable holds the current position of the rigid body.

- "m_Orientation" holds the angular orientation of the body, represented as a quaternion.

- "m_Velocity" represents the current linear (non-rotational) component of the velocity for the rigid body.

- "m_Rotation" represents the current angular component of the velocity for the body.

- "m_TransformMatrix" represents the transform matrix which is used for converting between body space and world space. It is recalculated every time the state of the body changes.

- "m_InverseInertiaTensor" represents the inverse of the inertia tensor, described in 4.1.5. As with mass, we store its inverse as it makes calculations simpler.

- "m_InverseInertiaTensorWorld" is the inverse inertia tensor converted into world coordinates, taking into account the orientation of the body.

- "m_ForceAccum" and "m_TorqueAccum" hold the accumulators for linear and angular motion for the body. At integration we take these values and apply them as a single force and torque to the rigid body. This is allowed due to D'Alembert's principle discussed in **??** and 4.1.6.

- "m_Accel" is the constant acceleration of the body. This is added for convenience instead of creating a force generator for gravity, which is usually acting on all bodies.

- "m_LastFrameAccel" is the total acceleration as recorded for the last run of the physics engine loop. This is used in the "jfContact_x86" class to remove

any velocity induced by acceleration, so that objects don't keep interpenetrating due to acceleration.

- "m_Motion," "m_IsAwake" and "m_CanSleep" are all to do with the sleep state of a rigid body. A body is put to sleep if its recency weighted mean value for motion becomes too low. The motion value is updated at each integration step, and encapsulates the kinetic energy of the body into a single scalar. Being recency-weighted, newer values are taken as being more significant to the value. The "m_IsAwake" is set to false at the end of integration when the "m_Motion" value gets sufficiently low. The "m_CanSleep" flag is used to determine whether objects are allowed to sleep or not. This is a similar implementation to the one in [20, p. 387].

Now it is worth looking at the definition of the "jfRigidBody_x86" class and going through some of the methods in detail.

Listing 4.28: jfRigidBody_x86 definition

```
1class jfRigidBody_x86 : public jfRigidBody
2{
3    public:
4        jfRigidBody_x86 ();
5
6        virtual ~jfRigidBody_x86 ();
7
8        virtual void calculateDerivedData();
9
10       virtual void addForce(const jfVector3& force);
11
12       virtual void integrate(jfReal duration);
13
14       virtual void clearAccumulators();
15
16       virtual void addForceAtBodyPoint(const jfVector3& force,
             const jfVector3& point);
17
18       virtual void addForceAtPoint(const jfVector3& force, const
             jfVector3& point);
19
20       virtual void getPointInLocalSpace(const jfVector3& point,
             jfVector3* result) const;
21
22       virtual void getPointInWorldSpace(const jfVector3& point,
             jfVector3* result) const;
23
24   protected:
25       virtual void transformInertiaTensor(
26                           jfMatrix3* iitWorld
27                           , const jfMatrix3& iitBody
28                           , const jfMatrix4& rotMat) const;
29
30       virtual void calculateTransformMatrix(
```

```
31                              jfMatrix4* transformMatrix,
32                              const jfVector3& pos,
33                              const jfQuaternion& orientation) const;
34    private:
35};
```

The "calculateDerivedData" method is called at the end of every integration step and updates the value of the transform matrix and updates the "m_InverseInertiaTensorWorld" to be transformed into world coordinates:

Listing 4.29: jfRigidBody_x86 calculateDerivedData method

```
1void jfRigidBody_x86::calculateDerivedData()
2{
3    m_Orientation->normalize();
4
5    calculateTransformMatrix(m_TransformMatrix, (*m_Pos), (*
         m_Orientation));
6    transformInertiaTensor(m_InverseInertiaTensorWorld,
7                           (*m_InverseInertiaTensor),
8                           (*m_TransformMatrix));
9
10}
```

The "addForce" method simply adds a force to the force accumulator and sets the body to be awake.

Listing 4.30: jfRigidBody_x86 addForce method

```
1void jfRigidBody_x86::addForce(const jfVector3& force)
2{
3    (*m_ForceAccum) += force;
4    m_IsAwake = true;
5}
```

The "clearAccumulators" method simply zeros the force and torque accumulators, this is called at the end of an integration step to prepare for the next physics loop :

Listing 4.31: jfRigidBody_x86 clearAccumulators method

```
1void jfRigidBody_x86::clearAccumulators()
2{
3    m_ForceAccum->clear();
4    m_TorqueAccum->clear();
5}
```

The "integrate" method takes all of the forces and torques applied to the rigidbody and applies them to the rigid body's position and orientation. It also adds any constant acceleration such as gravity. The derived data is then calculated and the force and torque accumulators cleared. Then the "m_Motion" value for the body is checked to see if it is low enough to put the body to sleep.

Listing 4.32: jfRigidBody_x86 integrate method

```
1void jfRigidBody_x86::integrate(jfReal timeStep)
2{
3    //Linear Accel
4    if(!m_IsAwake)
5    {
6        return;
7    }
8
9    jfVector3_x86 angularAccel;
10   (*m_LastFrameAccel) = (*m_Accel);
11   (*m_LastFrameAccel).addScaledVector((*m_ForceAccum),
         m_InverseMass);
12
13   m_InverseInertiaTensorWorld->transform(*m_TorqueAccum, &
         angularAccel);
14
15   //Update velocity and rotation
16   m_Velocity->addScaledVector((*m_LastFrameAccel), timeStep);
17   m_Rotation->addScaledVector(angularAccel, timeStep);
18
19   //Drag
20   (*m_Velocity) *= jfRealPow(m_LinearDamping, timeStep);
21   (*m_Rotation) *= jfRealPow(m_AngularDamping, timeStep);
22
23   //Adjust position and orientation
24   m_Pos->addScaledVector((*m_Velocity), timeStep);
25   m_Orientation->addScaledVector((*m_Rotation), timeStep);
26
27   //Drag
28   (*m_Velocity) *= jfRealPow(m_LinearDamping, timeStep);
29   (*m_Rotation) *= jfRealPow(m_AngularDamping, timeStep);
30
31   calculateDerivedData();
32
33   clearAccumulators();
34
35   // Update the kinetic energy store, and possibly put the body to
36   // sleep.
37   if (m_CanSleep) {
38       jfReal currentMotion = m_Velocity->dotProduct(*m_Velocity) +
39                              m_Rotation->dotProduct(*m_Rotation);
40
41       jfReal bias = jfRealPow(0.01, timeStep);
42       m_Motion = bias*m_Motion + (1-bias)*currentMotion;
43
44       if (m_Motion < SleepEpsilon)
45       {
46           setAwake(false);
47       }
```

```
48        else if (m_Motion > (10 * SleepEpsilon))
49        {
50            //Limit Motion
51            m_Motion = (10 * SleepEpsilon);
52        }
53    }
54}
```

The "addForceAtPoint" method below adds a force and torque corresponding to the point in world coordinates and force provided. The "addForceAtBodyPoint" method is a convenience method which converts the passed in point to world coordinates and then passes it to the addForceAtPoint method.

```
1void jfRigidBody_x86::addForceAtBodyPoint(const jfVector3& force,
2                                          const jfVector3& point)
3{
4    jfVector3_x86 pointInWorldSpace;
5    getPointInWorldSpace(point, &pointInWorldSpace);
6    addForceAtPoint(force, pointInWorldSpace);
7}
8
9void jfRigidBody_x86::addForceAtPoint(const jfVector3& force,
10                                         const jfVector3& point)
11{
12    jfVector3_x86 pointCrossForce;
13    jfVector3_x86 pointCopy = point;
14    pointCopy -= (*m_Pos);
15
16    (*m_ForceAccum) += force;
17    pointCopy.crossProduct(force, &pointCrossForce);
18    (*m_TorqueAccum) += pointCrossForce;
19    m_IsAwake = true;
20}
```

These two methods, "getPointInLocalSpace" and "getPointInWorldSpace" allow for points to be converted to world space from object space and from object space to world space using the rigid bodies transform matrix repectively.

Listing 4.33: jfRigidBody_x86 getPointInLocalSpace method

```
1void jfRigidBody_x86::getPointInLocalSpace(const jfVector3& point,
2                                           jfVector3* result) const
3{
4    m_TransformMatrix->transformInverse(point, result);
5}
6
7void jfRigidBody_x86::getPointInWorldSpace(const jfVector3& point,
     jfVector3* result) const
8{
9    m_TransformMatrix->transform(point, result);
10}
```

This is an internal function called by the "calculateInternals" function to update the transform matrix to correspond to the bodies orientation and position. This function is as in [20, p. 195]

Listing 4.34: jfRigidBody_x86 calculateTransformMatrix method

```
1void jfRigidBody_x86::calculateTransformMatrix(
2                    jfMatrix4* transformMatrix,
3                    const jfVector3& pos,
4                    const jfQuaternion& orientation) const
5{
6    transformMatrix->setElem(0, (1 - (2*orientation.getJ()*
         orientation.getJ()) -
7                                    (2*orientation.getK()*orientation
                                         .getK())));
8    transformMatrix->setElem(1, ((2*orientation.getI()*orientation.
         getJ()) -
9                                (2*orientation.getR()*orientation.
                                     getK())));
10   transformMatrix->setElem(2, ((2*orientation.getI()*orientation.
         getK()) +
11                               (2*orientation.getR()*orientation.
                                     getJ())));
12   transformMatrix->setElem(3, pos.getX());
13   transformMatrix->setElem(4,((2*orientation.getI()*orientation.
         getJ()) +
14                               (2*orientation.getR()*orientation.
                                     getK())));
15   transformMatrix->setElem(5, ((1 - (2*orientation.getI()*
         orientation.getI())) -
16                                   (2*orientation.getK()*orientation
                                         .getK())));
17   transformMatrix->setElem(6, ((2*orientation.getJ()*orientation.
         getK()) -
18                               (2*orientation.getR()*orientation.
                                     getI())));
19   transformMatrix->setElem(7, pos.getY());
20   transformMatrix->setElem(8, ((2*orientation.getI()*orientation.
         getK()) -
21                               (2*orientation.getR()*orientation.
                                     getJ())));
22   transformMatrix->setElem(9, ((2*orientation.getJ()*orientation.
         getK()) +
23                               (2*orientation.getR()*orientation.
                                     getI())));
24   transformMatrix->setElem(10, (1 - (2*orientation.getI()*
         orientation.getI()) -
25                                   (2*orientation.getJ()*orientation
                                         .getJ())));
26   transformMatrix->setElem(11, pos.getZ());
27}
```

The "transformInertiaTensor" method transforms the body's inertia tensor by the objects rotation to give the inertia tensor in world coordinates. It's implementation is modified from [20, p. 203] and was generated using an optimising compiler.

4.4.3 Force Generators

To apply forces to bodies, there needs to be some method of generating a force. One way illustrated by [20] is to use the concept of *Force Generators*.

Force Generators are entities which apply force to rigid bodies at a certain point based on the kind of force generators and the state of the current game world.

An example of a force generator is the "jfAeroForceGenerator_x86" class which generates aerodynamic forces based on the windspeed, the relative position of the surface and the aerodynamic tensor of the body. The other example of a force generator implemented in the final engine is the "jfAeroControlForceGenerator_x86" which allows to control the aerodynamics of an object using a "control". This value is used to set the current aerodynamic tensor for the body as an extrapolation between three tensors, one providing maximum resistance, one minimal and one in between. This allows effects such as controlling the wings of an aircraft as is true on the "jfFlightSimulation_x86" demo on the CD.

All force generators inherit from the base class "jfForceGenerator" which allows us to treat all force generators equally. The interface for this follows:

Listing 4.35: jfForceGenerator definition

```
1class jfForceGenerator
2{
3    public:
4        jfForceGenerator ();
5        virtual ~jfForceGenerator ();
6
7        /* Interface */
8        virtual void updateForce(jfRigidBody* body, jfReal duration)
             const = 0;
9
10   protected:
11   private:
12};
```

It is then trivial to provide a new force generator class which inherits from this class and provides new and unique effects.

[20] provides further examples of force generators including a spring force generator and a gravity force generator. Gravity is implemented in the "jfpx" engine by using the constant acceleration value in the rigid body class.

4.4.4 Collision Detection Pipeline

It is now necessary to look at the collision detection and response part of the engine.
This is by far the most complex part of the engine and took the most time to implement.

It is also the area where I felt deserved the most attention in terms of optimisation using CUDA. This is due to the large amount of operations which are being performed in this process.

Handling collisions in a physical simulation is a non-trivial problem. In physics engines, the usual way to split the problem up is into two distinct phases:

- Collision Detection
- Collision Response

Collision Detection is entirely concerned with determining whether bodies are colliding in the physics world. It is usually split up into two phases:

- Broadphase Collision Detection
- Narrowphase Collision Detection

4.4.5 Collision Detection

Broadphase collision detection is concerned with finding out potential bodies which could be colliding. It is usually implemented with bounding volumes such as Axis Aligned Bounding Boxes, Oriented Bounding Boxes or Bounding Spheres. There is also usually a hierarchy of bounding volumes in a data structure like a tree which lessens the amount of work to be done, as each bounding volume only needs to be checked for collisions with the other bounding volumes at its level in the hierarchy. Once the leaf nodes of the hierarchy are reached, all colliding bounding volumes pass all of the bodies with their volume to the narrowphase collision detection system. A bounding sphere hierarchy was implemented for the jfpx engine but was not used due to the fact that narrowphase collision detection was where it was decided optimisation would be focused.

Narrowphase collision detection involves determining empirically whether two objects are colliding, and if so handle that collision somehow. In the "jfpx" engine, the narrowphase collision detection system defers resolving collisions to the collision resolution system.

Bodies are represented as either spheres, boxes or planes. An assembly of primitives could be assembled using these objects if so desired as discussed in [20, p. 265] The collision geometry of objects is represented by the "jfCollisionPrimitive" class and its derivitive, one for each of sphere, box and plane.

If bodies are colliding, the collision detection system needs to represent the information about the collision so that the collision response system can handle it. The way collisions are encapsulated in the "jfpx" engine is with a "jfContact_x86" class which represents a collision between two objects and has the following information :

- *The Point Of Collision* is the point where the two objects collide in world coordinates.
- *The Collision Normal* is the direction in which impact will be felt when applied in the collision resolution step.

- *Penetration Depth* is the amount of interpenetration there is between two objects. This is necessary as collision detection occurs at discrete intervals and collisions will not usually be seen until after the bodies penetrate. The penetration depth ensures that we apply a force proportional to the penetration depth to seperate the objects.

- *Friction* Holds the amount of static friction at the contact point.

The "jfCollisionData" class encapsulates information which is common to all collisions such as friction and restitution. It also has a resizeable array, or vector, which stores all contacts to be processed by the engine.

For collision detection, methods were written for all of the different possible collision types:

- Sphere-Sphere collisions
- Sphere-Box collisions
- Sphere-Plane collisions
- Box-Box collisions
- Box-Plane collisions

I will describe the sphere-sphere collisions in-depth to show how the collision detection and contact generation are performed. The other collision detection algorithms are available in the "jfCollisionDetector_x86" class on the CD. This method will also be relevant during discussions about CUDA optimisations in 4.5.

Here is presented the method to compute the collisions between spheres based on [20, p. 275] The method follows an idiom followed by all of the collision detector's methods:

- Pass in the collision geometry objects as parameters
- Pass in the global collision data, common to all collisions
- Return the number of contacts generated by the method

Listing 4.36: jfCollisionDetector_x86 sphereAndSphere method start

```
1unsigned
2jfCollisionDetector_x86::sphereAndSphere(const jfCollisionSphere& one
3                                         ,const jfCollisionSphere& two
4                                         ,jfCollisionData* data) const
```

Firstly, the method finds the distance between the two centre points of the spheres and if the distance is too small or too big for the spheres to be colliding, it simply returns:

Listing 4.37: jfCollisionDetector_x86 sphereAndSphere method collision check

```
1    jfVector3_x86 positionOne;
2    jfVector3_x86 positionTwo;
3    jfVector3_x86 midline;
4
5    one.getAxisVector(3, &positionOne);
6    two.getAxisVector(3, &positionTwo);
7
8    positionOne.subtract(positionTwo, &midline);
9    jfReal size = midline.magnitude();
10
11   if ((size <= 0.0f)
12       || (size >= (one.getRadius()+two.getRadius())))
13   {
14       return 0;
15   }
```

Now that we know the spheres are colliding, we need to generate the contact between the two spheres. We set the contact normal to be the unit vector in the direction of the line between the first and the second sphere's centres:

Listing 4.38: jfCollisionDetector_x86 sphereAndSphere method setting contact normal

```
1    jfVector3_x86 normal;
2    midline.multiply((((jfReal)1.0)/size), &normal);
3
4    jfContact_x86 contact;
5    contact.setContactNormal(normal);
```

Next we set the contact point, which we detemine to be midway between the two spheres, or halfway along the vector between the spheres' centres:

Listing 4.39: jfCollisionDetector_x86 sphereAndSphere method setting contact point

```
1    midline.multiply((jfReal)0.5, &midlineHalved);
2    positionOne.add(midlineHalved, &contactPoint);
3    contact.setContactPoint(contactPoint);
```

Following from this, the penetration depth for the contact is set. It is set to be the size of the distance between the centres of the spheres subtracted sum of the radii of the spheres :

Listing 4.40: jfCollisionDetector_x86 sphereAndSphere method setting penetration depth

```
1    contact.setPenetration(one.getRadius()+two.getRadius()
2                           - size);
```

Next, pointers to the two rigid bodies are set, along with the friction and restitution for the collision.

Listing 4.41: jfCollisionDetector_x86 sphereAndSphere method setting body data

```
1    contact.setBodyData(one.getBody(),
2                        two.getBody(),
3                        data->getFriction(),
4                        data->getRestitution());
```

Finally we add the contact to the global contact storage and we return 1, the number of contacts added:

Listing 4.42: jfCollisionDetector_x86 sphereAndSphere method adding contact

```
1    data->addContact(contact);
2    return 1;
```

The other collision detection routines employ a similar strategy:

- Firstly, check whether the objects could be colliding (Early out)
- Now check exhaustively to see if the objects are colliding.
- Generate a contact for each point where objects are colliding.
- Write the contact to the global contact object.
- Return number of contacts found.

In fact, the first step is aided by a class called "jfIntersectionTester_x86"(adapted from [21]) which provides fast methods based on the principle of seperating axes to determine whether two objects are colliding or not. Most collisions will be ruled out in this first step.

4.4.6 Collision Response

This section will give an overview of the collision response system employed by the "jfpx" engine. The collision response system gets as input, a set of contacts which have been calculated by the collision detection system. It must then use these contacts to resolve the collisions between bodies.

In the "jfpx" engine, the collision response system is contained in a class known as "jfContactResolver_x86." It would be instructive to look at the main steps involved in responding to collisions :

- Prepare the contacts for resolution
- Resolve the interpenetrations of bodies
- Resolve the changes in velocities of bodies

Each step of the process will now be described in detail.

Preparing contacts for resolution

Each contact needs the following information before it can be processed:

- The *Contact Basis* - Orthonormal set of axes to represent a contact's *local* space. The x-axis is the contact normal for the contact, with the other two axes chosen at 90° to it. A matrix is constructed to transform between these coordinates and world coordinates.
- The position of the contact relative to each body.
- The relative velocity of the bodies in contact coordinates.

These are the values which are calculated in the calculateInternals method for each contact. The method in the contact resolver simply loops over all contacts, calculating these internal values:

Listing 4.43: jfContactResolver_x86 prepareContacts method

```
1void jfContactResolver_x86::prepareContacts(vector<jfContact*>&
     contacts,
2                                            jfReal timeStep) const
3{
4    vector<jfContact*>::iterator contact;
5    for(contact = contacts.begin(); contact != contacts.end();
         contact++)
6    {
7        (*contact)->calculateInternals(timeStep);
8    }
9}
```

Now, the contacts are ready to be resolved.

Resolving interpenetrations

Resolving the interpenetration of a contact could possibly change the interpenetration of another object. For this reason, once we solve the interpenetration of a contact, we must recalculate the penetrations of the other contacts which are affected. The algorithm is iterative, and at each iteration solves the contact with the deepest penetration, as this is most likely to have a profound effect on the environment, resolves the contact's penetration and then recalculates the penetration of each other contact which has one of the bodies of the resolved contact as one of its bodies. Pseudocode for the algorithm follows(full method available on CD):

Listing 4.44: Pseudocode for penetration resolution algorithm

```
1for 0:max_iterations
2do
3    maxContact := findContactWithDeepestPenetration
4    resolvePenetration(maxContact)
```

```
 5    for contact in contacts
 6    do
 7        for body in contact.bodies
 8        do
 9            for maxBody in maxContact.bodies
10            do
11                if body == maxBody
12                then
13                    recalculatePenetration(body)
14                end
15            end
16        end
17    end
18end
```

The resolution of individual contacts is done using the "applyPositionChange" method of the "jfContact_x86" class. This method uses nonlinear projection as shown in [20, p. 325] to resolve the penetrations in a realistic fashion. Basically, the method uses penetration depth along with the inverse mass and inverse inertia tensor to determine how much impulse to impart in order to cause sufficient angular and linear motion to resolve the interpenetration. The implementation of this is available on the CD. Recalculation of penetration depths can be done without running the collision detection algorithm again by using the technique shown in [20, p. 341], and this is what was implemented.

Resolving the change in velocity

The velocity resolution system is an adapted version of the one from [20, p. 344]. The algorithm for processing contacts is very similar to that as for resolving penetrations:

Listing 4.45: Pseudocode for velocity resolution algorithm

```
 1for 0:max_iterations
 2do
 3    maxContact := findContactWithBiggestProbableVelocityChange
 4    resolveVelocity(maxContact)
 5    for contact in contacts
 6    do
 7        for body in contact.bodies
 8        do
 9            for maxBody in maxContact.bodies
10            do
11                if body == maxBody
12                then
13                    recalculateProbableVelocityChange(body)
14                end
15            end
16        end
17    end
18end
```

The probable velocity change variable is stored in "m_DesiredDeltaVelocity" variable of the "jfContact" class. It is recalculated using the "calculateDesiredDeltaVelocity" method, which calculates the velocity which would result from a "bounce" in the collision. The velocity resolution method is the "calculateVelocityChange" method in the "jfContact_x86" class, the code for which is on the CD. The method calculates the impulse, using different methods for contacts with and without friction imparted in the collision and then applies this to both bodies.

4.4.7 Collision Detection System Review

The collision detection system has now been described in detail. The collision detection pipeline is the most complex and, arguably, the most useful part of the engine. Figure 4.2 shows how the collision detection pipeline fits in with the rest of the engine.

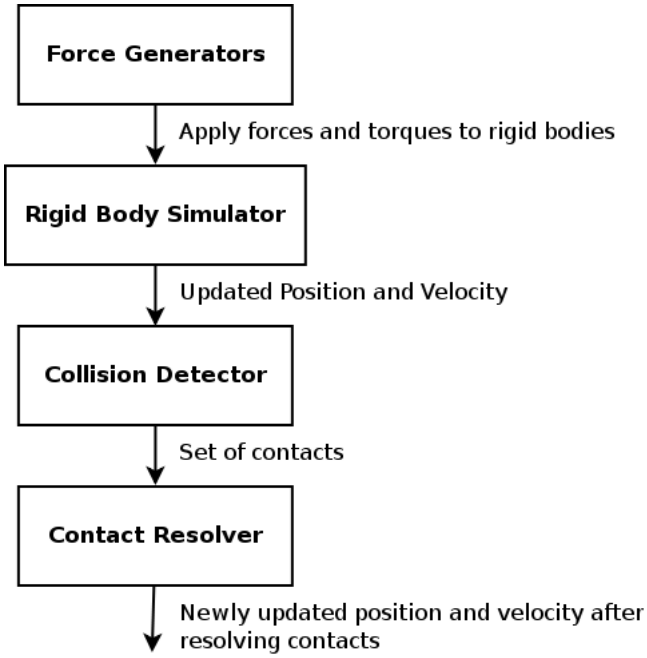


Figure 4.2: Physics Engine Collision Detection Pipeline

4.5 CUDA-optimised Physics Engine

CUDA executes small programs called "kernels" on the GPU. These kernels are run for a number of threads.

CUDA has the concept of grids, blocks and threads. Grids are made up of multiple distinct units known as blocks. Each block of threads is executed independently of threads in other blocks. Threads in a block have a shared memory

space and can synchronize with low overhead. Threads are physically executed in groups, known as "warps" and warps make up blocks. For all CUDA compute capabilities, the number of threads per warp is 32 [10, p. 140]. Therefore using any number of threads that is not a mutliple of a warp size, results in wasted effort.

For CUDA, one of the drawbacks of using it is that memory has to be transferred to and from the GPU. There are multiple ways of doing so but the general model for executing a kernel using CUDA is the following :

1. Allocate memory on GPU
2. Copy memory from host to GPU
3. Run kernel on GPU
4. Copy memory from GPU to host

It should be noted that the memory allocations and memory copies are quite expensive and incur latency. Therefore a kernel which is executed by one thread alone will be slower than the CPU performing the operation. The potential speedup using CUDA comes when multiple threads can execute similar operations in parallel, such as is the case in a long, unrolled loop. While the above x86-only engine was being implemented, potential for using CUDA was discovered in two main locations in the engine :

- The Collision Detection routines
- The Collision Response routines

The reasoning begin the above speculation will now be given.
It should be noted that the only classes in the physics engine where a lot of operations are executed in a loop are in the collision detection and collision response routines. Everywhere else in the engine, the number of floating point operations performed per memory access is too small to be considered feasible for CUDA. In the "jfVector3_x86" class for example, we only perform a handful of floating point operations per instruction. Collision detection was seen as the ideal candidate to test the performance of CUDA in the engine. This was due to the fact that the time taken for collision detection is not constrained, as is the collision response routine, which only runs for a certain number of iterations.
Therefore, in the worst case, collision detection is where performance will decrease the most when many objects are introduced into the physics world.

4.5.1 Collision Detection using CUDA

Once it was decided to implement collision detection using CUDA, a sample routine had to be taken. Of all of the collision detection routines, the sphere-sphere collision detection routines is the most useful for the purposes of this project. This is due to multiple factors:

- Broadphase collision detection can be done using bounding spheres, this can be implemented using the same kernel.
- Rigid bodies can be represented as a collection of spheres as in [16]. Here we could use a similar algorithm to exhaustively test a large amount of spheres with little overhead.
- Narrow-phase collision is simplest using spheres, it is a good starting point to prove that the concept of scaling the number of collision tests works well with CUDA.

To implement this collision detection between spheres using CUDA, firstly a model for testing collisions between multiple spheres at once was come up with. This involves sending the spheres to the collision detector in batches. An exhaustive routine is then called which tests each sphere for intersection with every other sphere. So for N spheres, we do (N^2) collision tests.

This is the routine implemented on the x86 version:

Listing 4.46: jfCollisionDetector_x86 sphereAndSphereBatch method

```
1unsigned jfCollisionDetector_x86::sphereAndSphereBatch(
2                            vector<jfBall*>& spheres,
3                            jfCollisionData* collisionData) const
4{
5    vector<jfBall*>::iterator sphere = spheres.begin();
6    vector<jfBall*>::iterator otherSphere;
7    unsigned contactCount = 0;
8    sphere = spheres.begin();
9    for(sphere = spheres.begin(); sphere != spheres.end() ; ++sphere)
10   {
11       for(otherSphere = (sphere+1) ; otherSphere != spheres.end() ;
             ++otherSphere)
12       {
13           contactCount += sphereAndSphere((*(*sphere)), (*(*
                 otherSphere)), collisionData);
14       }
15   }
16   return contactCount;
17}
```

As we can see, this algorithm is of asymptotic order $O(N^2)$. This means as N increases, the length of time taken to perform the collision tests will be non-linear and will increase exponentially. In fact the time taken will increase dramatically for large N. As an example, if one sphere-sphere collision test takes 0.001 seconds to complete on the CPU, the time taken to compute the collisions between 1000 spheres is 1000 seconds or 16.66 minutes. For 2000 spheres it will take 4000 seconds or 1 hour and 6.66 minutes. For a large number of spheres, such an algorithm is not feasible on the GPU.

However on the GPU using CUDA, we can execute N operations in parallel for N small enough to fit on the GPU. The only factor that will slow us down is the time taken to transfer memory to and from the GPU. But this will only increase

linearly as N increases, so the CUDA version should be faster as N increases as its speed will only decrease linearly while the x86 version decreases exponentially.

4.5.2 CUDA implementation of Sphere-Sphere Collisions

I will explain each part of the CUDA version of the "sphereAndSphereBatch" method in detail.

At the start of the method we copy the "jfBall" objects to "jfCollisionSphereStruct". This is to copy the information represented in the C++ class "jfBall" into the C struct "jfCollisionSphereStructs." This is due to CUDA's incompatiblity with some C++ code, most notably virtual functions[10]. Since most of the classes in the "jfpx" engine use virtual functions, we have to convert between C++ and C data structures when calling functions in a CUDA file. Due to this fact, cuda "struct" versions of many of the core classes were made, including "jfVector3," "jfContact" The "jfCollisionSphereStruct" is a simple struct which holds information needed for generating contacts.

Listing 4.47: jfCollisionSphereStruct definition

```
1typedef struct {
2    jfVector3Struct m_Centre;
3    jfReal m_Radius;
4} jfCollisionSphereStruct;
```

The other C structs used with CUDA are implemented in a similar manner, see the CD for details. So the "assignStruct" method simply copies over the data from the "jfBall" objects to the "jfCollisionSphereStruct" objects. The "sphereSphereCollisionTiled" function from the CUDA file "jfCollisionDetectorKernel_cuda.cu." A prototype for this function has to be kept in the header file "jfCollisionDetector_cuda.h" so that linking can occur. This function processes the collisions between all the spheres using CUDA and will be described in more detail in 4.5.3. Once this function has been completed, the data can be copied from the C-style "jfContactStruct" objects to C++-style "jfContact_x86" objects. It is necessary now to explain how the CUDA kernel is configured and run.

4.5.3 CUDA Configuration

The "sphereSphereCollision" function takes as parameters the spheres to be processed and a pointer to an array where it is to store the resultant contacts. Since the size of the arrays is defined in a common header file "jfCudaConstants," the size of the array need not be passed in.

Listing 4.48: jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionTiled function - start

```
1extern "C" void
2sphereSphereCollisionTiled(jfCollisionSphereStruct* sphereStructs,
```

```
3                                   jfContactStruct* contactStructs)
4{
5    float4 contactNormal[N_CONTACTS];
6    float4 contactPointPenetration[N_CONTACTS]; //Holds point and
         penetration
7    int4 contactValidBodies[N_CONTACTS]; //Holds validity and body
         pointers
8    float4 sphere[N_SPHERES];
```

At the start of the function, memory is allocated which will be used to copy memory to and from the device. "float4" and "int4" are CUDA-specific types which represent a vector of 4 floats and 4 integers respectively. [10] advises the use of these types when transferring to / from the GPU as this coalesces memory addresses and increases the speed of memory transfers.

Next the spheres need to be copied into a form which is easily transferred to the GPU. Then memory is allocated on the device for all of the information that is needed, the spheres and the results for the contact, the contact normal, penetration and validity of the contact. The validity of a contact is important as contacts are begin processed in parallel so there is no means of stopping when a contact is invalid, we simply flag it as invalid and continue. This will then be picked up when the results from the kernel are checked.

Listing 4.49: jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionT function - device allocation and memory copy

```
1    //Copy spheres over to local float4s
2    copySpheresToFloat4s(sphere, sphereStructs, N_SPHERES);
3
4    //Alloc mem on device
5    float4* d_ContactNormal;
6    cudaMalloc((void**)&d_ContactNormal, N_CONTACTS*sizeof(float4));
7
8    float4* d_ContactPointPenetration;
9    cudaMalloc((void**)&d_ContactPointPenetration, N_CONTACTS*sizeof(
         float4));
10
11   int4* d_ContactValidBodies;
12   cudaMalloc((void**)&d_ContactValidBodies, N_CONTACTS*sizeof(int4)
         );
13
14   float4* d_Sphere;
15   unsigned sizeSphere = sizeof(float4) * N_SPHERES;
16   cudaMalloc((void**)&d_Sphere, sizeSphere);
17   cudaMemcpy(d_Sphere, &sphere, sizeSphere, cudaMemcpyHostToDevice)
         ;
```

Next the kernel is called. To call the cuda kernel we specify the number of grids, blocks and threads to run the kernel for. The kernel's execution is described in detail in 4.5.4 :

Listing 4.50: jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionT function - kernel configuration and running

```
1    //Configure Kernel
2    dim3 threads(BLOCK_SIZE, 1);
3    dim3 grid((N_SPHERES / BLOCK_SIZE),1);
4
5    //Call Kernel
6    calculateContacts<<< grid, threads >>>(d_ContactNormal,
7                                           d_ContactPointPenetration
                                                ,
8                                           d_ContactValidBodies,
9                                           d_Sphere,
10                                          N_SPHERES);
```

And then the results are retrieved from the device and device memory is freed.

Listing 4.51: jfCollisionDetectorKernel_cuda.cu sphereSphereCollisionT function - retrieving results from device

```
1    //Get results from device
2    cudaMemcpy(&contactNormal,
3                d_ContactNormal,
4                N_CONTACTS*sizeof(float4),
5                cudaMemcpyDeviceToHost);
6
7    cudaMemcpy(&contactPointPenetration,
8                d_ContactPointPenetration,
9                N_CONTACTS*sizeof(float4),
10               cudaMemcpyDeviceToHost);
11
12   cudaMemcpy(&contactValidBodies,
13               d_ContactValidBodies,
14               N_CONTACTS*sizeof(int4),
15               cudaMemcpyDeviceToHost);
16
17   //Free memory
18   cudaFree(d_Sphere);
19   cudaFree(d_ContactNormal);
20   cudaFree(d_ContactPointPenetration);
21   cudaFree(d_ContactValidBodies);
22
23   //Fill in contact objects with contacts generated by kernel call.
24   generateContacts(contactStructs,
25                    contactNormal,
26                    contactPointPenetration,
27                    contactValidBodies,
28                    N_CONTACTS);
```

4.5.4 CUDA Kernel and Device Functions

Since each thread runs the kernel, the thread needs to figure out where in the array of contacts it is. The contacts are stored in a triangular array, as spheres

will only need to be tested against spheres that they haven't been already tested on, negating half of the potential collision calculations. Each thread has a sphere which it will test against every other sphere in the array of spheres.

Listing 4.52: jfCollisionDetectorKernel_cuda.cu calculateContacts function - start

```
1__global__ void
2calculateContacts(float4* devContactNormal,
3                  float4* devContactPointPenetration,
4                  int4* devContactValidBodies,
5                  float4* devSphere,
6                  unsigned numSpheres)
7{
8    //Define shared memory inside a block
9    //Spheres represented by 3 floats for position and 4th for radius
10   __shared__ float4 sharedSpheres[BLOCK_SIZE];
11
12   float4* globalSpheres = (float4*) devSphere;
13
14   float4 mySphere;
15   int i;
16   int tile;
17   int gtid = blockIdx.x * blockDim.x + threadIdx.x;
18
19   //Make float3 as we want to conserve registers
20   //We will only use as much as we need.
21   //Set all to maximum possible size
22   float3 currentContactNormal[N_SPHERES-1];
23   float4 currentContactPointPenetration[N_SPHERES-1];
24   int currentContactValid[N_SPHERES-1];
25
26   unsigned rowIndex = computeContactRowIndex(N_SPHERES, gtid);
27   unsigned numCols = computeContactNumCols(gtid);
28
29   //Make float4 as we want to coalesce memory accesses
30   float4* globalContactNormal = (float4*) &(devContactNormal[
         rowIndex]);
31   float4* globalContactPointPenetration = (float4*) &(
         devContactPointPenetration[rowIndex]);
32   int4* globalContactValidBodies = (int4*) &(devContactValidBodies[
         rowIndex]);
33
34   //Get current sphere to test all others against
35   mySphere = globalSpheres[gtid];
36
37   i = 0;
```

The first round of sphere calculations is different as the number of spheres will not be an even block for every thread, this is due to the triangular shape of the array.

Listing 4.53: jfCollisionDetectorKernel_cuda.cu calculateContacts function - loading and checking a triangular tile of spheres

```
 1    unsigned amountContactsFirstBlock = (numCols % BLOCK_SIZE);
 2    tile = blockIdx.x;
 3    int idx = tile * blockDim.x + threadIdx.x;
 4    //Collaboratively load sharedSpheres
 5    sharedSpheres[threadIdx.x] = globalSpheres[idx];
 6    __syncthreads();
 7    //Do the first loop, as it is slightly different, not dealing
          with square blocks, triangular ones instead
 8    tileSphereSphereCalculationStart(&(currentContactNormal[i]),
 9                                &(currentContactPointPenetration[i]),
10                                &(currentContactValid[i]),
11                                mySphere,
12                                sharedSpheres,
13                                amountContactsFirstBlock); //don't
                                      skip too far ahead, triangular
                                      array first
14    __syncthreads();
```

Spheres are read into shared memory on the device at each iteration of the loop. This is a technique known as tiling and is based on a similar method employed by [23] in a similar $O(n^2)$ problem. The contacts corresponding to collisions between these shared spheres and the thread's sphere are calculated. This calls the device function "sphereSphereCollision" for the thread's sphere with every other sphere loaded in by the block.

Listing 4.54: jfCollisionDetectorKernel_cuda.cu calculateContacts function - loading and checking square tiles of spheres

```
 1    //Start at our blockID, this ensures we don't check ones
 2    //that have been done by other blocks
 3    for (i = amountContactsFirstBlock, tile = (blockIdx.x+1); i <
          numCols; i += BLOCK_SIZE, tile++)
 4    {
 5        idx = tile * blockDim.x + threadIdx.x;
 6        //Collaboratively load sharedSpheres
 7        sharedSpheres[threadIdx.x] = globalSpheres[idx];
 8        __syncthreads();
 9
10        tileSphereSphereCalculation(&(currentContactNormal[i]),
11                                    &(currentContactPointPenetration[
                                          i]),
12                                    &(currentContactValid[i]),
13                                    mySphere,
14                                    sharedSpheres);
15        __syncthreads();
16    }
```

Finally the results are saved to global memory which is readable by the called function.

Listing 4.55: jfCollisionDetectorKernel_cuda.cu calculateContacts function - saving results to global memory

```
    // Save the result in global memory for the collision response
        step.
    for (i=0;i < numCols;i++)
    {
        //Coalesce memory accesses with float4s instead.
        globalContactNormal[i].x = currentContactNormal[i].x;
        globalContactNormal[i].y = currentContactNormal[i].y;
        globalContactNormal[i].z = currentContactNormal[i].z;
        globalContactNormal[i].w = 0;
        globalContactPointPenetration[i] =
            currentContactPointPenetration[i];
        globalContactValidBodies[i].x = currentContactValid[i];
        globalContactValidBodies[i].y = gtid; //Set body1 index
        globalContactValidBodies[i].z = (i+gtid+1); //Set body2 index
    }
}
```

The last CUDA function that has to be examined is the "sphereSphereCollision" function. The start of the function basically finds the line between the two sphere's centres and computes the distance between them. If the the distance between the centres is less than the radii, then the resultant contact is marked invalid, otherwise it is marked as valid. The "__device__" identifier specifies that the cuda compiler "nvcc" will compile the function for the GPU.

Listing 4.56: jfCollisionDetectorKernel_cuda.cu sphereSphereCollision device function - Validity Check

```
__device__ void
sphereSphereCollision(float3* contactNormal,
                      float4* contactPointPenetration,
                      int* valid,
                      float4 currentSphere,
                      float4 otherSphere
                      )
{
    float3 midline;
    midline.x = currentSphere.x-otherSphere.x;
    midline.y = currentSphere.y-otherSphere.y;
    midline.z = currentSphere.z-otherSphere.z;
    float size = sqrt((midline.x*midline.x) +
                      (midline.y*midline.y) +
                      (midline.z*midline.z));
    float radiiSum = currentSphere.w+otherSphere.w;
    //Don't want us to take it if we have inaccuracy.
    float tolerance = 0.0001;
    if((size > (0.0f+tolerance)) && (size < radiiSum))
    {
        (*valid) = 1;
    }
    else
    {
        (*valid) = 0;
    }
```

And finally the contact data is set to represent a collision between the two spheres. If the contact is invalid, these values will be garbage but they will never be used in that case.

Listing 4.57: jfCollisionDetectorKernel_cuda.cu sphereSphereCollision device function - Saving contact information

```
1    //Set contact normal
2    float invSize = 1.0f/size;
3    contactNormal->x = midline.x * invSize;
4    contactNormal->y = midline.y * invSize;
5    contactNormal->z = midline.z * invSize;
6
7    //Set contact point
8    contactPointPenetration->x = (currentSphere.x + (midline.x*0.5));
9    contactPointPenetration->y = (currentSphere.y + (midline.y*0.5));
10   contactPointPenetration->z = (currentSphere.z + (midline.z*0.5));
11
12   //Set penetration of contact
13   contactPointPenetration->w = (radiiSum - size);
14}
```

This concludes the demonstration of the CUDA part of the engine, for more information on the methods, please see the source on the CD.

Chapter 5

Analysis

This chapter looks at analysing the potential gains in performance obtained from the CUDA-optimised version of the engine with the x86-only version of the engine designed in 4 A detailed comparison of times taken to perform collision detection will be given for various numbers of spheres. An analysis is made based on these comparisons. A conclusion as to which version is better in which situations is given and a look at a hybrid approach is taken.

5.1 Test Machine

The tests were run on an 8-core Intel i7 2.67GHz, with 6 GB DDR3 RAM and a GeForce 250GS graphics card with 1024GB RAM. This GPU has a compute capability of 1.2, which is not the newest standard. Even faster speedups are possible with the newest compute capability GPUs available at the time of writing. The benchmarks were performed under Windows XP 64-bit, with the latest NVIDIA development drivers, 197.13. The system was also tested on linux, but benchmarks were slightly slower there for the CUDA-optimised version. The drivers for linux are not updated as frequently and are possibly not as well-implemented as the Windows XP drivers are, which may explain the slowdown. The linux used was Arch linux, kernel 2.6.32. The GPU tested with is only capable of single-precision floating point instructions, so the x86 was also tested with single precision floating point numbers. Newer CUDA-capable GPUs are capable of single or double precision floating point instructions.

5.2 Collision Detection Benchmarks

The test program,"jfBoxesAndSpheres" contains both an x86-only and CUDA-optimised version. The program allows for an arbitrary number of spheres or boxes to be inserted into the scene. The scene was designed deliberately to test the physics engine to the limit, with the spheres all being stacked on top of each other at the start so that collisions are inevitable. The program also contains a cross-

platform performance timer which was used to time the collision detection process to microsecond accuracy. The main metric which will be used to measure the performance of a collision detection system is the time it takes to finish checking for collisions between N spheres, where N is varied. Timings were taken over a period of around 10 seconds and the average value was taken. The results of these timings with rendering disabled are seen in 5.1 As can be see from 5.1, the

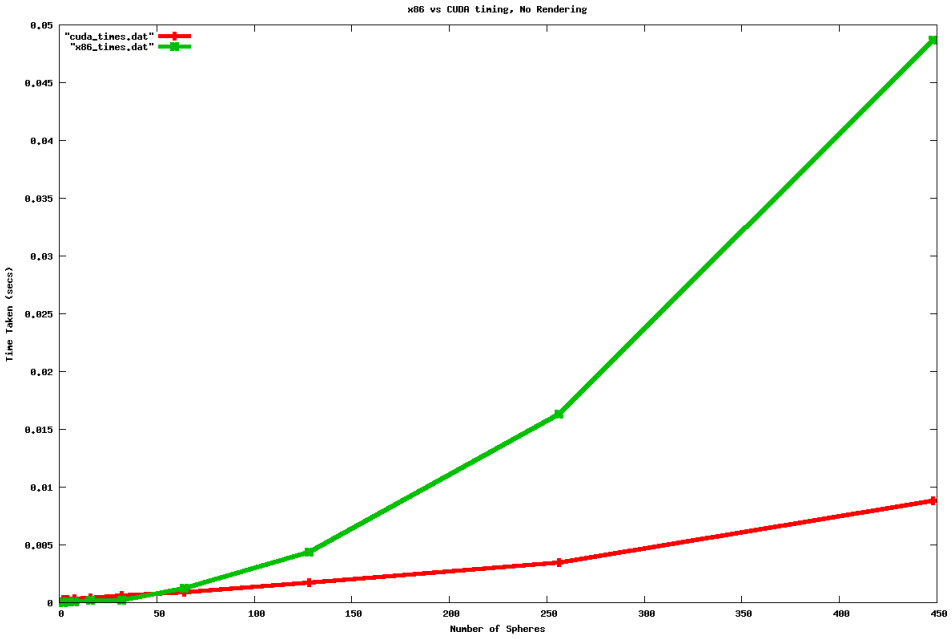


Figure 5.1: Rendering Disabled, Collision Detection times

times for the x86 version of the demo program start to run at substantially higher speeds for $N > 50$. This is what we would have expected, CUDA is displaying its main strength : scalability. As the x86 version starts to get slower and slower, the CUDA version's rate of slowdown is substantially lower. This is due to the many cores of the GPU all processing the contacts in parallel on the GPU. However, for low amounts of spheres, we can see that the x86 version is faster as in 5.2. We can see that the point at which the CUDA version becomes faster than the x86-version is around 50 spheres. Turning on rendering gave a speed improvement but didn't give any difference in the relative performance of the routines as can be seen in 5.3.

5.3 Limitations

One limitation of the CUDA-enabled version of the collision detection routine is the number of spheres with which we can test against. When the number of spheres N gets to about 500, CUDA disallows memory allocations to be the

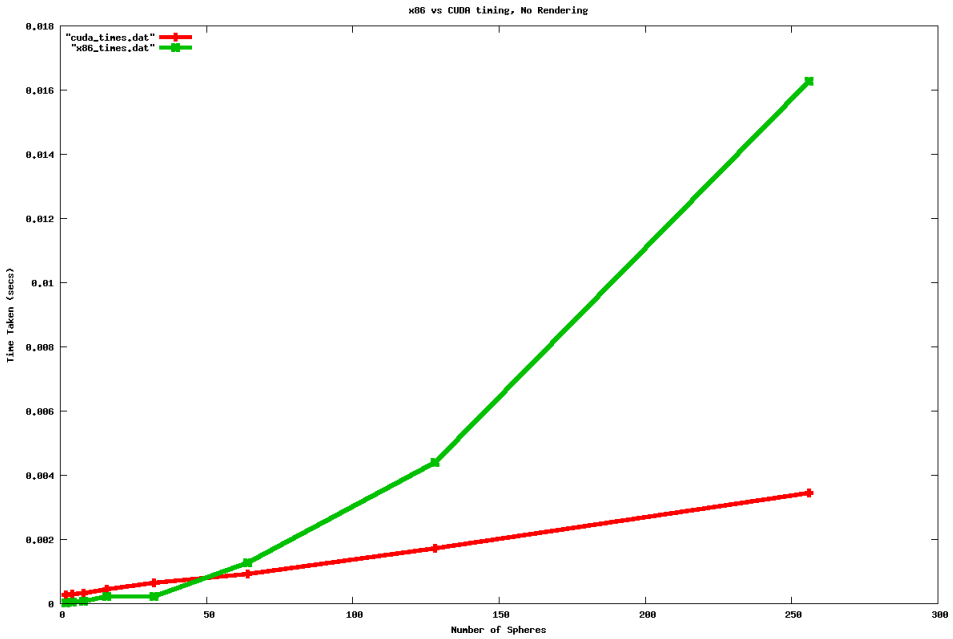


Figure 5.2: Rendering Disabled, Collision Detection times (Small numbers of Spheres)

size of the array of spheres passed into the CUDA kernel. A solution to this problem would be to send the spheres to the collision detection kernel in batches, test each sphere against every other sphere, collect the contacts found and then send another batch to the GPU and run the kernel again. Such a system would integrate well with the above system as it would take longer than the current CUDA routine to run for a small amount of spheres, but could scale to large numbers of spheres relatively easily.

5.4 Hybrid Approach

Looking at the results for timing, there is no doubt that the CUDA version of the code is indeed faster for a large numbers of spheres. The x86 version still outmatches it in certain situations, when the number of spheres is low, however. A worthwhile project would be to come up with a way of determining which version of the code is more suitable for certain physical situations. An easily implemented way of doing it would be to hard code in certain bounds which would indicate when the CPU or GPU is more suitable for the algorithm based on previous tests. This would not give perfect results however, especially in the realm of real-time physical simulation, where the state of the bodies is constantly changing unpredictably. A way of adapting the engine to intelligently "sense" when the GPU would be more suitable would be most useful. Indeed some work has already been done in the area of adaptive physical simulation using the GPU[18].

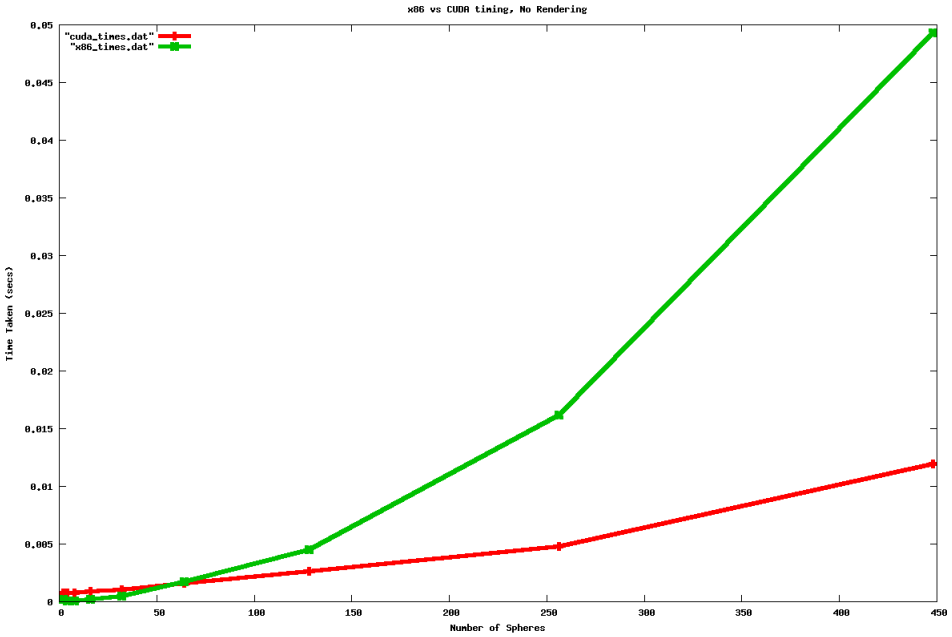


Figure 5.3: Rendering Enabled, Collision Detection times

Chapter 6

Conclusion

This chapter summaries what was achieved in the project.

6.1 Knowledge Acquired

Before beginning this project, I had little knowledge of how a physics engine worked. The first part of the project involved researching existing physics engines and how they worked in a general sense. The system was designed as in 3 at this stage of the project. This involved learning about how to construct a robust C++ API from scratch, polymorphism in C++ and the restrictions imposed by the language. The next step was to begin construction of a simple engine, so the x86 engine described in 4 was constructed. This required learning a great deal about linear algebra, mechanics and vector calculus. The next stage was to learn about how to integrate CUDA into the C++ engine. Following on from this, the CUDA part of the engine had to be constructed which involved learning about the architecture of the GPU. It also involved learning about memory transfers from the CPU to the GPU and how to do this in an efficient manner.

I also gained a lot of knowledge about OpenGL, and how to use it with a physics engine, while creating the test programs.

6.2 Future Work

Future additions that could be made to the project include using CUDA in the other narrow-phase collision detection routines such as checking collisions between boxes. Experimentation with using CUDA for the collision response routines may also yield some benefit, however the benefits of such an algorithm are limited as the number of iterations performed is already limited. Broadphase collision detection could be implemented using the sphereSphere kernel which was developed, bounding spheres could replace the collision spheres used currently. One aspect which was not possible to test, but would be worthwhile, would be testing the performance of the algorithm with multiple GPUs. To take full advantage of hav-

ing multiple GPUs, one would need to specially program the kernel to do so. One would need to figure out how to evenly distribute the load among the GPUs and to manage memory accordingly. Also testing the system using the newest CUDA Compute Capability, 2.0, would be valuable. The GPU with which the tests were performed was only of Compute Capility 1.3. Also newer NVIDIA GPUs are capable of double precision floating point arithmetic, and it would be instructive to test the performance difference between both engines with this. The hybrid approach discussed in 5.4 would also be useful to implement and there is a large amount of scope for engines of other architectures to be integrated with such an engine. The most suitable architecture could then be chosen on the fly, based on the current physical state of the system.

6.3 Summary

This project has involved the construction of a physics engine from scratch, optimised for two architectures. Ultimately however, the success of the project is judged based on the results of analysing the tests of the project. Based on the speedup achieved using CUDA in certain situations, the goal of the project has been reached. It can be concluded that CUDA is suitable for integration into real-time physical systems, and that great speedups can be achieved by doing so.

Bibliography

[1] Kipton Barros. Massively parallel computing with graphics processors and cuda, April 2010.

[2] Bullet community. Bullet engine homepage, April 2010. http://bulletphysics.org/wordpress/.

[3] Bullet community. Bullet engine homepage, April 2010. http://bulletphysics.org/wordpress/?p=50.

[4] Octave community. Octave homepage, April 2010. http://www.gnu.org/software/octave/.

[5] ODE community. Open dynamics engine homepage, April 2010. http://www.ode.org/.

[6] ATI Corporation. Ati corporation's website, February 2010. http://www.ati.com/.

[7] Intel Corporation. Streaming simd extensions - inverse of 4x4 matrix. ftp://download.intel.com/design/PentiumIII/sml/24504301.pdf, March 1999.

[8] Intel Corporation. Intel compiler options for sse generation, April 2010. http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/.

[9] Intel Corporation. Larrabbe website - intel software network, April 2010. http://software.intel.com/en-us/articles/larrabee/.

[10] NVIDIA Corporation. Cuda programming guide 3.0. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf, April 2010.

[11] NVIDIA Corporation. Nvidia corporation's website, February 2010. http://www.nvidia.com/.

[12] Oracle Corporation. Java 3d api customer success stories - mathengine plc, April 2010. http://java.sun.com/javase/technologies/desktop/java3d/in_action/mathengine.html.

[13] Dave H. Eberly. *Game Physics*. Elsevier Science Inc., New York, NY, USA, 2003.

[14] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*. Addison-Wesley Professional, 2003.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[16] Takahiro Harada. Real-time rigid body simulation on gpus. *IPSJ SIG Technical Reports*, 2007(13):79–84, 2007.

[17] Kitware Inc. Cmake, April 2010. http://www.cmake.org/.

[18] Mark Joselli, Esteban Clua, Anselmo Montenegro, Aura Conci, and Paulo Pagliosa. A new physics engine with automatic process distribution between cpu-gpu. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 149–156, New York, NY, USA, 2008. ACM.

[19] Koenig, Andrew Moo, and Barbara E. C++ made easier: The rule of three, June 2001. http://www.drdobbs.com/cpp/184401400.

[20] Ian Millington. *Game Physics Engine Development*. Elsevier, San Diego, CA, 2007.

[21] Ian Millington. Game physics engine development website, April 2010. http://github.com/idmillington/cyclone-physics/.

[22] Jun Ni. Cuda and opencl — development interfaces for multicore programming, December 2009. http://www.uiowa.edu/mihpclab/presentations/staffPresentations/2009_12_23_CUDA%20and%20OpenCL%20Development%20Interface%20for%20Multicore%20Programming.pdf.

[23] Harris Nyland, Mark Prins, Lars, and Jan. *GPU Gems 3*. Addison-Wesley Professional, 2008.

[24] PhysxInfo.com. Popular physics engines comparison: Physx, havok and ode, December 2009. http://physxinfo.com/articles/?page_id=154.

[25] Unittest++ Team. Unittest++, April 2010. http://unittest-cpp.sourceforge.net/.

Appendix A

Including Code

Please see “src” folder of the CD.