Large Polygon Collider

A Modular Physics Engine

Richard Falconer (0502730),
Ben Hallett (0504569),
Alan Hazelden (0523756),
Leigh Robinson (0123489),
George Stanley (0525142)

April 24, 2009

Abstract

A platform-independent physics engine that allows real-time simulation of rigid bodies in both 2D and 3D. The engine is heavily componentised, utilising a solid and extensible object-oriented design to facilitate its use in a wide range of interactive applications. Within this flexible framework there are implementations of numerous published algorithms for collision detection and collision resolution, with the facility to assess their strengths and weaknesses in particular scenarios. In addition to the core physics API and statistics we have created a sandbox application that facilitates the demonstration of the project by allowing an end-user to view and interact with live simulations.

Contents

List of Figures

Chapter 1

Introduction

This document is an account of the process of development of the Large Polygon Collider, a physics engine developed for our 4th year group project. This chapter will cover a brief description of physics engines, their intentions and their common uses. It also describes the aims, justification and motivation of the Large Polygon Collider project.

The following two chapters go into greater detail on the mathematics of physics simulation and the algorithmic methods proposed and employed for solving systems of formulae to produce a representation of Newtonian dynamics that is both accurate and quick to resolve.

The three chapters after that contain greater details on the development of the engine. Respectively, these chapters contain: a detailed specification of the aims of the project and the intended functionality of the physics engine; a description and justification of the modular design used; and a description and explanation of the various algorithms implemented within that modular framework.

We then move in the next chapter to a description of the testing procedures to which we submitted our engine and application, including unit and usability testing, as well as an examination of the results of the performance tests for a number of complementary algorithms.

The succeeding chapter is an account of the overall management of the project, from a description of the methodology employed to a consideration of the legal issues surrounding our use of third-party software libraries and code.

The final chapter is an evaluation of the project as a whole, in which we consider our successes and failures in development. We review the specification, accounting for features we did and did not manage to implement, and comment on the usefulness of the project and its suitability for its intended application. We conclude with

thoughts on what features would be best implemented in future development.

1.1 A brief introduction to physics engines

A physics engine is, at its most basic level, a discrete software component that encapsulates some useful physical simulation algorithms for easy re-use. The exact nature of these algorithms may vary wildly with the intended application, giving rise to classes of engines that each deal with a subset of physical laws. Within these classes there are two further categories of physics engine, namely: *high precision* engines and *real-time* engines.

High precision engines focus on the quantitative quality of the simulations produced, but are not intended for use in time-critical applications, as high accuracy simulations are generally very computationally expensive. They are utilised to produce accurate simulation data for a very wide range of applications within both scientific fields and industry. Real-time engines, by contrast, only simulate what is absolutely necessary to produce a qualitatively realistic simulation, sufficient for use in interactive applications; most commonly games. Both the accuracy and types of simulation supported by this kind of engine are restricted by computational requirements to maintain interactive rates of simulation. Since the focus of these types of engines is to produce an interactive experience they tend to restrict their simulations to 'everyday' phenomena such as Newtonian dynamics and fluids. It is these real-time, interactive simulations that this project focuses on. Note that even with this focus in mind there is still no such thing as a perfect (real-time) physics engine. Different applications will require different capabilities (fluids, smoke, numerical accuracy etc) and different levels of performance. The bottlenecks in physics engine performance will also be application specific; many small particles colliding places demands on different aspects of the engine than those caused by a stable pile of objects.

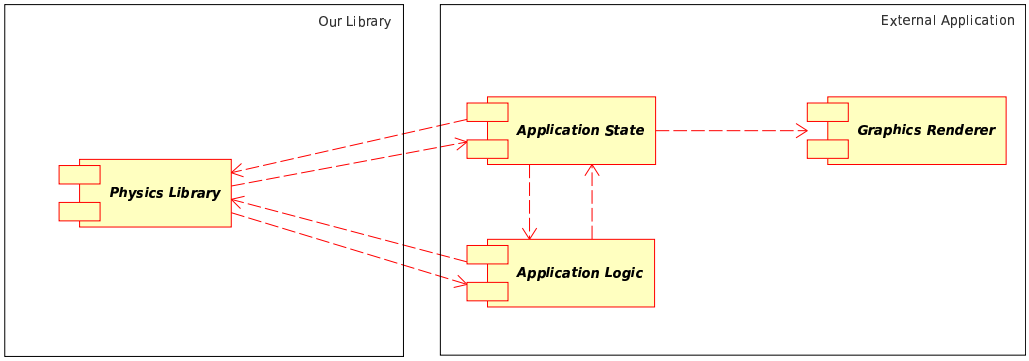


Figure 1.1: Component integration diagram highlighting how a physics component integrates within a host application.

Before physics systems became componentised into discrete packages they were bespoke and heavily tied to the original application, making extension and maintenance difficult. With the separation of the physics components from the host application the way was clear for a true object-oriented framework to emerge to allow the physics components to be easily extended and re-used in other projects. Figure 1.1 shows the modern relationship between a typical application and a modular physics component.

This change became especially important for real-time engines as the available computational speed increases made year after year allow previously intractable simulations to be feasibly added. Many physics engine packages now exist, each having advantages and disadvantages over others in different situations due to the way in which they represent and handle collision detection and resolution.

1.1.1 What makes a good physics engine?

There are several factors that can dictate the effectiveness of a real-time physics engine in a given scenario. The first is the performance of the engine. This obviously depends on the type and algorithmic complexity of the algorithms employed. Naïve implementations fail fairly quickly even with a moderate number of objects as their complexity is at best polynomial. Algorithms that have lower asymptotic complexity are always sought after, as the central goal is to maintain interactive simulation rates. As a consequence very good physics engines have algorithms that can achieve almost linear amortised running times.

Engines can also be judged on the quality of the simulation provided. As mentioned, a real-time physics engine must use approximations to achieve interactive framerates. These optimisations not only reduce the stability of the simulation but also tend to skew the underlying physics. These inaccuracies are often difficult to analyse objectively, but are easily observable to the average user in the form of (for example) "vibrating" objects that ought to be at rest, collapsing (or exploding) structures that ought to be stable, and objects passing through one another where collisions should occur.

With a framework for implementing different algorithms in similar test cases, it is possible to form some conclusions about exactly which algorithms are fastest and most accurate in general cases, as well as which scenarios a given type of algorithm is most suited to.

1.2 Large Polygon Collider

Large Polygon Collider (LPC) is an open-source, real-time, rigid-body physics engine, capable of simulating both two-dimensional and three-dimensional Newtonian dynamics. There are already many open source implementations of similar real-time physics engines for both 2D and 3D physics: some examples include Box2D [2], Simple Physics Engine [15] and the Open Dynamics Engine [20]. What differentiates LPC is that the engine was designed from the outset to support the implementation and analysis of multiple algorithmic solutions to each subcomponent, such as broad-phase collision detection or contact resolution. LPC makes it possible to compare two different implementations of algorithms and provide metrics to quantitatively analyse their performance.

It is outside the scope of this project to implement all of the features of the more established physics engines, especially within three dimensional environments. However it does implement all of the primitive operations (collisions between boxes, spheres, and planes) as well as a selection of more complex operations (e.g. joints and hinges; see 4.3). This has allowed the construction of reasonable comparison scenarios between various algorithm implementations.

The highly modular nature of our physics engine structure means that we can generally implement each algorithm without adversely affecting our ability to implement others. For this reason we did not have a strict ordering of algorithm implementations in our development timeline; we aimed to research their requirements then develop as many as time allowed.

1.2.1 Motivations

A physics engine provides an interesting software engineering challenge, because although it is an easily componentised problem, these subcomponents can prove challenging to implement efficiently. A physics engine structure is naturally represented by the object-orientated paradigm. This is important as it facilitates teamwork and interoperability while providing experience in a modern programming style that is widely used on large scale projects throughout the industry.

In addition to providing experience with team development, a physics engine by its very nature is designed to be reused by a third party as part of a larger application. This provides experience in developing code that is easily extended and reused, again a vital skill in industrial development.

Furthermore, an application that implements the physics engine with a visual representation of the collisions is not only entertaining to develop and use but a very

demonstrable end product. The project will actually be interactive and end users will be able to adjust the simulations to better understand exactly what has been implemented.

1.2.2 Customer

The customer for this project is Nick Pope, who is one of the lead developers for the Warwick Game Design C++ Library (WGD-Lib). One area in which WGD-Lib was lacking was that it had no physics component, and Nick requested that our project resulted in something which could be integrated into the library. We were not be responsible for this integration task, but the design and implementation of our system had to be completed in such a way that it could be easily embedded into WGD-Lib, or any other application or library.

Chapter 2

Fundamentals

Before any attempt is made to discuss how a physics simulation system is engineered, what families of algorithms exist and the issues that arise upon implementation, a discussion of the actual fundamental physical laws must be undertaken along with some supplmentary mathematics for representing these laws in a convenient manner.

2.1 Newton's Laws

Newton's laws of motion and their extension to handle rotations are at the heart of any physics simulation that seeks to simulate the everyday world around us. The three laws are summed up in the following statements:

- Law I :
  body persists in a state of uniform motion (or at rest) unless acted upon by an external unbalanced force

- Law II :
  The net force a body feels is the product of its (assumed constant) mass and acceleration; $\mathbf{F} = m\mathbf{a}$

- Law III :
  Every action has an equal though opposite reaction

With these rather informal definitions we shall now develop some mathematical descriptions that will allow these laws to be expressed within a simulation environment. Further discussion and more advanced treatment can be found in [32].

2.2 Particles

While Newton's laws are applicable to a general 3 dimensional body, we shall start off by only considering *particles*. A particle is the simplest object which can be considered in a dynamical system. A particle only has the properties *mass* (which we shall assume to be constant) and *position* which can vary under the application of *forces*. A particle in this sense is better named a *point mass* as it occupies no volume within its world.

2.2.1 Particle Dynamics

The position, x of a particle can be defined as a function of time, $\mathbf{x}(t)$ allowing the velocity $\mathbf{v}$ and acceleration $\mathbf{a}$ of the particle to be defined as:

$$\mathbf{v}(t) = \frac{d\mathbf{x}(t)}{dt} \tag{2.1}$$

$$\mathbf{a}(t) = \frac{d\mathbf{v}(t)}{dt} = \frac{d^2\mathbf{x}(t)}{dt^2} \tag{2.2}$$

Using Newton's second law from 2.1 we can see that the force acting on a particle to cause this acceleration is,

$$\mathbf{F} = m\mathbf{a} = m\frac{d\mathbf{v}(t)}{dt} \tag{2.3}$$

or if we are interested in the acceleration

$$\mathbf{a} = \frac{d\mathbf{v}(t)}{dt} = \frac{1}{m}\mathbf{F} \tag{2.4}$$

D'Alembert's Principle allows us to consider i forces acting on a particle very simply through the following expression:

$$\mathbf{F}_{net} = \sum_i \mathbf{F}_i \tag{2.5}$$

We now know how to update the position, velocity and acceleration of a particle by applying a set of forces to it. Unfortunately using a force to affect these changes is inconvenient in a simulation environment. The problem is time - the forces must act for a time. To sidestep this issue it is convenient to work with *impulses*, $\mathbf{I}$ defined as

$$\mathbf{I} = \int \mathbf{F}dt = \int m\frac{d\mathbf{v}}{dt}dt \tag{2.6}$$

For simulation this is ideal, as we will be able to assume that the force, $\mathbf{F}$ is *constant* when applied in a sufficiently short time Δt (say the simulation update time - which

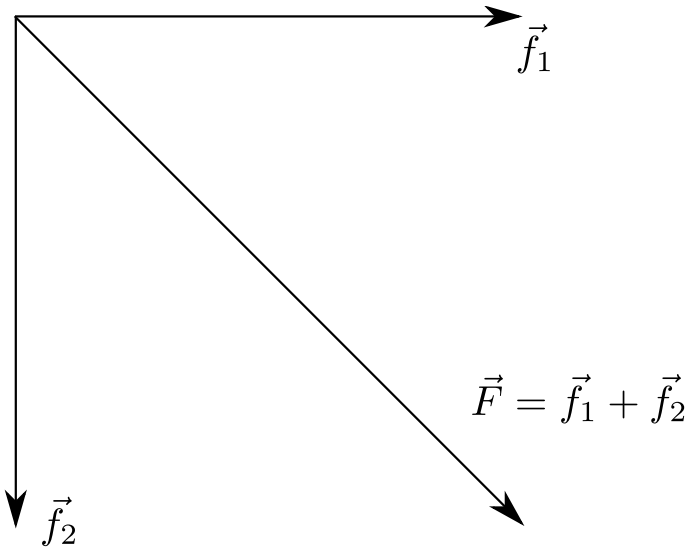


Figure 2.1: Multiple forces acting in an additive manner.

we can control to make this assumption as valid as required), reducing the integrals to

$$\mathbf{I} = \mathbf{F}\Delta t = m\Delta\mathbf{v} \tag{2.7}$$

These equations completely describe the motion of a particle under multiple forces and when coupled with a numerical integrator (see 3.4.1) in the next chapter allows us to calculate these quantites as time is incremented.

2.3 Rigid Bodies

A rigid body can be thought of as a collection of vertices that are separated by fixed distances. With this definition it is clear that a rigid body occupies a volume in the space and as such will be able to orient itself around a point in the space.

A rigid body has a constant mass m that can be calculated by integrating a density function over the volume of the body, though for simplicity we shall also assume the bodies are of constant density which makes the mass easy to calculate with well known formulae. A consequence of having spatial extent is that the body has a *centre of mass* which given our simplifying assumption of constant density will co-incide with the geometric centre of the body. A detailed treatment of the mathematics regarding non-constant density functions and the related problem of calculating the centre of mass can be found in [21].

2.3.1 Local and World Frames

When a rigid body moves through the world, each one of the vertices that make it up move along a path that we can measure in *world co-ordinates* or the *world frame*. That is an observer outside the body can trace a vector from the *world origin* to the point where the vertex of a body is located. This is not the only way we can measure the position of points - we can also measure distances in *body co-ordinates* or the *local frame*. An observer inside the moving body can be thought of standing at the *body origin* with his own set of *body axes*. The directions these axes point in and the body origin position never change from the point of view of the observer inside the body, but an observer outside the body in world space does see these move as the body translates and rotates. A rigorous treatment of linear transformations can be found in [24]. When we calculate physical values careful attention must be taken

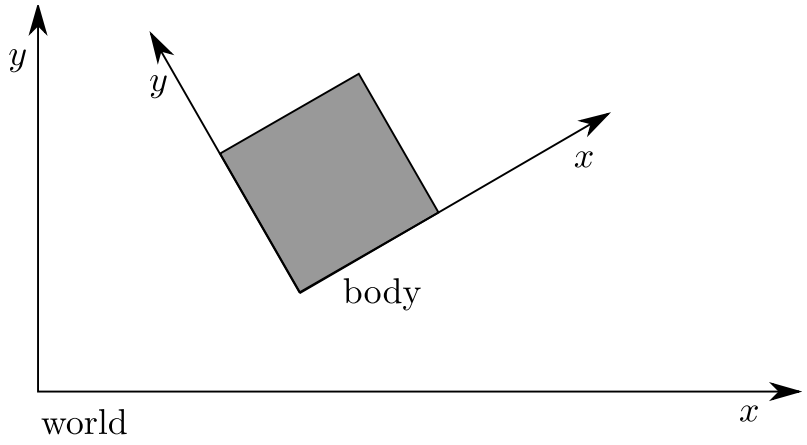


Figure 2.2: A body in world space with its local frame marked.

to ensure that the choice of co-ordinate system is an *inertial frame* - a frame where Newton's Laws are valid. The world frame is an inertial frame while the local frame is not. To appreciate this fact consider what happens as a car accelerates away from a standstill. Free objects inside the car will appear to accelerate backwards to observers inside the car and so by Newton's second law these free bodies should have experienced a force - this force does not exist [32].

Moving Between Frames

All of the vector properties, position, velocity and acceleration, etc must be determined in reference to *some* co-ordinate system. To redefine a value from one frame to another can be easily accomplished by a series of transformations. Consider the position of a point, $\mathbf{P}_{body}$, that is constant in body space. We see that

$$\mathbf{P}_{world} = \mathbf{T}_{transform}\mathbf{P}_{body} \tag{2.8}$$

where $\mathbf{T}_{transform}$ is a matrix derived from the orientation and position of the body origin in world space.

2.3.2 Orientations

The orientation in space of a body (and its implied frame) is a fundamental property that requires a robust mathematical representation. In 2-dimensions orientations are trivial to represent - the body only has 1 degree of rotational freedom, and so only requires a single scalar value, namely the *angle*. The situation is significantly more complex when moving up to 3-dimensions where there are three rotational degrees of freedom.

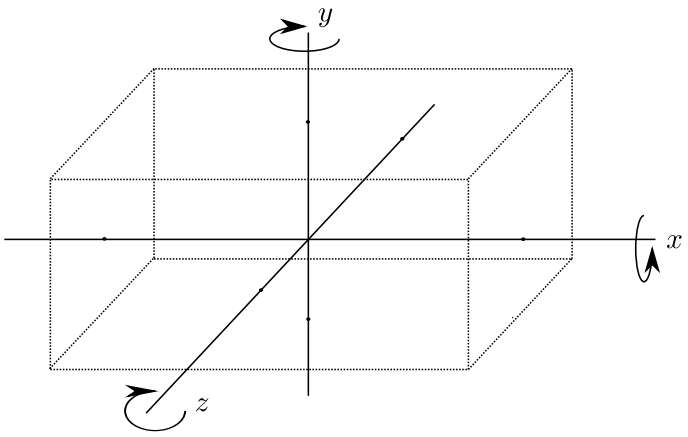


Figure 2.3: A 3-dimensional body with its 3 rotational degrees of freedom marked.

Euler Angles

The intuitive solution is simply to extend the 2-dimensional solution and store three values that represent the angle by which the body has been rotated about each of its local axes. In this fashion any orientation can be defined in terms of combinations of rotations about these three axes. These three angles when taken together are called *Euler angles*. They unfortunately have some issues that make then difficult to use for general orientation representation. It is straightforward to see that the composition of successive rotations are not commutative leading to inconsistent orientation operations. A naïve solution to address this may be to not transform the axes of the body so that they remain fixed relative to the world frame. This unfortunately makes matters even worse. Although fixing the unique representation issue, such a system experiences a phenomenon called *gimbal lock* which means that there are

orientations that this system simply cannot represent when you try to rotate more than $\frac{\pi}{2}$ radians.

Matrix Representation

A rotation can be represented as a transformation by a matrix $\mathbf{R}$. A vector can then easily be rotated by the applying the matrix to the vector, $\mathbf{v}$

$$\mathbf{v}_{rotated} = \mathbf{R}\mathbf{v} \tag{2.9}$$

With this representation we can define any rotation and compositions are well defined with the usual matrix multiplication. Unfortunately using a 3×3 rotation matrix for rotations in 3D incurs the penalty of having nine degrees of freedom. Since floating point precision is not infinite these additional degrees of freedom tend to drift when used in successive computations. This drift has the unfortunate side effect of making the rotation matrix also represent skew or some other undesired transformation. To ensure that the rotation matrices stay representing rotations they need to be checked and 'normalised' periodically, though since they have a much higher degree of freedom than is required this operation may need to be carried out more often than is desired (and is non-trivial in itself).

Quaternions

A way of representing a rotation that offers the best middle ground between degrees of freedom and straighforward well defined composition turns out to be *quaternions*. A quaternion represents a rotation using only four values and has well definied compositions. A quaternion, $\mathbf{Q}$ has the form

$$\mathbf{Q} = xi + yj + zk + w \tag{2.10}$$

where i, j and k are imaginary numbers satisfying $i^2 = j^2 = k^2 = -1$. In this sense quaternions can be thought of generalised 4-dimensional complex numbers. Quaternions though have four degrees of freedom and thus to represent a rotation in three dimensions requires some constraint. This is accomplished by enforcing that the quaternion has unit length by the usual pythagorean definition, $\sqrt{x^2 + y^2 + z^2 + w^2} = 1$. In the same way that normalising a 2-dimensional vector constrains the vector to lie on the edge of a circle this quaternion normalisation can be thought of constraining the motion of a 4-vector to the surface of a 4-dimensional hypersphere, where each point upon the sphere defines an orientation.

Composing quaternion based rotations is accomplished by simply multiplying the two quaternions (taking care to ensure that they both are normalised). The definition of quaternion multiplication and the other common operators (such as transforming a vector by a given quaternion) are defined in [7].

2.3.3 Linear Dynamics

An assumption that the origin of the local co-ordinate frame co-incides with the centre of mass of the body greatly simplifies the dynamics. The linear dynamics of a rigid body now reduce to that of a particle with mass m and furthermore allows the total separation of linear and rotational dynamics, with the body undergoing a linear momentum change *only* when a force is acting through the centre of mass.

With this convention the equations governing the motion of the rigid body can be succinctly defined as

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \tag{2.11}$$

$$\frac{d\mathbf{v}}{dt} = \frac{\mathbf{F}_{net}}{m} \tag{2.12}$$

2.3.4 Rotational Dynamics

When the net forces on a body do not act through the centre of mass we have a rather more complicated situation. The force develops a *torque* on the body which will result in a change in angular velocity. The nature of this change will depend on both the position on the body that the force is being applied to and the resistance of the body to spinning (see Figure 2.4).

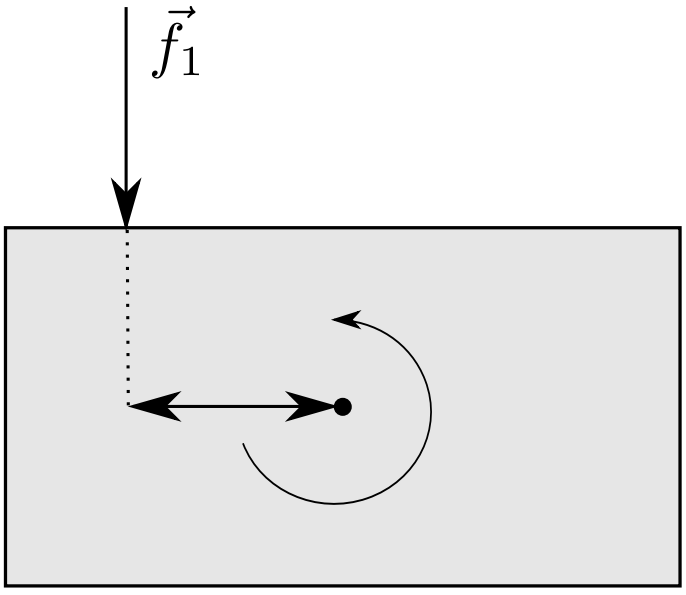


Figure 2.4: A 2-dimensional body with a force producing a torque about its centre of mass.

Moment of Inertia Tensor

The mass of a body describes the inertia or "resistance to movement" for changes in linear motion. This property has a rotational analogue, the *moment of inertia tensor*. In the 2-dimensional case the inertia tensor is simply a scalar value since as discussed in section 2.3.2 there is only one degree of freedom. In 3-dimensions one value is not sufficient we need to define the resistance to start to rotate about all three axes. To appreciate this imagine a long thin rod, it is much easier to spin about the long axis than it is to spin end over end.

Thus for dimensions higher than two we require a tensor to capture the potentially varied behaviour along general axes of rotation. Luckily since the inertia tensor is both real and symmetric it can be simplified by defining it in a basis where only the leading diagonal is non-zero. Additionally for the common geometric primitives this basis aligns with the axes of symmetry through the geometric centre further simplifying the application of the inertia tensor as it can be transformed into arbitrary orientations via a basis change with orientation of the body involved in the calculation. Equation 2.13 defines the principle components of inertia for a cuboid - for more geometric bodies see [7].

$$\mathbf{I} = \begin{bmatrix} \frac{1}{12}m(y^2+z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(x^2+z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(x^2+y^2) \end{bmatrix} \tag{2.13}$$

Given this definition for "rotational mass" we can define the equations that update the rotational velocty, $\dot{\mathbf{q}}$ and rotational acceleration $\dot{\omega}$ (note: $\mathbf{q}$ here is the quaternion representing the orientation of the body).

$$\frac{d\mathbf{q}(t)}{dt} = \frac{1}{2}\omega(t)\mathbf{q}(t) \tag{2.14}$$

$$\frac{d\omega}{dt} = \mathbf{I}^{-1}\tau \tag{2.15}$$

This section has defined all the mathematics that is required to evolve a simulation of rigid bodies that exhibit both linear and rotational dynamics.

Chapter 3

Simulation Details

Going from the mathematical definition of the physical laws to a real time simulation is a non-trivial step. To attain interactive update rates clever algorithm techniques are required. This chapter will develop these techniques and provide references to papers that provide the full details to the interested reader.

3.1 Simulation Scope

The preceding chapter defined many laws of physics relating to the behaviour of rigid bodies under the influence of forces. These laws are rich enough to provide a simulation basis for a wide range of everyday phenomena such as bouncing balls, spinning tops, and with some (small) simplifying assumptions cars and planes.

3.1.1 Why Rigid Bodies?

The work presented so far only applies to rigid bodies, that is bodies whose shape can't change under any amount of force. Removing this simplifying assumption would introduce significant complexity into the mathematics developed in the previous chapter. So called "soft" bodies so not have a fixed centre of mass, a constant moment of inertia tensor or a constant boundary (for collision checking) all of which require a lot more work to take into account.

For the majority of objects in the real world assuming they are perfectly rigid does not lead to significant visual problems. This holds equally for things that are very incompressible such as bricks, tables, and for objects that we usually consider soft

such as the human body. Unfortunately there are a class of interesting objects that cannot be simulated in this manner the most notable of these are cloth.

3.2 Simulation Method

The first thing that the system has to do is setup the physical properties for the objects being simulated and the initial conditions of the simulating "world". For rigid bodies this only entails setting the mass, moment of inertia tensor and the physical dimensions of the object. This can either be done at the start of the simulation or the relevant data can be loaded from an external format.

When the simulation is running careful attention now needs to be given to the *timestep*. The simulation advances by incrementing the time of the system by the timestep, Δt. After a timestep is presented the new positions and orientations of the bodies are calculated and can be assumed static until the next timestep is incremented. The calculations that are required to take place per iteration are listed in table 3.1.

Action	Description
Force Accumulation	The resultant force each body experiences is calculated.
Velocity Update	The forces and torques are used to compute new accelerations which are used to update the velocities by numerical integration.
Position Update	The velocity of the body is updated from the new acceleration by numerical integration.
Broadphase Collision	A list of pairs of bodies that *may* be colliding is generated.
Narrowphase Collision	Each possible collision is checked to see if they are *actually* colliding.
Collision Resolution	Physically correct calculation of resulting velocities after collision.

Table 3.1: List of general processes that need to be carried out for each simulation step.

3.3 Force Accumulation

The task of the force accumulation stage is to simply calculate the resultant force and torque that each body is experiencing due to external agents. All direct forces

are converted into impulses, given the size of Δt (see 2.6) and the torques they produce around the body's centre of mass is calculated.

Friction is not taken into account at this stage as it is a *contact* phenomenon. Bodies only experience friction when they are in contact with each other and will be handled seperately in the contact resolution stage when all the required information is present. It may be tempting to not bother with friction as it does introduce extra complexities to the contact system but many systems require it for realistic simulation : a sphere rolling down a slope will not *roll* if friction is not present. Unfortunately while friction will allow objects to roll, rolling friction is much harder to simulate and will not be considered. A possible solution to this problem is provided by [14].

3.4 Updating Velocity and Position

The task of updating the velocity and positions of the bodies reduces to finding the solutions to the differential equations presented in sections 2.3.3 and 2.3.4 for simulation time $t + \Delta t$ using the forces and torques calculated in the force accumulation step. Finding these solutions falls to the task of a numerical integration algorithms.

3.4.1 Numerical Integration

There are many approaches to numerical integration and we can't hope to detail them all in this section. For a detailed account [1] has a thorough treatment of both mathematical basis and implementation details.

Explicit Euler Integration

This is probably the simplest and most intuitive way to arrive at the new states, for velocity $\mathbf{v}(t + \Delta t)$ and position, $\mathbf{x}(t + \Delta t)$.

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t \frac{d\mathbf{v}(t + \Delta t)}{dt} = \mathbf{v}(t) + \Delta t \frac{\mathbf{F}_{net}}{M} \tag{3.1}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \frac{d\mathbf{x}(t + \Delta t)}{dt} = \mathbf{x}(t) + \Delta t \mathbf{v}(t) \tag{3.2}$$

Unfortunately this is also very inaccurate for anything other than very small Δt. This may not be a problem if the simulation can maintain very small Δt values, but this is not always possible.

Semi-Implicit Euler Integration

Explicit Euler integration fails to use the updated $\mathbf{v}$ which can lead to further inaccuracies when velocity change during the timestep is large (say due to a collision response). This can be remedied by simply using $\mathbf{v}(t + \Delta t)$, when calculating the new position.

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t \frac{d\mathbf{v}(t + \Delta t)}{dt} = \mathbf{v}(t) + \Delta t \frac{\mathbf{F}_{net}}{M} \tag{3.3}$$

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \frac{d\mathbf{x}(t + \Delta t)}{dt} = \mathbf{x}(t) + \Delta t \mathbf{v}(t + \Delta t) \tag{3.4}$$

Accurate Methods

Both of the methods defined above are less than perfect generally and rapidly degrade to poor if Δt increases substantially. This issue can be somewhat mitigated by ensuring the time step cannot grow too large by fixing it to some small value. If this is not acceptable (it will make the simulation rate of evolution linked to simulation complexity) then it may be necessary to move to higher order integration methods such as Runge Kutta, though these are much more expensive to compute and will have a negative impact on performance. Again, full details are available in [1].

3.5 Collision Detection

So far we have developed extensive techniques for evolving the simulation of bodies but have yet to consider what happens in the event that two or more bodies come into contact. We will develop the simulation techniques to handle collisions in the following sections.

3.5.1 Broadphase Collision Detection

Checking if two bodies are about to collide is a potentially expensive procedure (depending on dimension and object complexity) so we would like to minimise the number of checks to the smallest possible number. A naïve method would be to simply process all possible pairs but this quickly becomes infeasible due to the $O(n^2)$ number of comparisons required to check. We can do much better than this if information about the spatial distribution of the bodies is taken into account when building the list of bodies that are potentially colliding.

Uniform Grids

The simplest method is to simply overlay a uniform grid over the extent of the simulation world. The grid partitions the world into a number of equal sized bins with each object associated with the bin that it spatially overlaps. In this system the only possible bodies that can be colliding are ones that share a neighboring grid cell. The farther apart two bodies are the less likely they occupy neighboring grids if the grid size is appropriately chosen and so lessens the likelihood of more expensive pairwise test being required.

Since the world is divided into *equal* sized grids calculating what grid a body overlaps is trivial: simply divide the position of the body by the grid size. Uniform grids however have some serious drawbacks. The most problematic is that the performance gains are highly dependent on the size of the grids chosen - which themselves are highly dependent on the relative sizes of the bodies in the simulation. A full treatment of the optimal usage of grid partitioning is given in [25].

A related approach uses the position directly utilising an efficient spatial hashing function (functionally equivalent to the grid partitioning, but has different optimal conditions) is discussed in [22]. Although the discussion there is focused on deformable objects, the same advice and results will be effective in the general case since, there is a greater demand placed on the spatial partitioning system for deformable objects (as there are usually farm more objects with a closer spatial clustering).

Octrees

Octrees (and their 2-dimensional cousins - the quadtree) are a hierarchical axis aligned partitioning of a space. The root node is usually taken to be the bounding box of the entire world and as the name suggests has eight children. These eight children subdivide the parent node into eight smaller equal-sized subcubes by subdividing along the parent axes by half. These child nodes are then recursively subdivided until a desired subdivision *depth* is reached.

Octrees have an advantage over uniform grids in that they handle objects with a relative size difference. If an object would overlap multiple grid cells it does not need to be entered into each leaf node of the tree - it can simply be entered into the first child that entirely contains it. Further discussion of the scenarios in which octrees perform better can be found in [25].

3.5.2 Narrowphase Collision Detection

Once the number of potentially colliding pairs has been trimmed by the broadphase, a more expensive test must be performed to evaluate which of these are actually colliding, and to generate contact information for each.

Contact information

The exact data required by the contact resolution stage is of course algorithm-dependent, however most implementations require the following information to operate correctly:

- Contact position
- Contact normal
- Penetration distance

Separating Axis Theorem

The separating axis theorem states that for any two convex objects, either there exists a hyperplane separating them (in 2D a line; in 3D a plane) or they are intersecting [8]. In the case that there is no intersection, the separating axis is the vector perpendicular to the separating hyperplane (see diagram 3.1).

We can determine whether an axis separates the shapes by projecting their extents onto the axis vector (see diagram 3.2 for a non-separating axis).

An important property to realise is that for simple shapes, you can enumerate all possible separating axes. Then if none of these directions separate the shapes, then they must be intersecting. In 2D you must test along the normal from each edge and along the direction between the closest pair of vertices (see diagram 3.3 for an example in which just testing edge directions is not sufficient). In 3D there are more possible separating axes, especially for shapes that are more complex than simple geometric primitives.

When finding contacts there are issues of stability to consider. Ideally we would find all contacts and all points which are "close" to being in contact, however in practice it can be too time consuming to find more than one (maybe two in 2D). In 3D the situation is more complex as for stability we may require a contact manifold (3 or more point contacts).

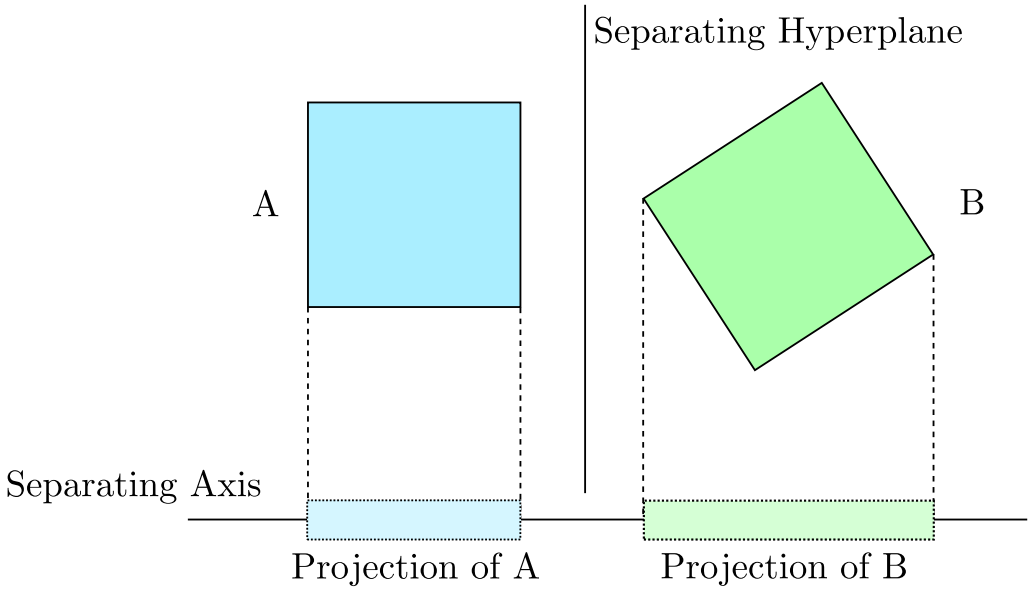


Figure 3.1: A separating axis for two bodies

Advanced convexity-based methods

The separating axis test is usable when the number of potentially separating axes is small, but for more complex shapes it quickly becomes untenable. This makes it unsuitable for collision detection between large meshes. When fast collision detection between convex objects is required, one of two more advanced algorithms is normally used [8], though naturally there are others.

Briefly, and without explaining the mathematics behind it, GJK works by building a representation of the Minkowski difference between two objects [13]. If this Minkowski difference contains the origin, the objects are intersecting; if not then they are separated. However, this only outputs a boolean result: further calculation is required to derive the contact information necessary for collision resolution. The expanding polytope algorithm (EPA) given by Gino van den Bergen [25] accomplishes this, and combined with GJK is known together as GJK-EPA.

Another approach is the V-Clip algorithm, which works by tracking the closest pair of features between objects [19]. This should usually be extremely coherent from frame to frame. Therefore by using a fast hill-climbing algorithm and caching the data from frame to frame, updating the closest points is a basically constant time operation.

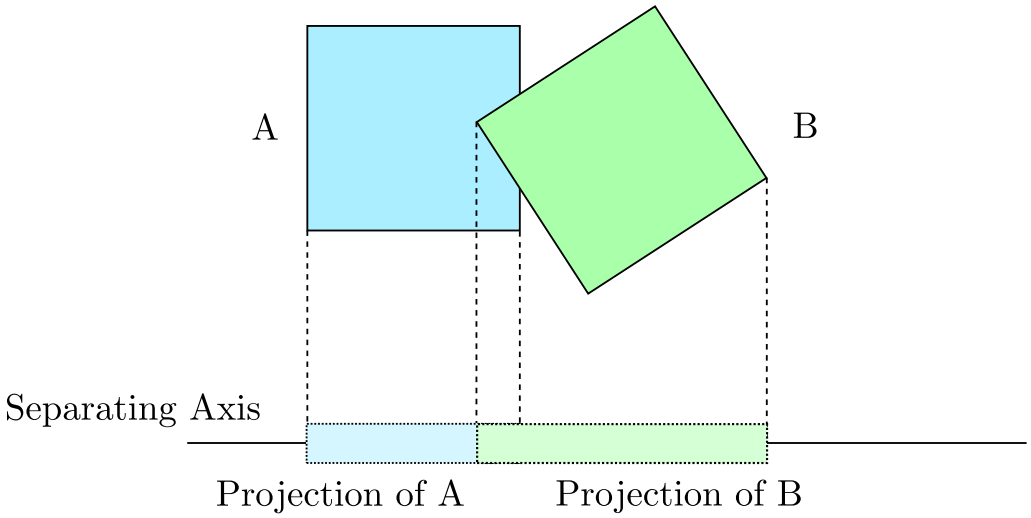


Figure 3.2: These bodies are intersecting, so there is no separating axis

Non-convex collision detection

The above techniques apply only to convex shapes. Further complications occur when the objects involved may be non-convex, or even non-convex and deformable.

Real time interaction of rigid bodies with deformable meshes or cloths (formed from a large number of particles with constraints) is discussed in [11]. The paper *Nonconvex Rigid Bodies with Stacking* [14] discusses the problem of robust concave rigid-body against concave rigid-body testing but the solution they implement is decidedly not real-time: they reference times of 3+ minutes to simulate one frame. However, their mesh data is also significantly more detailed than a real-time engine would require.

3.6 Collision Resolution

Once contacts have been created they are passed to the final stage of the collision pipeline, the contact resolver. The contact resolver must modify the position, orientation, velocity and angular velocity of every body in contact, in a physically realistic manner.

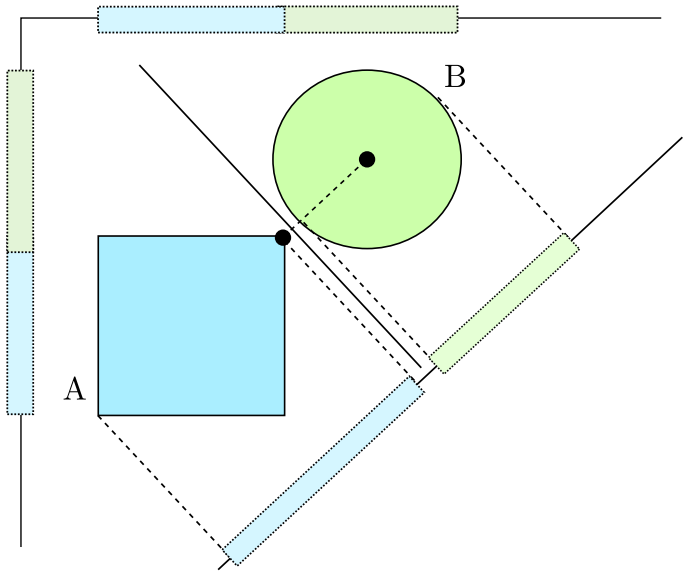


Figure 3.3: Additional tests for seperating axis between box and circle. Notice that in the naïve left and top projections there is an overlap.

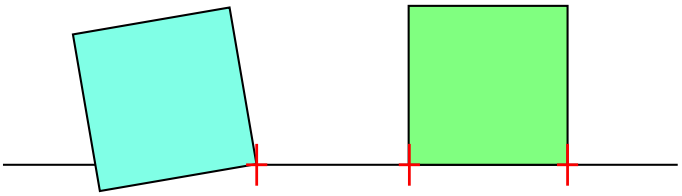


Figure 3.4: Bodies that are at rest require at least two contacts points to remain stable.

3.6.1 Resolving Velocities

At the point of contact, there are two active forces (see diagram 3.5). The contact forces acts to oppose interpenetration, while the friction force acts in a perpendicular direction to oppose motion.

The magnitude of the contact force can determined by solving a system of equations including conservation of momentum and the coefficient of restitution.

Friction is determined by the magnitude of the contact force and by the coefficient of friction.

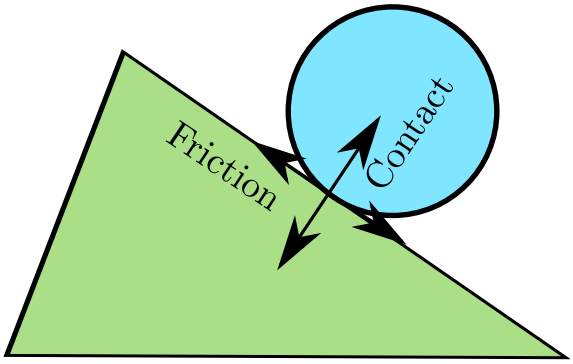


Figure 3.5: Contact force and frictional force.

3.6.2 Resolving Position

The velocities should now be set so that colliding objects are no longer travelling towards one another. If we had detected the collision at the instant these bodies came into contact, this would be fine, but it's more likely that they have interpenetrated slightly. We don't want the simulation to appear with any penetration, so at the end of every frame it is also necessary to adjust the positions of colliding bodies to minimise this penetration.

3.6.3 Iteration

It is unfortunately not enough to do the above once for every contact. Resolving one contact may make another adjacent contact worse, and when objects are densely stacked this is in fact highly probable. The solution to this is to iterate over the list of contacts multiple times: it has been proven that over enough iterations the state will converge to a global solution [3] but you can perform only a fixed number of iterations and get an approximation to the solution which is usually acceptable. In this way, simulation speed can be traded against accuracy.

3.7 Joints and constraints

In addition to handling contacts, a physics engine should be able to constrain the motion of bodies relative to one another.

For example a pivot joint constrains the rotation of two bodies around a common axis. There are a variety of constraint types, especially in 3D where every body has six degrees of freedom.

In fact, joints, arbitrary constraints and even contacts can all be generalised to a representation in matrix form [3] known as a Jacobian matrix. If handled like this, they can all be resolved in the same pipeline in a fairly generic manner.

Chapter 4

Specification

4.1 Project Objectives

Our objectives for the project were summarised in our initial specification as:

> *We aim to create a highly modular and extensible physics engine component written in C++. It will efficiently support the simulation of rigid dynamic bodies under the influence of arbitrary forces (on a world and per body basis) and friction in both 2D and 3D environments. This engine will also allow us to conduct automated profiler tests to investigate the efficiency of the current widely used algorithms. To support the creation and demonstration of this physics engine we intend to create a sandbox application that will allow users to setup and interact with simulations.*

This chapter elaborates the full specification for both components of the project (the API and the sandbox application), including core features and optional features. It does not discuss their design or implementation, which are saved for later chapters. For a full discussion of how well we achieved the goals specified below, see chapter 9.

4.2 Core features

Core features are those on which we intended to focus the greatest development time, since they provide the most important functionality to the final product. It was considered critical that development made significant progress on these objectives for the project to be considered a success.

4.2.1 Collision of primitive shapes and compound shapes

Collision of primitive geometric shapes (2D and 3D) is the most basic level of rigid body dynamics simulation. We hoped that simulating collisions of compound shapes made by simple intersection of sets of primitives would be a fairly straightforward extension.

4.2.2 Modular architecture

The framework needed to be sufficiently modular to allow the implementation of multiple algorithms for each problem, and ultimately to allow other developers to integrate the engine into their own applications. See Figure 5.1 for a high level relationship between the main components.

We aimed to design the architecture in such a way that it can be easily extended: either by ourselves or by others. The tendency of physics engines to be used differently by different applications (see section 1.1) illustrates the importance of a modular architecture. If a particular component of the engine is optimisable for one specific task, then the end user should be able to perform optimisation without altering the engine itself. For detail on how this was implemented see section 5.3.2.

4.2.3 Multiple broad-phase collision detection algorithms

The "broad-phase" is the first attempt the engine makes at determining which objects are *possibly* colliding (rather than naïvely check all possible $O(n^2)$ pairs). A suitable broad-phase algorithm can greatly reduce the amount of computation required for the simulation by avoiding costly detailed collision detection. We aimed to implement some of the following algorithms:

- Sort and sweep
- Bounding volume hierarchies
- Quadtrees/Octrees
- Uniform grid
- Spatial hashing
- BSP trees

4.2.4 Efficient narrow-phase collision detection for geometrical primitives

The "narrow-phase" collision detection looks at the pairs of possibly colliding objects determined by the broad-phase and carries out expensive calculations to determine if they are indeed colliding. If they are deemed to be colliding then the interpenetration depth and contact normal are passed onto the contact resolver.

4.2.5 Multiple collision resolution algorithms

Collision resolution algorithms take a list of colliding objects and resolve the collisions. The choice of algorithm we made here would not only affect performance but also the quality of the simulation, with regards to accuracy and stability. The following are the options we had already identified for the specification:

- Resolving different amounts of penetration per-frame
- For low-speed collisions, altering the coefficient of restitution
- Non-linear penetration resolution
- Generate contacts between nearby (non-touching) objects
- Contact caching
- Using Jacobian matrices to handle joints/constraints
- Warm starting
- Shock propagation

4.2.6 Profiler

Within the physics library it will be important to keep track of per-component performance statistics (such as the amount of time taken to perform an operation, or the number of bodies currently in the simulation) to be generated and analysed. The profiler will be responsible for logging all this information to memory efficiently, and then providing mechanisms for other components to output this information. This information can later be processed into graphs and figures to aid with the analysis of algorithms within the system.

4.2.7 Collision callbacks

In addition to taking the position of objects in the world, we believed it would be important for the physics engine API to be able to inform the application using it of certain events within the simulation. For instance, a game application may require the library to alert it if an object (such as a bullet) collides with the player.

4.2.8 Force/torque generators

Force generators allow the application to define forces on bodies within the world. They can be used to implement attraction or repulsion between objects and more complex constructs such as springs. Torque generators allow for the simulation of motors and other rotary actions.

4.2.9 Simulation visualiser

In order to test the Physics Library and ultimately to demonstrate its functionality, we concluded that a visualiser would be required. This would take an initial state of the world, defined either in the program or within an external file, and then allow the user to see the movement of the objects in real time. It would also require the option to output relevant statistics from the profiler to the screen.

4.2.10 Stable simulation of objects

A challenging task within physics engines is making the simulation *stable*. Many of the optimisations that yield interactive simulation rates do so by making simplifications to the model. Some such simplifications can introduce visible errors into the system, so that, for example, objects resting upon one another might appear to move or collapse when you would expect them to remain upright. Different algorithmic approaches needed to be researched in order to achieve a high level of stability while sacrificing as little of the speed as possible.

4.2.11 Customisability

An important part of the design was presenting all compile time variables to the user in an easy to use class. To some extent this avoids "magic-numbers" obscuring meaning in code.

4.2.12 Dimension-agnostic core features

A key foreseen difficulty in implementing the above features was to ensure they would should work in both 2D and 3D simulations. The design of broad-phase/narrow-phase/collision resolution algorithms and collision callbacks would clearly become especially complex in 3D; their methods either have to be made dimension-agnostic (ruling out a lot of simplifying assumptions) or entirely new classes made. This was going to be further complicated when ensuring the modular architecture was not impaired in the process. We did not foresee the drawing code required in 3D visualisation and tracking code in the Profiler being much of a problem.

4.3 Optional features

We identified other features that would increase the utility of the project, but appreciated that they were more complex and thus would not have been feasible in the limited time available. We decided to closely monitor our work in this area to ensure that we did not waste time working on components that we could not complete. Our justifications for cutting such features from the project are discussed in the Evaluation section.

4.3.1 Joints and constraints

The purpose of these is to specify relationships between bodies; for example saying that two bodies are attached at (but can rotate around) a point. This enables the simulation of vehicles, ragdolls and other more complicated objects.

4.3.2 Sandbox

This is an extension of the visualiser; a fully-featured sandbox application to accelerate testing and demonstrate the capabilities of the physics engine.

While a simple visualiser would suffice to confirm realistic looking behaviour, in order to adequately test all features, a more advanced sandbox application would be required. This would allow the construction of additional test scenarios and greater interaction with the simulation. We appreciated that whilst some user interface components may be shared between the 2D and 3D versions of a sandbox, some sections would be more complex in 3D and would have to be dropped. In our implementation of the Sandbox the 3D version remained essentially a pure visualiser, with only limited interaction.

4.3.3 Fluid dynamics and/or soft body simulation

These can be to a certain extent simulated as a set of particles, with forces acting between the particles, which will both be supported by our engine. At project conception we were pessimistic over how computationally feasible these were, agreeing that it would certainly only be available in 2D (with large amount of optimisation) and certainly not feasible in 3D without spending more time than we had available.

4.3.4 Advanced narrow-phase collision detection : arbitrary meshes

Not all shapes can be easily represented as a union of primitive shapes (i.e. boxes and spheres). A more general approach would be to allow more complex shapes, such as arbitrary meshes. A naive implementation of these would likely give unacceptable performance, so we would have to turn to more advanced algorithms for computational geometry such as GJK or VClip. These would require a significant amount of work to both research and then implement. A discussion of the implemented compromise is discussed in the Evaluation.

4.4 Quality Assurance

All of the above should be stable on all supported platforms (see section 5.4.3), without (for example) throwing fatal exceptions or segmentation faults under any circumstances.

4.4.1 Documentation

Because our end product is a library, we must provide adequate documentation for potential users of that library. We decided to present formal customer documentation in LaTeX, since it is an easily-readable and easily-convertible standard for the production of such documentation. For certain tasks requiring more specific documentation tasks, we decided to use Doxygen [28] and Umbrello [30] to automatically generate class and method descriptions and UML class diagrams, respectively, automatically from the code. Although this was not as trivial as first anticipated, it greatly simplified the task and ensured that the resulting diagrams are provably accurate representations of the real implementation.

API documentation (Doxygen)

For our engine to be useful to end users, detailed API documentation is required. Every class that an end user might want to use or extend has a clear interface and accompanying explanation. This API can be found on the accompanying CD.

User manual

A user manual giving a general overview, instructions for library usage, and a library tutorial can be found in appendix A.

Chapter 5

Design

This chapter describes the design we eventually devised to meet the needs of the specification. It also discusses the research we performed prior to drawing up the structure of the system, and justifies the structure we finally decided upon.

5.1 Research

Before embarking on the design of our project, we sought out implementations of similar physics engine packages by other developers, as well as critically evaluating prior work on the subject performed by Alan.

5.1.1 Existing physics engines and tools

There are many examples of prior art in physics engines. Many are open-source, and some were developed fro the sole purpose of educating others in their construction. The most useful ones we discovered are listed below.

We also considered a number of standards and tools developed around physics engine implementations intended to improve their interoperability with (among others) visualisers, sandboxes and games development tools.

Cyclone Physics System

Cyclone Physics System is an educational 3D physics engine developed for the book Game Physics Engine Development by Ian Millington [18]

By the author's own admission, it is not intended to be a robust physics engine suitable for game development, but rather is aimed at people who want an introduction to how physics works.

Box2D

Box2D is an open source physics engine written by Erin Catto [2].

Box2D_Lite is a simpler version designed for educational purposes. It supports only boxes for collisions and has only one type of joint.

Physics Abstraction Layer

We initially thought that implementing PAL would mean being able to import from Scythe Physics Editor, however it soon became apparent that significant tweaking of the interfaces would be required. The modular nature of our engine also complicated interfacing it with PAL, to the point where interfacing with PAL would have severely reduced the amount of algorithms we could have implemented and compared; one of the main goals of the project. This was unfortunate, as it meant we would not be able to directly compare our engine against existing ones.

COLLADA, Blender

COLLADA [26] is a COLLAborative Design Activity for establishing an interchange file format for interactive 3D applications. It defines an open standard XML schema for exchanging digital assets (in our case, scenes) among various graphics software applications in a standardised manner. Such applications include PAL, the Bullet Physics Library and nVidia's PhysX. These products support reading the abstract found in the COLLADA file and transferring it into a form that the middleware can support and represent in a physical simulation. Supporting COLLADA would allow scenes created in our sandbox to be displayed in this animators as well as loaded into and edited by programs such as Maya, 3ds Max, and Blender. Following further investigation in the design phase, we decided that COLLADA was too heavyweight for our requirements. In order to specify a basic format many base attributes and sections had to be specified. The time cost of interfacing with these requirements outweighed the bonuses we would gain from COLLADA over using a simple XML format such as TinyXML [23].

Scythe Physics Editor

Scythe [12] is a modelling tool specifically designed for use with physics engines. It would have replace our planned sandbox. Investigation into the API also found that it was far too heavyweight for our requirements. The method implementation required by Scythe's interface would have required an inordinate amount of time to implement. The planned process and workflow pipeline initially planned for Scythe/PAL/COLLADA can be found in the minutes in Appendix D.

5.1.2 Prototype

Our design was aided by Alan's third year project, which served as a prototype. It served as a complete feasibility study demonstrating that a physics engine was a viable 4th year project; the project is near limitless in scope and academically challenging whilst being useful outside of academia.

Although not built as a library, much of the code was still applicable to our project, both in the physics component and also in the simulation visualiser. Window creation, basic input handling (key → action), and timers would all be reusable if we chose C++ as a development language (see below). Alan's experience also proved invaluable when bringing the rest of the team up to speed on development.

Once we had decided on a language, much development was still required to establish a framework for library development from the prototype. With the prototype code being ported to a library, encapsulation was much more important. The project structure was completely redesigned. Fundamental distinctions between Bodies, Areas and Shapes also needed to be refined (a major issue with the prototype was that each body could only have one collision shape).

The prototype was also 2D only. The many classes (Body, World) were reworked to be either general enough for use in 3D as well as 2D worlds or split into two copies.

5.2 Development tools

5.2.1 Choice of language

The first step of the design process was to decide which programming language we would be targetting. We looked at:

- Java: Known to all team members. Potentially slow.

- C#: Like Java but better. Language itself is cross-platform but most libraries are not.
- C: Fast. Not object-oriented.
- C++: Fast. Object-oriented. Potential code re-use with Alan's 3rd year project. Warwick Game Design (WGD) Library and most games are written in C++.

We quickly decided to use C++ for all development. The language itself satisfied all the properties we were looking for, and there were other non-language factors that made it attractive. The WGD library written in C++ (and ideally we wanted our engine to complement that), as was Alan's third year project. In addition, it was realised that C++ provided some language features which could aid 2D/3D generic coding (see section 5.4.2) which other languages would not.

5.2.2 Use of external libraries

We endeavored to keep these few in number. Additionally, if possible they were to use static linking and not be referenced in header files. This way, while they would still be required to compile the project, if a pre-compiled binary was created no dependencies would have to be downloaded by the user. A further restriction on libraries is that they be released under a compatible license. For more information on the licensing of the LPC and included libraries, see section 8.4.

5.3 API design

5.3.1 Worlds, bodies & shapes

The interface for a physics engine has become somewhat standardised: all major engines use the same world-body-shape paradigm and there seems to be no obvious reason to do anything different.

To simulate anything, the user must create a world (in other physics engines also known as a space). Rigid bodies are added to the world and collision shapes (in other engines also known as geometry) are added to the bodies. Additionally, there must be a 2D and a 3D version of each.

World

The world must contain a list of bodies. Additionally, it must include implementations of every stage of the collision pipeline (see 5.3.2, below).

Body

As discussed in section 2, a body has a position, orientation, velocity and rotation. It also has a mass and a moment of inertia, which are considered to be constant over time. A list of shapes that are attached to the body must be stored, and these shapes will determine the mass and moment of inertia.

Shape

Shapes must have a position and orientation relative to the body they are attached to (i.e. they are in local coordinates). It will also be necessary to calculate the position and orientation in world coordinates: this information could be cached whenever the body moves to save frequent recomputation.

The shape provides the collision geometry for the contact generator. Therefore the interface of the shape class is affected by whatever the contact generator needs. It was decided not to specify this interface in advance but to leave it open for the implementation.

In 2D the shape class was called `Area` with a synonym `Shape2D`. In 3D the class was named `Volume` with the synonym `Volume`.

In 2D we chose to support the following shapes:

- Circles
- Rectangles
- Polygons
- Capsules
- Line segments

In 3D we chose to initially develop just spheres and boxes, for simplicity. If time permitted we were to implement capsules and cylinders also. Meshes would be probably be far too complicated.

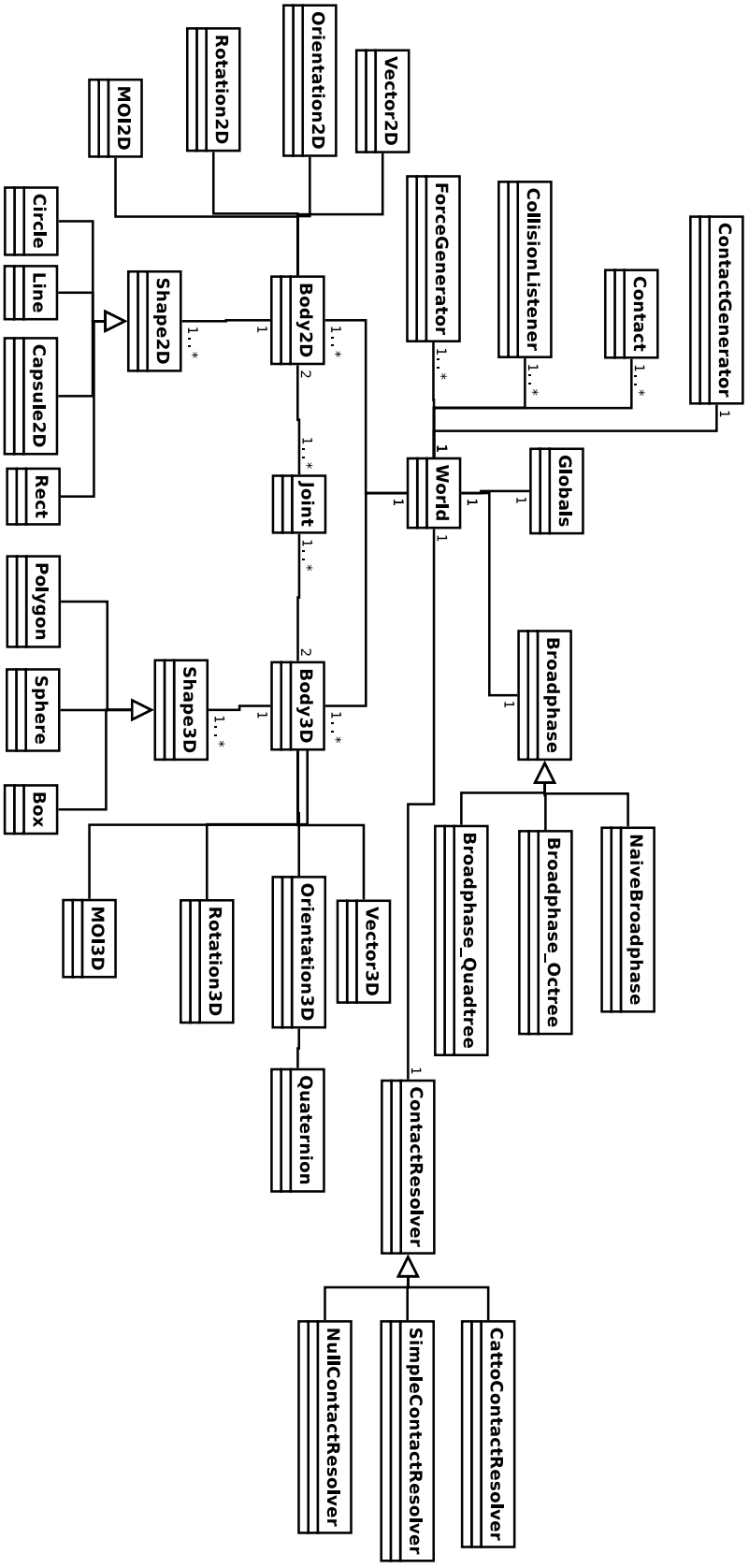


Figure 5.1: UML relationship diagram showing the important classes within the simulation pipeline.

5.3.2 Pipeline

Each frame the following sequence of operations occurs:

- Apply forces
- Move bodies
- Find pairs of potentially colliding bodies (BroadPhase)
- Find contacts between bodies (ContactGenerator)
- Fire callbacks to notify listeners about collisions
- Resolve collisions (ContactResolver)

The user will then draw the bodies at their new positions and repeat the process.

There is an abstract class for each stage of the collision pipeline (broadphase, contact generation, contact resolution). The world has an instance of each. A consequence of this is that a user can implement his own version of one (or more) of these, and as long as the appropriate interface is provided it will work with our pipeline.

An initial design had these classes directly calling methods of one another (i.e. BroadPhase would call a method of ContactGenerator every time it found a potentially colliding pair of bodies). This was quickly changed for two reasons: first to simplify the control flow and second to enable per-component profiling. The revised design had all components being called from World, taking input as a list and returning a new list.

5.3.3 Profiler

Our specification called for some method of recording and outputting performance statistics and data about the state of the simulation. The purposes of this profiler are twofold: first, to aid the identification and elimination of performance bottlenecks during development; second, to quantify the difference in performance between different algorithms and implementations for a given component. Our design therefore includes a `Profiler` class that will handle this task.

The `Profiler` class will require a dedicated timer that is capable of measuring divisions of time to the highest resolution possible within a computer, since its aim is to measure the execution time of short sub-steps within a single execution step. The most accurate such timer we were able to find accessible in C++ is capable of measuring microseconds, so the class that handles this will be called `MicroTimer`.

Finally, the specification requires that the data gathered by the application must be available for graphing and analysis post-execution. We therefore decided that the profiler should output the data it has gathered in comma-separated-variable (csv) format, since that format is both easy to generate from a C++ program, and easy to import into many other programs for analysis. We briefly considered writing our own graphing implementation to handle the graphical display of the information for analysis, but it was decided that was well beyond the scope of the project. Instead, we suggest that the user installs the popular open-source graph-plotting program gnuplot [31], and offer an option that automatically calls a gnuplot batch routine on the output data file.

For information on how the profiler was used in testing, see 7.4.

5.4 Design implications

5.4.1 Speed & efficiency

Certain sections of the design could have an impact on performance. Because we are creating a physics engine as a deliverable which will hopefully be capable of being used in a variety of situations, it must not run prohibitively slowly.

Although the overall infrastructure should be designed to be efficient, individual algorithms should probably initially be created with a focus on speed of implementation rather than speed of execution. This is due to the limited development time available and the large number of desired features. Once something works, it can be tested for efficiency to see if a faster solution would be something worth developing.

Virtual methods vs. inline methods

In C++, to override a method of a class and have it actually work correctly, you must mark it as *virtual*.

You can also mark a method as *inline* to suggest to the compiler that the call to the method should be replaced with the exact code that the function itself. This avoids the overhead of a method invocation and allows many more optimisations to be performed by the compiler. It should be noted that in general this is just used for small methods (such as get or set methods). Inlining is not a magic bullet: it is possible to reduce performance by blindly inlining methods [4].

If a method is virtual, then the inline keyword will be ignored (in all but a handful of instances): since virtual means that the method could be overridden, it is generally

not possible to know at compile time which version of the method will be used.

The upshot of this is a choice between making methods inline and making them virtual.

Container types

The Standard Template Library (STL) provides implementations for many common container types, such as resiseable arrays (`std::vector`) and linked lists (`std::list`). These will have been thoroughly tested and include generalised iterator types that assist in writing generic code.

However, these will be optimised for the general case. It is possible that by developing custom containers specifically for our needs we could optimise further and gain performance improvements. However, this would take valuable development time and the difference in performance is unlikely to be significant enough to justify this (and may not be noticeable at all). Therefore we will use the STL containers.

In some instances there may be a performance difference between using different container types. For example, a vector provides constant time access to any index but is slow to insert elements at any position other than the end. In contrast, a linked list is slower to find the element at a given index, but has faster insertions. We should design our code so that the type of a container can be modified without significant code changes (i.e. using typedefs).

Memory usage and cache coherency

With modern computers have steadily increasing amounts of memory, reducing the memory usage of our library will not likely be a relevant factor.

However, it (and many other factors) could affect cache coherency: the property that data stored together in memory may be faster to access due to the increased liklihood that the data is contained in the same level of cache.

5.4.2 Code re-use between 2D and 3D

Conceptually, a 2D and a 3D physics engine are very similar. They will have the same components, the same control flow, and many large sections of the code will be identical. However the data structures that algorithms are working on, and some of the algorithms themselves, have to be different.

We wanted to create both a 2D and a 3D engine, but clearly just duplicating the shared code was not an acceptable solution. Very quickly problems of synchronisation would have popped up, where one version of the code could be updated without the other being changed. Possible solutions to this problem are discussed below.

To our knowledge, there are no existing physics engines which directly support both 2D and 3D simulation.

2D as a special case of 3D

The only approach to this problem seen in existing engines is to basically ignore the third dimension when working in 2D. However this is easier said than done, and suffers from some code repetition (in those select few cases where either 2D methods can be generalised 3D methods, or one method can work for both dimensions).

All movement has to be constrained to a 2D plane and you must ensure that all forces and impulses act only in this plane as well.

3D simulation is significantly more computationally complex than 2D, so using this approach incurs a significant overhead.

Preprocessor macros

The C preprocessor is run as a preliminary stage of compilation of any C or C++ code, before it is sent to the compiler. It is possible to use the preprocessor to conditionally compile parts of the code, which would allow us to mark certain sections to only be used in 2D or in 3D.

However, in general macro usage is discouraged as being dangerous and non-type-safe. We would probably end up with a large number of `#ifdef TWODEE` statements, and would need to compile the library twice: once for 2D and once for 3D. This strange compilation step would have been scriptable under Linux, but there was uncertainty as to how we might convince Visual Studio to handle this.

Templates

Templates are a C++ feature, whereby a class or function can be made "generic". Many problems tackled by macros can be solved more elegantly using templates as it reduces code duplication.

The template parameter can be plugged in at compile time to generate an instantiation of the class with the required implementation.

The syntax is as follows:

```
template <typename ClassName> function(ClassName parameter);
template <typename ClassName> class TemplateClass {
    ClassName memberData;
};
```

At compile time, the templated code is converted to exactly the same raw code as would have originally been written, so there is no performance overhead.

Although first created to solve fairly simple code duplication issues, over time considerably more complex tasks have been accomplished with the use of templates; they have even been discovered to be Turing-complete.

Because almost every class in our library must be made generic between 2D and 3D, some level of template metaprogramming will be required.

Template syntax is often confusing or unintuitive, and there are occasional compiler discrepancies which must be worked around. Problems we encountered with our templating are discussed in section **??**.

5.4.3 Portability

We will be developing under Linux and Windows, so obviously the code must compile and give correct results on both platforms. Ideally, given the same input, we will produce identical output on both platforms, down to the exact floating point values. This will ensure users of software made using our library will have different experiences if they are running on different platforms.

In theory this will be true for any valid C++ program, but in practice it may be necessary to work around incomplete/different implementations of the C++ standard between GCC and the Visual C++ compiler.

We are also limited to using only cross-platform libraries, although for mainstream libraries this was not a problem.

5.5 Development methodology

Our agile project management methodology is discussed in greater detail in chapter 8. This section describes the development conventions and quality assurance routines we had in place.

5.5.1 Conventions for development

Agreeing upon and following consistent conventions is an important step to avoiding simple and common development errors. (Accessing files by the wrong case, variables by the wrong name, and so on). By having a predictable syntax methods and member data were easy to use and specify.

Class naming conventions

CamelCase with first letter uppercase (for example `CattoContactResolver`).

Method/variable naming conventions

CamelCase with first letter lowercase (for example `applyForces()`).

Code formatting conventions

For the most part, K&R 1TBS C++ style was used.

Filename conventions

Header files: Lower case, words separated by underscores
(for example `broadphase_octree.h`). Source files: Lower case, words separated by underscores
(for example `broadphase_octree.cpp`).

Other conventions

- Getters and setters: `getPosition()`/`setPosition(p)`
- Names should be meaningful (except loop variables)

- Private member data should be prefixed with `m_` (e.g. `m_boundingRadius`).

5.5.2 Quality Assurance

Stability of our physics engine was of obvious importance to us. A physics engine is no use to an application if it throws overrun exceptions or segfaults constantly. We used cross-platform exception trapping to hide any exceptions from the application, should any be raised. Force and velocity member data has sanitation code to set them to 0 if they become too small or handle them gracefully if they become too large. Similar sanity checks exist in various places throughout the code base.

5.6 Future expansion

Our extensible design means that broadphase, contact generation, and contact resolution algorithms can be added easily by us or other developers. We did however not have many different broadphase algorithms planned. Had we planned for or ruled out algorithms One limitation of our design is how to implement particle system, and by extension, particle based fluids. At the time of the specification plans for this were not finalised.

Chapter 6

Implementation

6.1 Broadphase Collision Culling

A naive implementation of the broadphase was fairly simple to implement.

One change that was made was to promote the naive Broadphase implementation from an implementation of the BroadPhase interface to the actual BroadPhase interface itself. This was done so that other implementations would not have to implement the helper methods (`findBodyAt` and friends) unless they chose to; by default the naive implementation of these operations would be called.

We originally intended to develop a sort and sweep implementation, but it proved to be significantly more challenging than anticipated. At the time the broadphase was far from becoming a bottleneck, so the decision was made to focus development elsewhere.

During the development of the DOOMinoes game [1] there were many times more active objects at a time than ever previously. The broadphase component became a significant factor in performance. A first approach to overcome this was to write a custom broadphase algorithm exploiting properties of the game (testing the extensibility of the broadphase component). While this improved the situation it did not completely negate the issue. After this, an octree implementation was developed in the library using example code from WGD-Lib. This octree code was later adapted to 2D to provide a quadtree.

[1] A 3D dominoes game made by Alan making much use of particle-like bodies

6.2 2D Contact Generator

The preliminary list of 2D shapes to perform collision detection on was:

- Circles
- Rectangles
- Polygons
- Capsules
- Line segments

All of these shapes have the property that they can be represented as a polygon which can be expanded in all directions by some radius (see diagram 6.1).

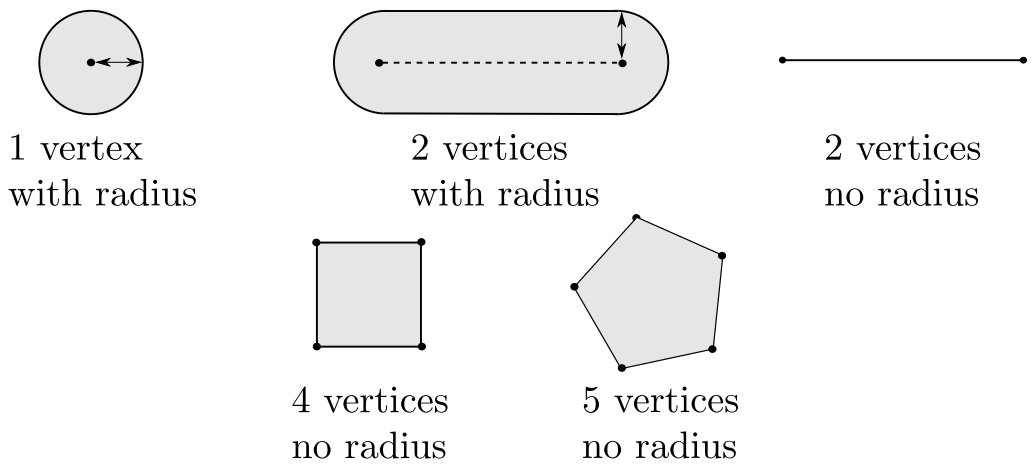


Figure 6.1: Supported shapes represented as a set of vertices plus a radius

We leveraged this property both in the interface for the Area class and for the implementation of the 2D ContactGenerator class.

6.3 3D Contact Generator

In 3D, the only shapes we initially attempted to simulate were spheres and boxes. Sphere-sphere contacts were quickly developed; to get a 3D testing arena running a floor was constructed made up of hundreds of spheres.

The additional tests (box-sphere and box-box) proved more troublesome, though were eventually completed - albeit with subtle bugs that were never fully understood.

6.4 Contact Resolver

The initial implementation of the ContactResolver class was based on code from Alan's third year project, which itself was partially based on a design from Game Physics Engine Development [18].

For every contact, it would calculate the approach speed (from the velocities of the two bodies involved). If this is negative the bodies are moving apart and the contact is ignored. It would then proceed to iterate over the list of contacts again to resolve any penetration.

The order in which you iterate over contacts is an important factor in the stability and running time of the algorithm. Millington strongly recommends always resolving the most extreme (highest velocity or greatest penetration) contact, however another option is just to iterate over contacts in the order they are stored. This saves the need to do an expensive search for the most extreme contact, so more iterations can be run in the same time.

To this end, the storage of contacts was delegated to a new class named `ContactList`. This facilitated the ability to choose which iteration order to use and also the development of new techniques of finding the most extreme contact.

At project conception we assumed any resolution algorithm would resolve both position and velocity together. As we began to find and implement algorithms that changed only position resolution or velocity resolution, a decision was made to split the ContactResolve class into separate PositionResolver and VelocityResolver classes.

Box2D_Lite [2] is an example of such an algorithm. The CattoPositionResolver and CattoVelocityResolver classes are based on his design.

6.5 Sleeping

In a normal dynamic system, most bodies will eventually find a stable, immobile configuration. They will remain in that configuration until some force acts upon them, usually from collision with another object. If we can avoid expending a great deal of computation time on simulating these immobile objects, we can make great gains in step processing speed. These gains are particularly pronounced in a normal game scenario, in which most of the action in a large world occurs in a small area around a single player entity, while objects in the rest of the world remain idle.

Sleeping is a common optimisation to the physics simulation that involves marking

stable objects as 'asleep', so that certain elements of the simulation can ignore them when processing. One such system is described by Millington in [18], which consists of some state-tracking variables added to bodies, and two methods: one to put a body to sleep when it is idle, and another to wake it up when appropriate. In his book, Millington advises the addition of three new variables to the `Body` class:

- `isAwake` — A boolean variable that tells us whether the body is currently asleep, and therefore whether it needs processing.
- `canSleep` — A boolean variable that tells us if the object is capable of being put to sleep. Certain implementations of the engine might require certain objects to be immune to being put to sleep.
- `motion` — A float variable that tracks the current movement speed (both linear and angular) of the object. This is used to decide if the object should be put to sleep.

A new method in the `Body` class (`setAwake(bool)`) also needs to be declared to allow the state of isAwake to be set. This method performs two additional tasks: when the object is put to sleep, it zeroes both the object's linear and rotational velocities; and when the object is awakened, it sets `motion` to some arbitrary high enough value (in our case, 2ϵ; see below) that the body will not immediately fall asleep again.

Bodies are put to sleep when their `motion` falls below some 'epsilon' (ϵ) value, which can be tweaked until the simulation appears correct. `motion` can obviously not simply take its value directly from the body's velocities, otherwise (for example) bodies might fall asleep at the apex of their path when thrown straight up in the air. Also, the velocities are vector quantities, but their direction is of no interest to us for the purposes of deciding that they should or should not become dormant. Millington therefore recommends that `motion` should be a recency-weighted average (RWA) of the sum of the squares of the scalar values of the body's two velocities:

$$motion_{n+1} = b \times motion_n + (1 - b) \times (|v|^2 + |\omega|^2)$$

where v is the body's linear velocity, ω its angular velocity, and b is the bias of the RWA, a number between 0 and 1 that affects how much new input values affect the RWA's value. Obviously, a value of 1 for b ignores all new input, and a value of 0 would simply set the value of the RWA to the new input value. A moderate value (0.5 – 0.9, say) acts to smooth the input motion values and produce a value that reflects recent motion, rather than instantaneous motion. The RWA is also limited to a certain maximum value to prevent periods of high speed movement setting the

RWA so high that it takes too long to subsequently put the object to sleep; in our implementation, this value was 10ϵ.

Dormant bodies are awoken when an awake, non-static object collides with them with sufficient velocity. In practice, this means adding code to the contact resolution[2] methods that wake sleeping bodies before a collision is resolved. This method can cause a 'wave' of awakened objects in a stable structure when one of them is awakened by a collision; such behaviour is correct, since it allows force from the collision to propagate through the structure.

The second case in which a body needs to be awakened is when a force is applied to it directly, so the methods in `Body` that allow forces or torques to be added to an object must check if the object is asleep and, if required, wake it up before continuing.

The saving in computation time gained by sleeping is from not having to solve `applyForces()` and `resolveVelocities()` for sleeping bodies. Code was therefore added to `applyForces()` and `resolveVelocities()` so that they do not execute for a sleeping body, and the contact resolver was told to ignore contacts that only involve dormant bodies.

6.5.1 Refinement

While Millington's system provides a simple sleeping system, we found after testing that it did not handle certain situations well, so we modified elements of its implementation to fix these problems.

One problem was that while objects did eventually sleep, they did not do so satisfactorily quickly in certain situations, particularly if they were rolling spheres or circles with very low velocities. We eventually concluded that we could better tailor the sensitivity of the sleep system by splitting the `motion` value into two values: one RWA for the linear velocity, and one RWA for the angular velocity. This way, we can have two values of ϵ: one for each velocity. We also discarded the squaring of the velocity values in favour of simply taking a positive scalar value of each vector:

$$linearMotion_{n+1} = b \times linearMotion_n + (1 - b) \times |v|$$

$$angularMotion_{n+1} = b \times angularMotion_n + (1 - b) \times |\omega|$$

By far the more serious problem however, was that Millington had neglected a case in which bodies should be woken up. Bodies asleep atop a pile of other bodies were

[2]Contacts that need not be resolved are considered to have insufficient velocity to wake an object.

left sleeping, hanging in the air, when the bodies underneath them ceased to support them. This happened when, for example, the supporting body was deleted, or they simply fell away without colliding with the sleeping body. We solved this by having bodies wake up in two new cases: when a body with which they were colliding was deleted, and when a generated contact with a sleeping body reported that the distance between it and the other body had increased above a certain threshold.

The second solution was not perfect; a stack of non-sleeping bodies, when the bottom body is removed, all fall down at once, whereas a stack of sleeping bodies, since they need a certain separation distance to occur before they awake and fall, fall one after the other in a 'wave' effect. Ultimately, it was decided that this did not substantially damage the appearance of the realism of the simulation[3], and was a reasonable tradeoff for the gains in performance that the sleeping system provides (see 7.4).

A final optimisation was made late in development. It was realised that a significant speed boost could be gained by not checking for collisions between bodies that are both either asleep or static. While Millington holds that these collisions must still be generated – in case one of the objects is woken up by a third – it was decided that it would probably not cause particularly noticeable problems, and that the performance benefits would be worth it. A global setting to toggle this option on and off was introduced so that if it does cause problems, it can be disabled.

6.6 Profiler

The first key part of the implementation of the `Profiler` class was the creation of the `MicroTimer` class to measure execution times. `MicroTimer` stores a value of the number of microseconds elapsed since midnight as a start time when its `reset()` method is called. Thereafter, it can be called upon at any time to return the number of microseconds elapsed since `reset()` was called, which it obtains by subtracting the stored value from the current value of the clock. Through this mechanism, the execution time of a single step (or frame) can be measured. Additionally, `MicroTimer` stores a map, relating strings (method names) to microsecond values, which acts as a lookup for the start times of various methods. This map can be used to time the execution times of an arbitrary number of sub-steps within the processing of a single frame.

During testing, it became clear that the method we were using to fetch the number of microseconds elapsed since midnight (`gettimeofday()`) was not platform-independent: it worked correctly and accurately on Linux systems, but on Windows

[3]Certainly no more than the appearance of sleeping bodies hovering in mid-air.

systems, it was limited to a precision of 0.15ms (150 μs), which was inadequate. After researching the problem, we found a Windows-specific solution using an in-built high-performance timer method that would operate at the required level of accuracy, and could be substituted in for the other method at compile-time.

The `Profiler` class itself provides methods that are called at the start and end of every frame and the start and end of every subtask that needs timing. `Profiler` resets the `MicroTimer` at the start of each frame, and adds markers whenever sub-methods start, then stores the data that `MicroTimer` generates when methods or the frame conclude.

Initially, each distinct dataset was stored in a separate `std::list`: the frame times in one, and the method times in many, each stored in a map, keyed to a string (the method's name). As development progressed, and the Profiler was required to store ever more non-time-related statistics[4], it became clear that this solution was inadequate.

To solve the problem, a new class was created: `ProfilerFrameData`. This object stored all the data relating to a single frame, including its execution time, a map of method names to execution times, and a map of variable names to values (e.g. Body Count). The `Profiler` was then free to simply store an `std::list` of `ProfilerFrameData` objects.

Generating useful output files at the end of the simulation was simple: the `Profiler` opens a file in the working directory and simply writes out comma-separated data (and metadata) to it, then calls a shell command that runs gnuplot according to instructions stored in a batch file in the source directory, which produces a number of useful graphs from the data, suitable for analysis of the simulation (see 7.4 for some examples).

6.6.1 Director

The profiler produced sufficiently accurate data from a given run of the simulation to allow the diagnosis of slow execution times; but we concluded that to produce reasonable comparison data between two different implementations of algorithms, we would have to be able to guarantee that the same simulation was being run each time. Consequently, we implemented the `Director` class in the sandbox. The 'Director' is capable of adding bodies in a specified order to the world at specified times (step numbers, rather than real times), ensuring that the simulation run is the same every time. It is intended to be run in full profiling mode, with graphing turned on, and the sandbox application launched in non-interactive mode, so that

[4]Such as the numbers of bodies and contacts present in the world.

the user cannot interfere with the operation of the simulation.

6.7 Libraries

The only library used in the development of the physics engine component (as opposed to the sandbox) was Boost [6].

Chapter 7

Testing

The nature of a real-time physics engine makes it difficult to test: its joint aims are realism and speed. While the latter can be measured in a reasonably objective manner, the former is both qualitatively assessed and somewhat subjective. The accuracy of the simulation prompted a number of minor disagreements between members of the development team, based on such nebulous notions as whether collisions 'looked right' or whether or not one object should balance atop another.

In particular, the question of whether spheres and circles perfectly aligned along a line parallel to the y-axis should stack or not proved a point of contention for some time[1]. On the one hand, if one adheres strictly to the mathematical model of Newtonian mechanics, it is clear that they should; on the other, given that our aim is to produce a simulation that is realistic *in appearance* for use in games, surely it should follow the expectations of the user, which will be based on their observations of objects in real life, in which such stacks simply do not occur.

The accuracy of the simulation, then, is near impossible to test conclusively; the accuracy of the calculations performed and their application in the various laws of physics, on the other hand, is not. Unit testing of mathematical classes allowed a number of bugs to be eliminated (which in turn improved the appearance of realism).

Beyond the core aims of accuracy and speed, other elements of both the library and the sandbox application had to be subjected to testing simply to prove their utility as software. Both were tested on their compatibility with the intended target systems, and on their usability by their respective probable audiences.

[1] See preset 2 in the sandbox for a demonstration of this in 2D.

7.1 Compatibility testing

7.1.1 Methodology

The collaborative nature of the project meant that development was done on a variety of platforms and architectures. The following operating systems were reguarly used for project compilation and testing as part of the development process:

- Red Hat Enterprise 5
- Ubuntu Fiesty/Gutsy/Heron/Intrepid
- Windows XP SP2/3
- Windows Vista
- Ubuntu Intrepid via VirtualBox

We had planned to test the project on further systems once development was complete, but with such a large list above we decided it was unnecessary

7.1.2 Results

The aforementioned multiple-platform development meant that cross-platform compilation or execution problems were easily traced back to very recent commits and resolved quickly. However when cross-platform problems remained un-noticed for long periods of time, pinning down the exact cause became very difficult. The only example of this we suffered from was a significant difference in performance between Windows-based and Linux-based operating systems (Windows performing less quickly).

The problem went unnoticed as we had no true stress tests (see 7.2 below) before the profiler was developed, so cross-platform performance differences were assumed to be a result of *hardware* differences rather than *operating system* differences. After introduction of the stress-test presets, the differences became pronounced enough for us to query that assumption and investigate (see 7.4.2). This investigation operated under vague notions of time periods in which performance of the simpler presets had deteriorated. With no obvious significant commits that could have caused slow downs, the investigation was arduous and time consuming. The problem was anticlimactically dismissed an issue with compilation settings.

7.2 Regression testing

A set of well-designed presets ensured no regressions; should the addition of a feature impair the operation of another feature it inevitably became obvious in at least one of the presets.

1. Reloads initial sandbox
2. Stack of circles and stack of squares parallel to Y-axis. Stability test. Cause of the aforementioned (7) "does that look right?" discussion.
3. Jenga tower. Stability test.
4. Free-falling box. Box-plane collision test
5. Free-falling 15-base box pyramid, no row spacing.
6. Free-falling 25-base box pyramid, small row spacing. Stress-test for speed of contact resolution: there was a time during development when a pyramid this large would have slowed the simulation to a crawl.
7. Peggle small body stress test. Tests bodies which are fixed to the background but rotate. Tests broadphase.
8. Springs, joints, pendulum, bridges, see-saw
9. Dominoes Realistic-behaviour tests. Used to callibrate default world values, such as friction.
10. 4 blocks on a slope (10th preset bound to key 0) Friction test. Blocks have different friction values so should slide at different speeds.

There exist further presets (beyond those mapped to keys 0-9), including a world 0, which is empty except for the four bounding walls, and is useful when running automated routines. These can be accessed by passing the sandbox the appropriate parameter when running via command-line (see B.2.2).

7.2.1 Methodology

- Project (debug) compiled
- Scenario loaded (or created and saved)
- Results inspected

- Relevant changes made

- Process repeated until developer satisfied with behaviour

7.2.2 Results

This agile method of quick and tight design-development-testing cycles saved a lot of time and allowed us to implement all of our core features and many of our optional features, with some bonus features too.

7.3 Unit testing

Beyond the simple collisions results tested in our presets (Box-Box, Box-Plane, Box-Circle, Circle-Circle) we had few classes where unit tests seemed appropriate. We did use them for vector & matrix classes, which proved useful.

7.3.1 Methodology

- Vector & Matrix classes This class was created to facilitate basic operations on vectors. For example, rather than have vectors added in the form a.add(b), the + operator was overloaded to allow for the "a+b" vector addition syntax to be valid. Outputs from this class were compared against outputs from DirectX-Utility matrix classes.

7.3.2 Results

- Exposed problem with matrix arithmetic.

  The first execution of unit tests for Vector & Matrix revealed that the 2D-matrix multiplication operator had not been implemented at all. This would have later caused problems with all manner of contact resolution code, possibly leading the developer to believe that the problem lies with the new code rather than the matrix class.

7.4 Performance testing

Profiling the performance of various collision-resolution algorithms was one of the aims of the project; the `Profiler` class (see 5.3.3) in the library was designed with this functionality in mind, and provides general-purpose statistics on the speed of each execution step in the simulation process.

The profiler additionally allows us to analyze how the engine handles various usage scenarios, and to identify areas of the code that are causing bottlenecks under certain conditions. We can then alter problematic code, or focus our optimisation efforts on the areas where they will reap the greatest gains.

7.4.1 Methodology

Prior to the completion of the profiler, all performance assessments were purely subjective. Stress tests were added to make the effects of code changes more obvious. When profiling was added we could also compare physics time and draw time for a given frame. Ultimately, graphical output was developed to show performance over time. Weekly tests were then performed to profile new additions to the code.

The profiler can produce data for any simulation task run in the library, but the most valid results for comparing performance will obviously come from running the same (or similar) routines. The sandbox is capable of running a deterministic, non-interactive benchmark through the use of the `Director` class (see 6.6.1), although results were also produced by comparing runs based on preset worlds (see B.2.1), or from using the sandbox's facility to save and load a custom arrangement of objects.

Primarily, however, we used results produced by running the director scripts. Director script 1 is a simple test with 3 phases:

1. Starting from an empty world, circles are 'rained' down in ten vertical lines from high above the floor. This increases the body count quickly and steadily, generating stable towers of balancing circles as it does so.

2. On step 5000, the rain stops, and a 'firework' of 36 small circles shot in every direction is set off in between the bases of the two central towers. This exerts a small force on both towers, causing them to collapse slowly. Debris from these towers knocks over each other tower in turn, and all the structures descend into chaos. This keeps the body count constant, but greatly increases the number of contacts the engine must process.

3. Eventually, all the bodies reach a stable state, and neither the number of bodies nor the number of contacts changes. At frame 10000, the simulation

ends.

This test is both simple – meaning it produced very clean output in which clear trends could be discerned – and comprehensive, since it tests three stress cases: number of bodies and contacts growing equally; number of contacts growing quickly, number of bodies remaining constant; and both number of bodies and number of contacts high, but with little change in the simulation.

Director script 2 is more of a stress test. Every 20 frames it creates a heavy circle in a pseudorandom location with a pseudorandom velocity, before quitting at frame 30000. This quickly fills the world with many bouncing objects, particularly if it is run with gravity set to 0 and default restitution to 1. This test produces less smooth results for number of contacts, and isn't suitable for testing bodies in a stable configuration, but it is perfect for assessing the performance of algorithms that depend on the number of bodies present in the world.

7.4.2 Results

Our performance tests exposed few particular flaws in our implementation, although it did allow us to target our development efforts on those methods that were taking the longest time to execute. The best example of this can be seen in the introduction of the sleeping system (6.5), which helped optimise the previously slowest method for a significant time saving in certain situations.

We also took advantage of performance testing to compare different approaches to certain tasks. While some such approaches had other benefits than performance (Catto contact resolution, for instance), it is nonetheless instructive to consider their performance when analysing their suitability for our engine.

Sleeping

Figure 7.1 shows the execution times for a whole step and two of the sub-tasks (`findContacts` and `resolveVelocities`) in that step, as well as body and contact counts, for all steps in a full run of director script 1.

The three phases of the script have a clear impact on execution time. In phase 1, as the body and contact counts steadily increase, total execution time and sub-task execution time also increase. In phase 2 (which lasts from about step 5000 – step 6000), body count remains constant, but the contact count increases rapidly; this increase is matched by execution times. By step 6000, the bodies in the simulation have settled, and we have entered phase 3, in which the contact and body counts – and execution times – remain constant.

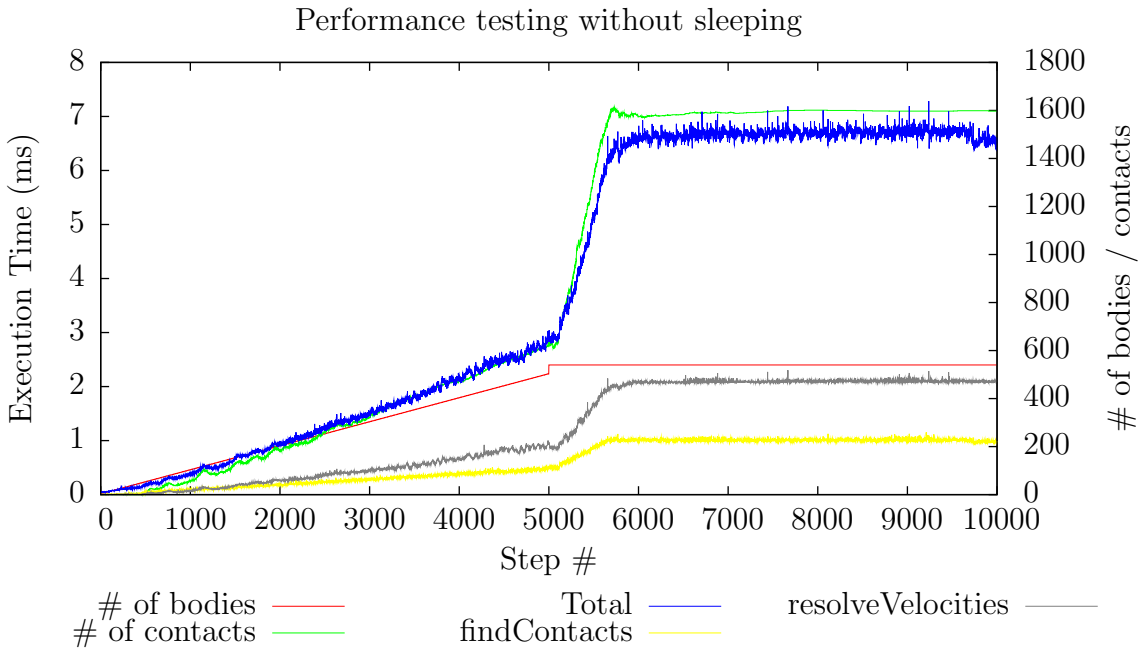


Figure 7.1: Step execution times and body and contact counts for director script 1, without sleeping.

The total execution time peaks at just over 7ms, and remains stable at about 6.5ms as the bodies settle in stage 3.

The following graph (fig. 7.2) shows the same result with the sleeping system enabled.

This execution profile shows a marked improvement over figure 7.1. The total execution time with sleeping peaks at just over 5.5ms, and then drops away quickly to stabilise at around 4ms in phase 3: an improvement of $\approx 20\%$.

The source of this improvement is easy to explain. Throughout the test, bodies in stable configurations – at the bottom of stable towers or sitting in a non-moving heap – will fall asleep. Sleeping objects need not have their velocities resolved (we know they aren't moving), and contacts between two sleeping objects need not be considered. The effect of this obvious from the graphs: the two methods `findContacts()` and `resolveVelocities()` both show steady drops in their execution times during phase 3 – as more bodies reach a stable state and fall asleep – and general reductions in execution times throughout the test.

The performance gain in `resolveVelocities()` is the most pronounced; indeed, it was the identification of that method as the slowest task during step execution that drove us to develop the sleeping system. The performance gain in `findContacts()` is less profound, and potentially sacrifices realism in the speedy awakening of large

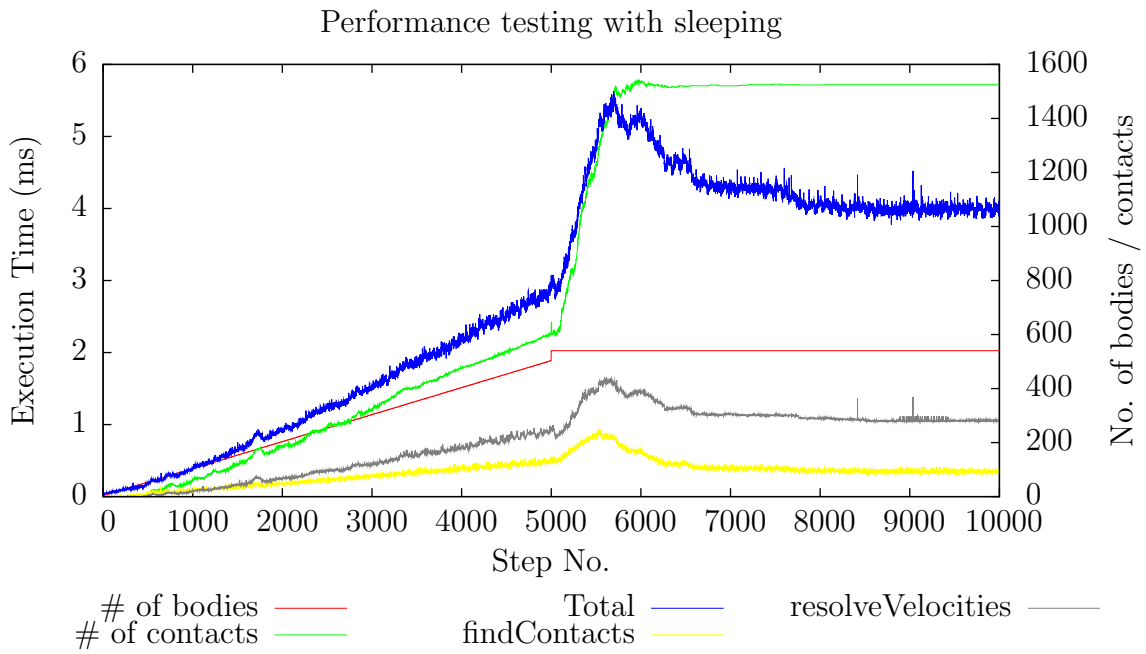


Figure 7.2: Step execution times and body and contact counts for director script 1, with sleeping.

groups of sleeping objects (see 6.5.1). A global variable can be used to toggle this behaviour, allowing a user to maintain realism but still enjoy the performance benefits of sleeping in the reduced computation time of `resolveVelocities()`.

The third method affected by sleeping, `applyForces()` is not discussed here, since its execution time is so low compared to the other methods' in any case (< 0.05ms throughout the test) that any performance boost would be negligible.

Broadphase

In the project, we implemented two algorithms for two-dimensional broadphase collision detection: naïve broadphase and quadtree broadphase (see 6.1 for details). The performance of the two implementations is displayed in figure 7.3.

The sub-task that handles broadphase collision detection is `findPairs()`, so we have only considered the performance of that method for this test. The naïve method (shown by the red data) actually performs better than quadtree for small numbers of bodies($\lessapprox 300$), but that is clearly insignificant against its terrible performance with increasing numbers of bodies. The naïve broadphase implementation runs in $O(n^2)$ time, whereas in this test quadtree shows a more linear relationship to body count.

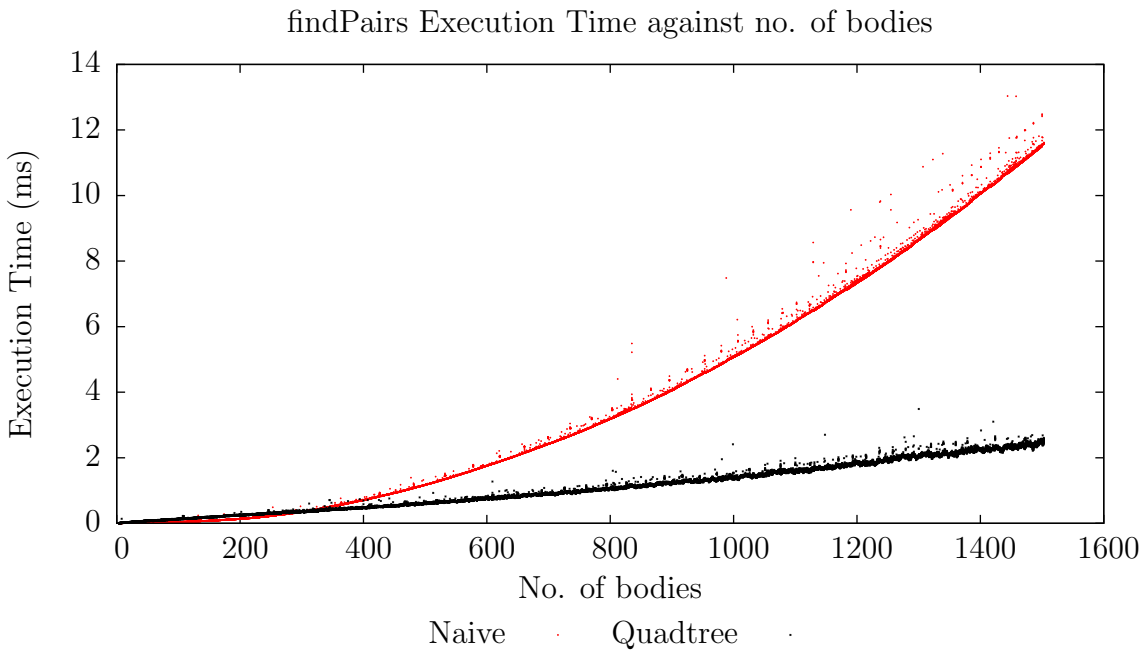


Figure 7.3: `findPairs()` execution time against body count for director script 2

Since the range in which naïve broadphase is superior is so short, and the superiority so insignificant ($\approx$ 0.1ms) in a real-time application, we recommend the use of quadtree broadphase in any use of the library, but naïve is still present should developers want to use it.

Contact resolution

During the course of development, we managed to implement two contact resolution systems. The first was an extension of Alan's work in the third year, which was based on Ian Millington's work [18]. This resolver came to be known both in the code and by us as the 'stupid' resolver. The stability of the simulation under this resolver was inadequate: bodies at rest on others vibrated, and stacks of objects never settled and sometimes oscillated wildly. The second was based on the work of Erin Catto [2], and was considerably more stable, even with large, carefully balanced stacks of objects.

Figure 7.4 shows the performance test results for the 'stupid' contact resolver. It ably demonstrates the impact of poor stability on performance in the number of contacts. The contact count quickly departs from a linear trend following the number of bodies (as it should be in towers of circles, since for each body, there is one contact: the one with the circle below it), as the towers compress and oscillate enough that circles start to generate contacts with the bodies below those they are resting on. Later,

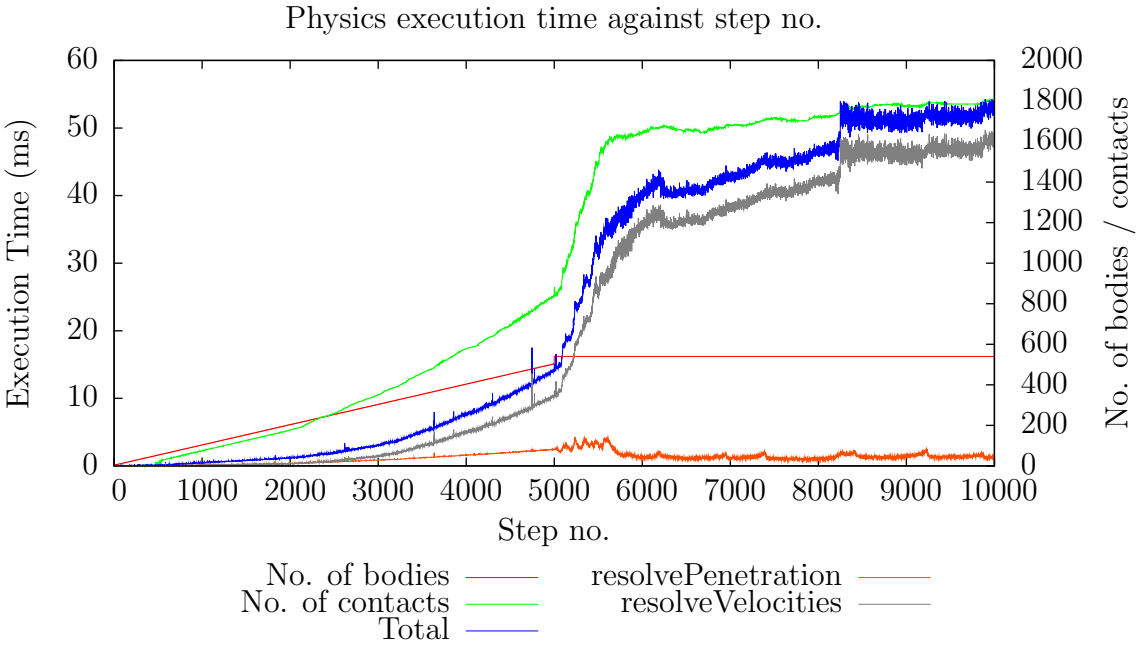


Figure 7.4: Execution profile of the 'stupid' resolver for director script 1

in phase 3, the number of contacts never settles, as the bodies themselves do not: they arrange themselves in a heap, but the heap is not stable.

Critically, this has a large impact on performance, since the vibrating objects never reach a stable enough state to fall asleep, so execution time for `resolveVelocities()` remains high in phase 3, where ideally, it should fall steadily.

Figure 7.5 is a profile of the same test run using the Catto resolver. Notably, the number of contacts here is more stable, so a performance increase from sleeping is realised in phase 3.

More importantly, this resolver is simply much faster. At the point the firework goes off in frame 5000, the execution time using Catto is 3ms, compared to the 'stupid' resolver's 16ms. by the end of the simulation, Catto is processing a step in 4ms, while the 'stupid' resolver is taking $\approx$ 50ms!

Since the Catto contact resolver is both more stable and considerably faster than the 'stupid' one, we unreservedly recommend its use in all applications, however trivial. The 'stupid' resolver is still included, but mostly for educational purposes only.

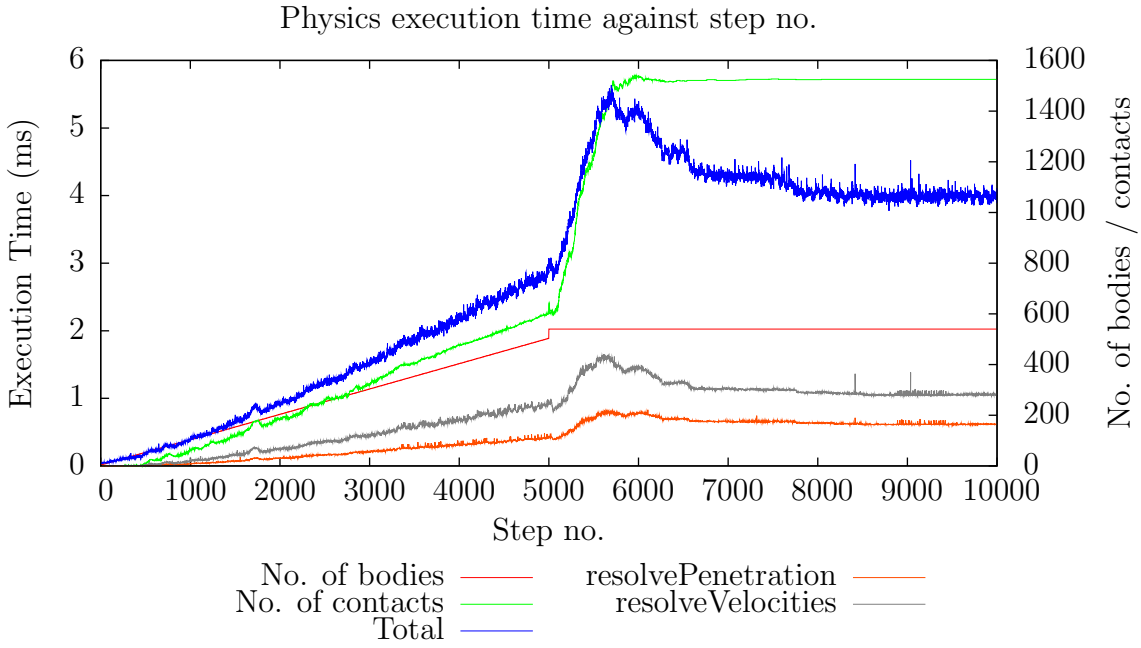


Figure 7.5: Execution profile of the Catto resolver for director script 1

Windows vs Linux performance

Since we specified cross-platform compatibility as one of our aims, it seemed prudent to test our engine's performance on the primary systems it was being used on, as well as performing wider compatibility testing (see 7.1).

Figures 7.7 and 7.6 show the results of these tests. They are profiles of the simulation of a pyramid of boxes at rest. Consequently, the number of bodies is permanently constant, and the number of contacts is constant once the boxes settle. The two tests were run on the same computer, alternately booted into: Windows XP, running a version of the engine compiled by the Microsoft Visual Studio C++ compiler; and Ubuntu 8.10 (Intrepid Ibex) running an engine compiled by gcc.

The profiles are broadly similar: the relative execution times of sub-tasks are proportional to each other and the total execution time; and both show a small increase in performance as the system settles and bodies sleep. However, the simulation run on Windows takes considerably longer to execute all methods: it reaches a stable execution time of $\approx$ 30ms, whereas the Linux system reaches a stable execution time of $\approx$ 14ms, around half as much.

These results are easily observable in general use and in all applications beyond this test: Windows binaries compiled with MSVC 2008 simply run slower than Linux binaries compiled with gcc. We believe that this issue could be resolved by compiling

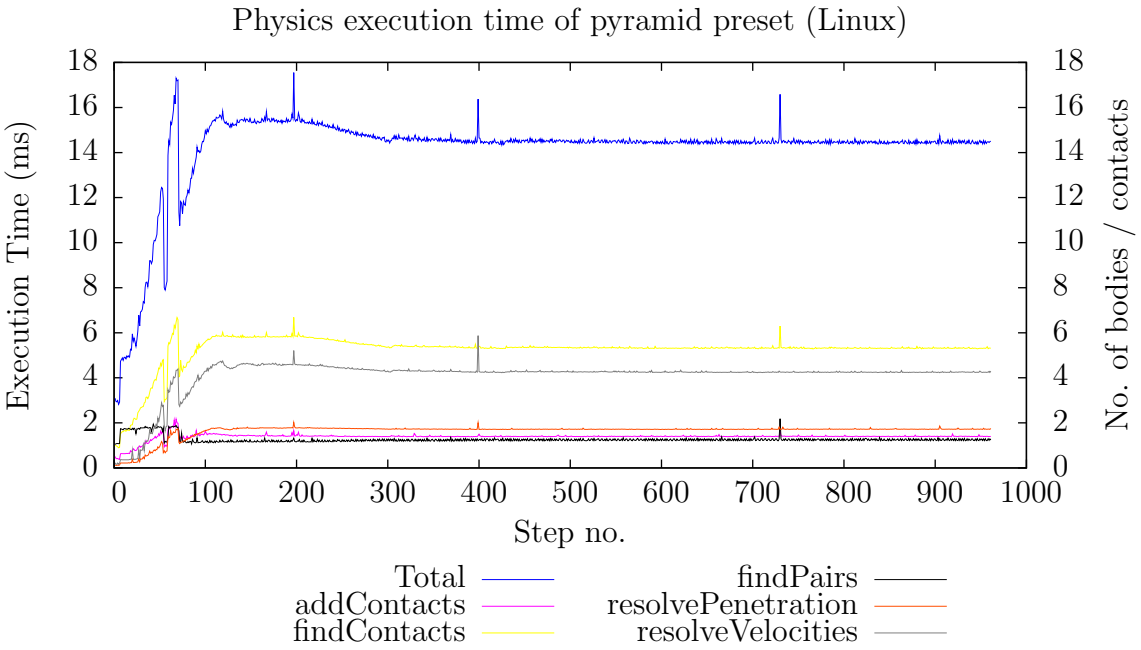


Figure 7.6: Execution profile of a pyramid preset run on Ubuntu 8.10, compiled by gcc

Windows binaries with an alternate compiler such as MinGW, but we have not yet tested this at time of writing.

7.5 Usability testing

7.5.1 Methodology

When user features were added to the sandbox their appropriate tickets were marked as "Feedback" on Redmine. They were then tested by other members for usability and intuitiveness. Semi-formal usability tests were also carried out, as a result of which the preview drawings for Sandbox Circle and Box creation were added.

7.5.2 Results

Two teams used the library in the term 2 Warwick Game Design 48 Hour Competition (27/02/2009 to 01/03/2009). One of these was DOOMinoes (developed by Alan), but the other team actually made a 2D game with essentially no help required and with no bugs found.

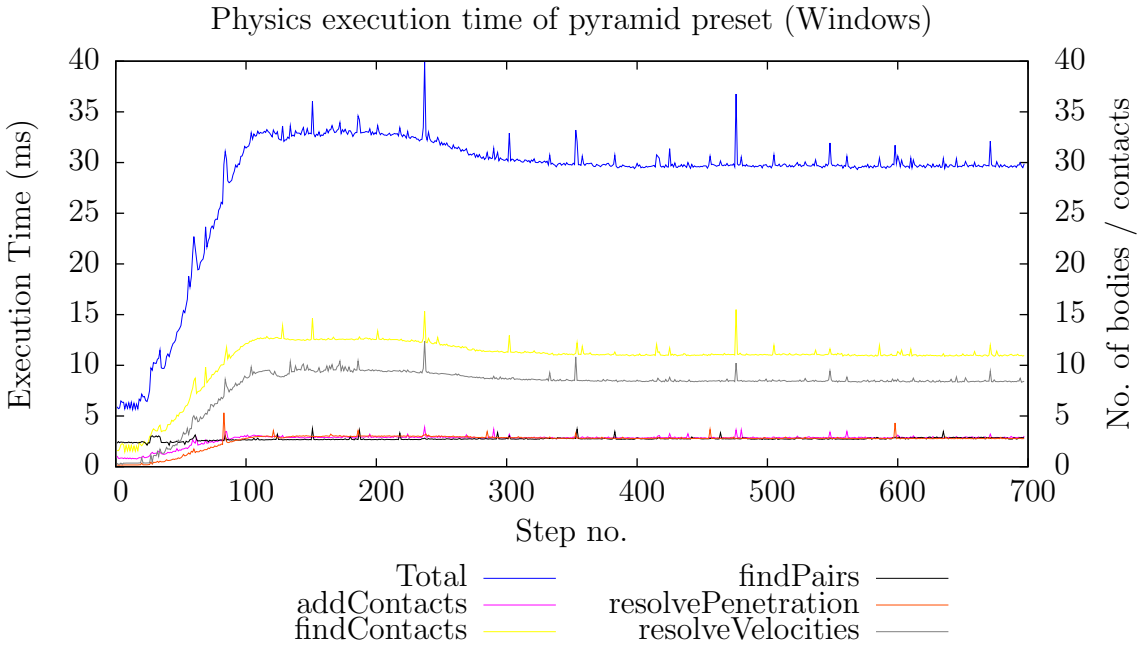


Figure 7.7: Execution profile of a pyramid preset run on Windows XP, compiled by MSVC 2008 compiler

Chapter 8

Project management

In this chapter, we examine how the development process was managed, the roles of each member within the group, and the tools and methodologies used for co-ordination and collaboration.

8.1 Group structure

Documentation Co-ordinator: Richard Falconer
The task of the Documentation Co-ordinator was to ensure that documentation is maintained in a suitable condition for the customer to easily understand and use the product. This individual also carries the responsibility of organising and overseeing the creation of other important documents (such as this report, the specification, and the project presentations), editing them for style, correctness, and consistency, and ensuring that they are completed on time.

Team A:
Leader: Leigh Robinson
Developer: George Stanley

Team B:
Leader: Alan Hazelden
Developers: Richard Falconer, Ben Hallett

The twin development teams were not intended to operate in isolation; rather, they formed small clusters of support, within which each developer first went to their teammate(s) if they needed to discuss an element of the project more immediately

than the next project meeting. Each team could be focused on a different area of the project at a given time, allowing development to avoid serious bottlenecks. The team leaders were responsible for selecting and delegating tasks, ensuring that deadlines did not perpetually slip and alerting the rest of the team to serious issues should they arise.

8.2 Methodology & Development Strategies

The project objectives lend themselves naturally to an agile development methodology, rather than a more static, planned approach. The aim is not to construct some monolithic application that must fulfill an immutable set of requirements. Rather, it is to implement a variety of features (in the form of various algorithms for the simulation of physical interactions) in a highly modular library. Consequently, requirements may change frequently as (for example) a new module depends on unimplemented functionality in another, or one feature is abandoned in favour of a more economical solution.

We therefore selected our development methodology to prioritise both inter-developer communication – to ensure each team member fully understands every aspect of the project – and the flexibility and freedom to alter previous plans when they were discovered to be untenable (or unambitious). We selected project maintenance tools and a modular design that not only allowed us to collaborate to achieve the various project aims, but granted us the extensibility to go beyond them if desired. Above all else, we felt it critical to our development process that rigorous testing was performed at regular intervals to allow us to identify problems at an early stage.

8.2.1 Developer communication

Agile software development methodologies heavily emphasise daily face-to-face communication between project members to keep everyone abreast of issues in the development, and ours was no exception. Project members met and updated each other on the project every day of the week; this process was augmented through regular virtual contact via e-mail and instant messaging software. We additionally organised more structured bi-weekly meetings, at which all group members were present, to assess and solve any problems that may have arisen. The meetings also served to ensure compliance with the schedule and objectives, and – after the first (poster) presentation – became lengthy coding sessions to speed development through techniques like pair programming. This process allowed issues to be raised at the start of a meeting, a solution developed, and feedback gathered before the end of the meeting.

Minutes

Our Documentation Co-ordinator took on the task of recording the minutes of group meetings, and making them available on the subversion repository (See 8.2.2, below), including issues discussed and their conclusions. This was useful to those not present at meetings, and to the group as a whole as reference material when we were acting on the result of a discussed issue. Issues to be raised at the next meeting were also recorded. These were either carried over from previous meetings or added to the repository when we found an issue that should be discussed as a group.

Tracking attendance at project meetings in the minutes allowed us to keep track of who was up to speed on project development and who needed to be updated when there were significant changes to some aspect of the project.

Tickets on Redmine (See 8.2.2, below) were updated in accordance with meeting minutes at the end of meetings.

8.2.2 Collaboration Tools

Co-ordinating the collaborative authoring of code by five group members simultaneously working on a sophisticated software engineering and documentation project gives rise to some not insignificant problems in organising and making available the work a given project member has done. Fortunately, however, these problems are also not insurmountable: there exist established tools for such collaboration, and by making use of them, much of the administrative overhead of the project was relieved.

Subversion

Subversion [5] is a version-control system designed for managing large projects with a great deal of code. It is able to merge two distinct versions of files modified separately by group members into a single, complete file (or, failing that, to highlight the differences between them to ease the task of resolving the conflict manually). It serves as a backup system, and allows one to roll back to previous versions if a newer one does not work for some reason. It (or something like it) is frankly essential for any group development effort.

Subversion is one of several competing technologies, other notable ones being CVS and GIT. CVS was discounted because of its age; Subversion is newer and provides more functionality, especially with regard to conflict resolution. GIT is newer than SVN and - arguably - more powerful. It is complicated to set up, however, and

poorly supported by the other tools being used on this project (such as Redmine, below, and project members' IDEs). Ultimately Subversion was chosen because it strikes the correct balance of functionality and simplicity for this project.

Redmine

Redmine [16] is a project management web application. It provides a group with tools to track both the development of features towards certain development milestones, and the emergence of bugs or problems with the project. It implements a forum for informal group communication and a wiki for more permanent developer documentation. It also integrates with the subversion repository, providing a more human interface for browsing the repository for specific files and viewing the most recent edits. Redmine can be configured to provide each project member with updates in real time via e-mail or RSS/Atom feeds when relevant changes occur to the project.

This web-app was vital to co-ordinating development efforts and preventing feature creep. While we obviously wanted our engine to support as much as possible, and our sandbox to be as usable as possible, we did not want to impair development of essential features by spending time developing smaller minor features. Tasks were therefore assigned to particular milestones and given appropriate priorities. Milestones were worked on in a strict order – so the features belonging to a particular milestone were worked on before the features of a later milestone – and features within the current milestone were nominated and delegated for development by the team leaders. Assigning tasks to a milestone was generally a trivial process, however in the few cases where there were disagreements on the priority of a task, it was resolved in the next meeting.

Initially we had planned to use the calendar features of Redmine to distribute the dates of meetings or deadlines, however it soon became apparent that it was easier to simply set fixed bi-weekly development days (Mondays and Wednesdays) for project work, and to use these meetings to remind project members of upcoming deadlines.

8.2.3 Testing

The most important element of any agile development methodology is frequent unit and performance testing. Without such testing, it is impossible to identify problems with the code (and thus with the objectives as they stand), and the advantage of flexibility that the methodology provides us could not have been realised. Therefore, we required that every component, once developed, was fully tested before being committed according to some test cases decided by the developer themselves, and

that each milestone was tested by the group as a whole, and the results reviewed in a weekly meeting. Once again, the modular nature of the project facilitates such testing, since small components can be isolated and tested to allow for the swift detection and location of errors and performance bottlenecks.

See the chapter on testing (7) for a full description of the methods and results of this process.

8.2.4 Disadvantages

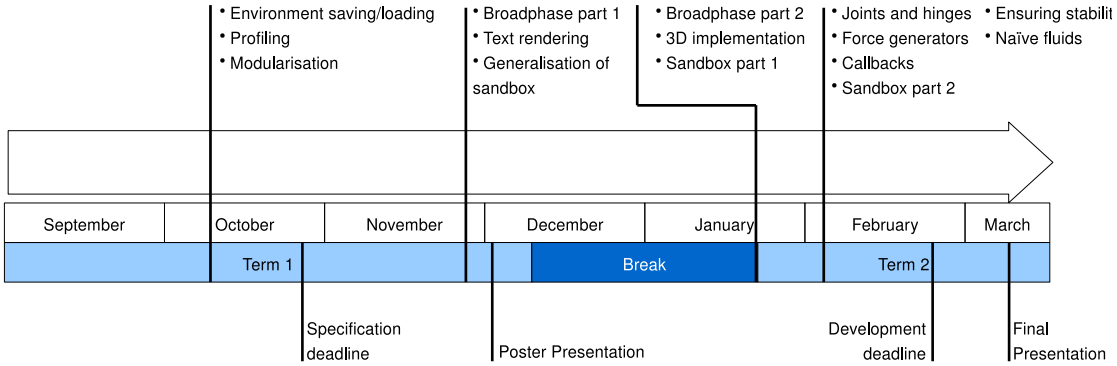
The principle drawbacks to this style of development – apart from the heavy time burden incurred by regular testing – were twofold. First, frequent face-to-face communication detracts from the need to maintain formal documentation throughout the project (indeed, many agile methodologies actively discourage it), but our project required such documentation for the end-product to be useful. Second, the ability to embellish and extend plans in the middle of development allowed for a tendency towards feature creep, which had the potential to (and occasionally did) distract us from our primary objectives.

Both of these problems were addressed within our strategy. For the former, documentation was considered a primary objective as important as any feature, and had a team member dedicated to its maintenance. Team members were encouraged to document features as they were coded, and the entire documentation was reviewed at the conclusion of each development milestone to eliminate any errors or omissions. Less-formal documentation, such as the information relayed between developers to maintain an understanding of others' work, was largely conveyed verbally, in keeping with our agile philosophy. That which could not be easily remembered or articulated was recorded in either the project wiki or in comments in the source code itself.

In the case of the second problem, our group organisation and collaboration methods and software (See 8.2.2, above) encouraged the creation of formal lists of feature requests that team leaders could select the most important tasks from to prioritise their completion. On the rare occasions on which project members did depart from the primary objectives, they were reined in by other members in the next meeting. The potential to cause damage to the project by feature-creep and abortive development is minimal in any case; thanks to its modular nature, a given new feature is unlikely to break older ones, and a single unimplemented feature does not render the entire product useless.

8.3 Timeline

Our poster presentation included the following development timeline, which describes the basic milestones we had laid out for our development efforts:



The deadlines for various features presented herein had been modified since the original Gantt chart we submitted in the specification, as we mentioned at the time: other commitments had forced us to push back the development of certain features to term 2. We realised that our working regime had been too lax in term 1, and expanded our bi-weekly meetings to become development sessions to combat this.

This enormously improved our development speed and allowed us to meet both the mid-term milestone and the final development milestone for every feature except an implementation of naïve fluid simulation, which was an optional feature that was discovered to be too complex (and therefore time-consuming) for our development schedule. See Chapter 9 for a more detailed analysis of which features were not implemented, and why.

8.4 Legal & licensing issues

The group did not build LPC from the ground up; it re-uses others' code for certain tasks not central to the project. We include a number of open-source libraries to perform various tasks in both the sandbox and the library itself. Algorithms and even code snippets have been included from books and the 'educational' physics engine Box2D. We must therefore ensure that we have complied with the license agreements of these pieces of software to remain within the law. For full details of exactly which libraries, algorithms and code snippets were used for which purposes, consult the Design chapter (5).

Both Box2D Lite and TinyXML are released under the zlib license (full text available at [17]). This license permits us to make use of the software and its source code in

any way, provided we do not misrepresent its origin, which – needless to say – we have not.

The SDL and FTGL are released under the GNU Lesser General Public License (full text available at [10]). This license requires that we either link to the library as a shared object, or to agree to provide the source code to our application to anyone who requests it for at least three years after its release. It also requires that to distribute ('convey') the library we must provide a copy of the LGPL and GPL with it. We chose to simply use both libraries as shared objects, and not to distribute them ourselves. Instead, the user manual (Appendix A) directs users to the libraries' respective websites to download the source (and licence agreements) from there.

Since our use of the software under these licenses does not impose any restrictions on our decision on how to license the project as a whole, we elected to release LPC under the zlib license (see Appendix C), permitting end-users to use, alter, improve and redistribute it as they see fit, provided they correctly represent the origin of the software.

Chapter 9

Evaluation

Here we discuss our implementation of features and justify the omission of any features mentioned in the specification.

9.1 Core Features

9.1.1 Collision of primitive and compound shapes

As predicted, once we had collisions of basic shapes completed, combining them together as compound polygons was a simple step giving us a lot more expressibility. With the exception of some stacking and stability issues (mentioned throughout this section) we are very satisfied with primitive basic and compound shape handling.

9.1.2 Modular Architecture

A modular design was one of the project's main goals. To this end a lot of the early design time was spent on it.

We succeeded in making every major component pluggable, so that a user could replace them if necessary. However, in some situations this requires an extensive knowledge of the internal library structure.

9.1.3 Broadphase Collision Detection

The development of the quadtree/octree gave a significant speedup and is an obvious improvement over the naïve $O(n^2)$ algorithm. Additionally the user can easily extend the system to implement any kind of application specific broadphase using domain knowledge.

9.1.4 Narrowphase Collision Detection

The naïve 2D narrowphase collision detection routines have been proven to be robust and fairly efficient (since the running time is $O(E + V)$, for bodies with E edges and V vertices) on bodies that are relatively simple. It should be noted that this algorithm will work for *any* convex polygon and as such 2D collision should be considered complete.

The 3D narrowphase collision detection however is largely incomplete, offering only sphere-sphere, sphere-box and box-box collisions - of which box-box doesn't perform all the required calculations to catch all collisions (edge-edge collisions are incorrectly handled) however for some limited applications this may not be a major concern.

9.1.5 Collision Resolution Algorithms

The original naïve velocity resolver produced initially acceptable results but over time the simulation became unstable.

Developing a contact resolution implementation based on Erin Catto's Box2D_Lite example code solved the issue, indicating that the original model and/or implementation was not mathematically sound.

The naïve position resolution method is woefully inadequate because it doesn't affect the orientation of the interpenetrating body. This leaves the positions in the next timestep incorrect and introduces significant instability in resting contacts.

Using Catto's resolver has two position resolution strategies, *Baumgarte* correction and *split impulse* correction. Baumgarte correction imparts too much springyness into the contacts, which in most cases adversly affects stability of high speed contacts. It was found that split impulses completely solves this issue and achieves dramatic improvements in both stability and efficiency.

9.1.6 Profiler

The profiler achieved its aims as stated in the specification: it allows per-component performance statistics to be generated, collated, and output textually or graphically for analysis. Its precision is high enough that we can distinguish subtle timing differences between different executions of code, and it is capable of handling an arbitrary number of arbitrarily-named sub-task timings and data regarding the state of the simulation.

The use of gnuplot to produce graphical output was reasonably successful: despite learning its syntax and creating batch file instructions for it being a non-trivial development exercise, its functionality ultimately saved us a great deal of time in generating correctly-formatted graphs for output (see 7.4). Unfortunately, if gnuplot is not installed, not located on the PATH, not able to find the batch file, or is an older version that does not support all the commands used in the batch file, all the `Profiler` can do is return an error and an apology. Additionally, while the csv files were generated with column headings describing the data in each column of the file, gnuplot is unable to read these, instead requiring users to refer to columns by number. So, if the number of datasets tracked (and output) by the profiler is changed (or re-ordered) by a developer for any reason, the gnuplot file will need to be altered to reflect the relevant datas' new positions.

Fortunately, most users do not require the graphical output, and those that do can set it up with minimal difficulty. In fact, by default most of the profiler's functions are disabled: it does not output anything, and does not store any data other than the most recent `ProfilerFrameData` object, which is used to return immediate on-screen statistics.

9.1.7 Collision callbacks

Our callbacks were designed and implemented concurrently with Alan's DOOMinoes program. As a result of this they have been throughly tested, and generalised design should mean they can be used with other programs.

9.1.8 Force/torque generators

Force generators (e.g. springs) were implemented and work correctly with the limitation that excessive forces may throw unhandled exceptions. Torque generators (e.g. self-rotating wheels) were not implemented. Time was already running short when development on these began, and with free-pivoting wheels being sufficient for a Car-like demo to be created, work on Torque generators never went ahead.

9.1.9 Simulation visualiser

This is discussed in the Sandbox extension of the originally envisaged visualiser, discussed below (9.2.2).

9.1.10 Stable simulation of objects

Our stability varied significantly as algorithms were tweaked, changed, and added. Once our initial presets became stable, more adventurous versions replaced them. Our final result has a range of stabilities in different situations;

Stable

- Preset 2, towers of circles and boxes, is stable with sleeping enabled.
- Preset 3, dominoes tower, is stable. The outermost pieces wobble until slept. The structure is still stable with sleeping disabled.
- Preset 6 is spongey, but stable

Unstable

- Preset 5 (falling pyramid with no layer spacing) explodes, as inter penetration between layers builds up until the contact generator generates invalid contacts and resolution occurs outwards. This happens as the resolution of contacts between the nth layer and the n+1st layer worsens the severity of penetration between the n+1st layer and the n+2nd layer, and this propagates upwards.

(The remaining presets do not test for stability)

9.1.11 Customisability

Our design pushed all user variables to our Globals class. From here a large range of compile-time options can be specified:

- Sleeping (on/off, linear/angular velocity thresholds, and more)
- Air resistance parameters (Although off by default, a limited implementation of linear and quadratic air resistance exists (9.4.4), and its options are set here)

- Default density/coefficient of friction/coefficient of restitution for newly created shapes
- Default value of gravity (in y axis)
- Default length of a simulation step
- Contact options:
  - Caching of contacts
  - Finding of multiple contacts
- Restitution (On/off, threshold)

For specific globals usage, see A.3.6.

9.1.12 Dimension-agnostic core features

The use of C++ templates worked to minimise code duplication between 2D and 3D and so from that perspective was a success.

However, obscure template syntax confused some developers and made tracing control flow more complex. It also hindered the development of new features.

Almost all features worked equally in both dimensions: however 3D collision detection was never completely developed due to increasing complexity.

9.2 Optional Features

These were features we appreciated would be complex to develop and prime candidates for omission should we be short on time.

9.2.1 Joints and constraints

Implemented late in the project, joints and constraints are not as complete as the core features.

Only the most basic type of joint has been implemented. However, this is sufficient for most tasks in 2D. There are various unimplemented types of joint in 3D, but some could be approximated by two or more of these basic joints.

Cunning use of these joints can simulate swings, chains, rope bridges and simple vehicles.

9.2.2 Sandbox

Our goals for the sandbox were initially modest; an uninteractive application to confirm realistic physics behaviour. These goals were later significantly expanded for two reasons:

- to better demonstrate the capabilities of the physics engine
- to accelerate testing of specific features

The fully-featured sandbox application allowed compile-time and run-time construction of test scenarios. A collection of compile-time scenarios ("presets") are provided and mapped to the keys 0-9. For explanation of what behaviours these tested, see 7.2. Run-time construction further allowed:

- custom or random bodies to be added after initialisation
- precise positioning of bodies (difficult to specify co-ordinates with compile-time scenarios)
- precise orientation of bodies
- pausing, modification of live scenario, unpausing

Anything constructed at runtime could also be saved to disk in a single keystroke. This facilitated the rapid unit testing behaviour described in 7.2.1.

While parts of the user interface are shared between the 2D and 3D versions of a sandbox, some sections were either too complex to be implemented in 3D (e.g. 3D shape manual drawing) or were not appropriate at all (e.g. drawing code for 2D joints).

9.3 Missing features

9.3.1 Fluid dynamics and/or soft body simulation

Our planned implementation of 2D fluids was to use a set of particles. However the particle system to handle these and intra-particle forces. As no particle system was developed this idea did not go ahead.

9.3.2 Advanced narrow-phase collision detection: arbitrary meshes

Not all shapes can be easily represented as a union of primitive shapes. A more general approach would be to allow more complex shapes, such as convex polygon meshes. A naive implementation of these was looked into, but the maths was simply prohibitively complex. It would also likely give unacceptable performance, so we would have to turn to more advanced algorithms for computational geometry such as GJK or VClip. These would have required a significant amount of work to both research and then to implement. As such they were not implemented. A computationally cheap alternative in 2D was to create bodies comprised of multiple separate convex polygons, without taking their union. Existing convex collision detection and resolution could then be performed on individual convex polygons, with any resultant forces applied to the possibly concave body via the concave shape involved in the collision.

9.4 Additional Features

After the specification was submitted, we found it tempting to introduce certain features that did not appear in it. In part, this was feature creep, but some proposed features were discovered to be required for the implementation of specified features; others were true (or truly necessary) optimisations of the specified system.

There were, however, a number of features we discussed which were dismissed before being implemented, or discontinued shortly after implementation had begun. In the latter case, development was halted generally because it became apparent that the feature was unnecessary, would take more work than was previously anticipated, or would not be feasible for other reasons.

9.4.1 Particle system

A particle system was an additional feature we thought we would implement. It would have paved the road to an implementation of fluids. However this was not finished purely due to time constraints and so was cut from the final project.

9.4.2 Time-of-impact Collision Detection

This modification to collision detection prevents fast-moving bodies from passing straight through other bodies. Under normal circumstances this can occur if the body's position is not colliding at time step n and completely passed the 2nd object at time *n+1*, so no collision is detected. Time-of-impact (or continuous) collision detection projects rays from moving bodies in the direction of their travel. The intersection point of ray and potential collision objects can then be calculated and the time of impact found. If we did implement this, it would have been likely only for particles. As we did not implement a particle system, this feature never entered development.

9.4.3 Sleeping

The sleeping system was proposed after profiling revealed `resolveVelocities` to be a major delay in the calculation of a frame. It was hoped that by introducing a method of marking bodies to be ignored during calculation, performance could be greatly improved without sacrificing any realism.

As described in the section on performance testing (7.4.2), our final implementation of the sleeping system worked correctly, and was able to provide performance gains of around 20% in testing, without harming verisimilitude. By altering the parameters of the sleeping system – such as the bias of the recency-weighted averages used and the two (linear and rotational) ϵ values – it is possible to tailor it to be much more aggressive, which produces further gains in performance. At that point, however, the effects of sleeping become more visible to the user, as bodies fall asleep when moving slowly, rather than just when stable.

Elements of the sleep system could stand to be refined further: bodies do not awaken immediately when their support falls away, and sometimes bodies are awakened by very gentle collisions that should not have disturbed their dormancy. The first issue affects the appearance of realism in the engine; the second, its performance (more sleeping bodies means shorter calculation times). The gains to be made from both, however, are slight, and future development work might be better spent elsewhere.

9.4.4 Air resistance

Development on air resistance began to prevent circles rolling forever on an even surface. (Friction is not responsible for rotational retardation). For such a simple problem the required development time was inappropriate, especially if we wanted to keep fluids on the horizon as a future development after the project deadline. As

such a more general implementation was begun, but abandoned before completion in favour of objectives that appeared in the specification. It is turned off by default but remains in the code, with the circle-rotation issue remaining.

9.4.5 Soft bodies

Deformable shapes. Requires mesh-mesh collisions in 3D, and was generally complicated in 2D. Being a low priority and a jump in difficulty, it was never a serious proposition and as such did not get implemented.

9.5 Documentation

Our documentation process increased in complexity almost as fast as the development of the project. Umbrello was replaced for UML design (although used for specification), which sent us on a search for a replacement tool. We subsequently tried to utilise metaUML, boUML, argoUML, Eclipse UML Modeling Framework, and OpenOffice(). None of these tools that can attempt to create some of the UML by parsing the source code completely failed when attempting to deal with our complex templated design.

A compromise was reached with the diagram authoring tool: Dia ([27]), although not automatic it provided an intuitive interface for manual UML generation. It was used for Final and intra-team UML.

With Leigh being the only member with significant experience using LaTeX, documentation formatting was initially slow. The group also had some difficulty establishing a cross platform tool chain for the inclusion of images - a common problem when utilising LaTeX. A system was eventually designed to include vector images as text only, allowing LaTeXto render them with our formatting. Inkscape (A popular opensource illustration package akin to Adobe Illustrator) with the TextExt plugin [29] was the start of this chain - and produced all the illustrations within this document.

9.6 Conclusion

Overall we are more than happy with how the project has come together, despite the problems mentioned. A regret that was mentioned multiple times amongst group members was the lack of time spent creating demos. We have the features to create

a fairly decent rag-doll (once a means to disable self-self collisions is developed, which is quite trivial) and yet we have no demonstration of it. The same applies This modification to collision detection prevents fast-moving bodies from passing straight through other bodies. Under normal circumstances this can occur if the body's position is not colliding at time step *n* and completely passed the 2nd object at time *n+1*, so no collision is detected. Time-of-impact (or continuous) collision detection projects rays from moving bodies in the direction of their travel. The intersection point of ray and potential collision objects can then be calculated and the time of impact found. If we did implement this, it would have been likely only for particles. As we did not implement a particle system, this feature never entered development. for more complex mechanical systems (clocks, zips) and similar things found in sandboxes such as Box2D [2].

9.6.1 Project usefulness

Large Polygon Collider has not yet been integrated into the Warwick Game Design Library, but it has been used in several games independently, including DOOMinoes and at least one other Warwick Game Design 24-hour-game game. Thus seems to fulfill the main requirement - it is flexible enough to be used in other projects.

9.6.2 Future work

The limitless improvements that can be made to physics engines means that there's always something that could be implemented that would make feasible a new simulation type (like recently achieved with fluids), or just generally improve the overall efficiency. Were we given another devlopment cycle to extend ours, the logical first step would be to develop the 'optional features' which were not included. These are mostly within our ability but discounted due to lack of time. There are numerous methods that would theoretically improve stability most notable of which is *shock propagation* - a method by which conservation of momentum is enforce far better (see [9] for details). An obvious feature that we didn't have time to implement is fluids. The design of the library should be able to be extended top support *particle based fluids* in a reasonable amount of time - this would open up a new avenue to explore bouyancy forces and other water based phenomenon.

Our work this year has led to an increased knowledge, enthusiasm and understanding of physics simulations - and we all feel that it was a worthwhile project to undertake.

Bibliography

[1] Richard Burden and J.Douglas Faires. *Numerical Analysis*. Brooks Cole, 2001.

[2] Erin Catto. Box2d homepage. `http://www.box2d.org/`.

[3] Erin Catto. Iterative dynamics with temporal coherence. `http://www.gphysics.com/?page_id=5`.

[4] Marshall Cline. C++ faq lite: Inline function. `http://www.parashift.com/c++-faq-lite/inline-functions.html`.

[5] CollabNet. Subversion: an open-source version-control system. `http://subversion.tigris.org/`.

[6] Beman Dawes, David Abrahams, and Rene Rivera. Boost c++ library homepage. `http://www.boost.org/`.

[7] David Eberly. *Game Physics*. Morgan Kaufmann Publishers, 2003.

[8] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Publishers, 2005.

[9] Kenny Erleben. *Stable, Robust, and Versatile Multibody Dynamics Animation*. PhD thesis, University of Copenhagen, 2004.

[10] Free Software Foundation. Gnu lesser general public license. `http://www.gnu.org/copyleft/lesser.html`.

[11] Arnulph Fuhrmann, Gerrit Sobottka, and Clemens Gross. Distance fields for rapid collision detection in physically based modeling. *In Proceedings of International Conference Graphicon*, 2003.

[12] Lo-Fi Games. Scythe physics editor homepage. `http://www.physicseditor.com/`.

[13] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 1988.

[14] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics*, 2003.

[15] Phyar Lab. Simple physics engine (spe) homepage. `http://spehome.com/`.

[16] Jean-Philippe Lang. Redmine: an open-source project management web application. `http://www.redmine.org/`.

[17] Jean loup Gailly and Mark Adler. zlib license. `http://www.gzip.org/zlib/zlib_license.html`.

[18] Ian Millington. *Game Physics Engine Development*. Morgan Kaufmann Publishers, 2007.

[19] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 1998.

[20] Russell L. Smith. Open dynamics engine (ode) homepage. `http://www.ode.org/`.

[21] James Stewart. *Calculus*. Brooks Cole, 2002.

[22] Matthias Teschner, Bruno Heidelberger, Matthias Muller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. *In Proceedings of Vision, Modeling, Visualization VMV'03*, 2003.

[23] Lee Thomason. Tinyxml main page. `http://www.grinninglizard.com/tinyxml/`.

[24] David Towers. *Guide to Linear Algebra*. Palgrave Macmillan, 1988.

[25] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, 2004.

[26] Various. Collada homepage. `https://collada.org/mediawiki/index.php/Main_Page`.

[27] Various. Dia homepage. `http://projects.gnome.org/dia/`.

[28] Various. Doxygen homepage. `http://www.doxygen.org/`.

[29] Various. Textext homepage. `http://www.elisanet.fi/ptvirtan/software/textext/`.

[30] Various. Umbrello homepage. `http://uml.sourceforge.net/`.

[31] Thomas Williams and Colin Kelley. Gnuplot homepage. `http://www.gnuplot.info/`.

[32] Hugh Young, Roger Freedman, and Lewis Ford. *University Physics*. Addison Wesley, 2007.

Appendix A

User manual

A.1 Introduction

This supplementary document describes the compilation and use of the Large Polygon Collider physics library. Short code examples are provided to get you started with a simplistic world, as well as explanations of further public methods to allow you to create more advanced scenarios.

A.2 Compiling & Installation

There are currently no precompiled binaries, so users of the library must first compile it.

A.2.1 Windows

Visual Studio

- Download the source code from the Subversion repository at `http://svn.draknek.org/torquetome/trunk/physics/`.
- Download Boost
  - Go to the Boost webpage: `http://www.boost.org/users/download/`
  - Download the latest packaged release.
  - Extract the archive to somewhere sensible (for example C:\Program Files).

- Tell Visual C++ where to find Boost
  - Start Visual C++ and go to Tools → Options → Projects and Solutions → VC++ Directories.
  - From the drop-down box in the top-right corner, choose 'Include files'. Add the root Boost directory (for example C:\Program Files\boost).
  - Click OK and save these changes.
- Open the project
  - Navigate to the physics\build\VS2008 directory.
  - Open the Physics Project.sln or Physics.csproj file in Visual C++.
  - Compile the library only. Note that the sandbox will not compile without further libraries (see Appendix B).

A.2.2 Linux

- Download the source code from the Subversion repository at `http://svn.draknek.org/torquetome/trunk/physics/`.
- Download Boost
  - In Ubuntu, you can do this with `sudo apt-get install build-essential libboost-dev`.
  - If you do not have installation privileges, download the latest release from the Boost webpage: `http://www.boost.org/users/download/`. Then extract the boost subdirectory from the archive into physics/include/
- Open a terminal and navigate to the physics directory.
- Run `make lib`

A.3 Using the Library

A.3.1 Hello PolygonWorld!

1. Including library

```
#include "physics.h"
```

2. Creating a world. Every LPC program begins with the creation of a world object.

   ```
   World3D* world = new World2D()
   ```

   Or for 3 dimensional world:

   ```
   World3D* world = new World3D()
   ```

3. Adding your first Body

   ```
   Body2D* myBody = new Body2D();
   myBody->add(new Circle(Vector2D(0,5), 1f));
   world->add(myBody);
   ```

   or, for a 3 dimensional world:

   ```
   Body3D* myBody = new Body3D();
   myBody->add(new Sphere(Vector3D(0,5,1), 1f));
   world->add(myBody);
   ```

4. You're all set!

A.3.2 Worlds

Worlds contain all your bodies, joints and force generators (e.g. springs). See the A.3.1 for how to create them.
As with all objects created with *new*, worlds should be deleted when finished with:

```
delete myWorld2D;
```

For the most part this will not be necessary as typically World will last until your program termination.
Worlds have no boundaries, so it is likely you will wish to create your own using static shapes. If you don't, bodies you add to the world will accelerate due to gravity and fall forever. The below code adds a floor along the origin;

```
myWorld2D->add(Body2D::createStatic(new Rect(-100, 0, 100, 0)));
```

for 2D worlds, or for 3D worlds:

```
myWorld3D->add(Body3D::createStatic(new Box(Vector3D(0, 0, 0), 100, 5, 100)));
```

The *World::add()* and *World::remove()* methods are generic, and can be used to add/remove Shapes, Bodies, or ForceGenerators to/from the world.

World parameters

```
World (float timestep, BroadPhase<d>* broadphase,

ContactGenerator<d>*, ContactList<d>*,

ContactResolver<d>*, ContactResolver<d>*)
```

1. timestep
   The time between frames
   Default - `Globals::DEFAULT_TIMESTEP`

2. broadphase
   The name of the custom broadphase class to use
   Default - 0 (Causes default algorithm to be used)

3. contactGenerator
   The name of the custom contactGenerator class to use
   Default - 0 (Causes default algorithm to be used)

4. contactList
   The name of the custom contactList class to use
   Default - 0 (Causes default data structure to be used)

5. velocityResolver
   The name of the custom velocityResolver class to use
   Default - 0 (Causes default algorithm to be used)

6. positionResolver
   The name of the custom positionResolver class to use
   Default - 0 (Causes default algorithm to be used)

World methods and member data

- step() - Simulates one frame
- findBeneath(Vector) Returns a pointer to the first body that contains that position vector
- findBeneath(AARect) Returns a pointer to the first body that intersects that Axis-Aligned Rectangle

- remove() Removes a Body or ForceGenerator from the world.
  Note: Shapes only exist in the world inside bodies. A shape added to a world directly becomes wrapped in a static body object automatically.

- setTimeStep() The time step is the simulation time between updates. A smaller time step means objects move shorter distances between collision detection/resolution, so inter-penetration at time of collision resolution is less, and the collision is resolved more accurately. A larger time step improves performance (fewer physics calls in a given time) but increases the chances of fast moving objects passing through other object; they pass past them in the time between physics calls.

A.3.3 Shapes

A shape is the generic term for either an area or a volume, depending on whether the context is 2D or 3D.
The basic 2D shapes (areas) supported by the library are:

- CIRCLE - A circle
  Specified by a blank constructor

  ```
  Circle myCircle* = new Circle();
  ```

  Specified by a position & a radius

  ```
  float radius = 1f;
  Circle myCircle* = new Circle(Vector2D(3, 6), radius));
  ```

  Specified by a radius

  ```
  float radius = 1f;
  Circle myCircle* = new Circle(2);
  ```

- LINE - A line
  Specified by a 2 position vectors

  ```
  Line myLine = new Line(Vector2D(1,2),Vector2D(3,4));
  ```

  Specified by a 2 x,y positions

  ```
  Line myLine = new Line(1,2,3,4); //x1 y1 x2 y2
  ```

- CAPSULE - A lozenge; a rectangle with two opposite ends replaced by semi-circles
  Specified by 2 position vectors and a radius

  ```
  Capsule myCapsule = new Capsule2D(Vector2D(8, 5.5f), Vector2D(8, 8), 1)));
  ```

- RECT - A square or rectangle
  Specified by 2 x,y positions (any two opposite corners)

  ```
  Rect myRect = new Rect(1,2,3,4); //x1 y1 x2 y2
  ```

  Specified by a two position vectors (any two opposite corners)

  ```
  Rect myRect = new Rect(Vector3D(1, 2), Vector3D(3, 4)); //or
  Rect myRect = new Rect(Vector3D(1, 4), Vector3D(3, 2)); //or
  Rect myRect = new Rect(Vector3D(3, 4), Vector3D(1, 2)); //or
  Rect myRect = new Rect(Vector3D(3, 2), Vector3D(1, 4));
  ```

  Specified by a position vector (centre of mass), width, height.

  ```
  Rect myRect = new Rect(Vector3D(1.5, 2.5), 1, 1)));
  ```

  Specified by a position vector (centre of mass), width, height, and an orientation

  ```
  Rect myRect = new Rect(Vector3D(4, 7), 1, 1, 0.75f)));
  ```

- POLYGON - A polygon with 3 or more edges/vertices
  Polygons creation is slightly different as a concave check is performed before creation, so they must be called with `Polygon::create(v,n)` (where v is a vector of vertices, and n the number of vertices to create the polygon with)
  If the Polygon is invalid, `Polygon::create(v,n)` returns null. This happens under these situations:

  - Fewer than 3 points were specified (n $\leq$ 2)
  - All points are incident on a single line

  Polygons are made convex by taking the convex hull of the polygon. Metaphorically this is essentially wrapping an elastic band round the (potentially concave) polygon and taking the result (see figure A.1).

  Specified by a vector of co-ordinates and an int representing the number of vertices

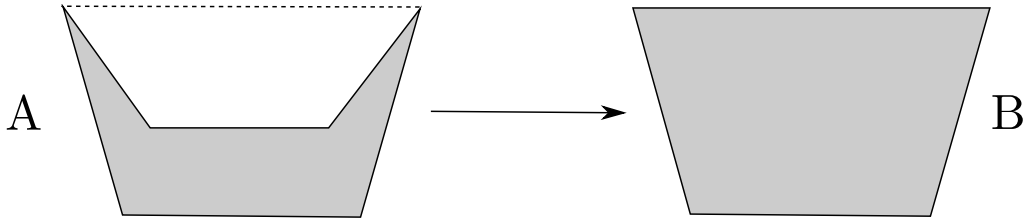


Figure A.1: Finding the convex hull of a polygon

```
//Triangle
Vector2D triangle_points[3] = {Vector2D(0,0),Vector2D(1,1),Vector2D(2,2)}
Polygon myTriangle = Body2D::create(Polygon::create(triangle_points,3);

//Pentagon
Vector2D pentagon[5];
for (int i = 0; i < 5; i++) {
pentagon[i] = Vector2D(-10, 5);
pentagon[i].x += 1.5f * cos(2 * 3.1415f * i / 5);
pentagon[i].y += 1.5f * sin(2 * 3.1415f * i / 5);
}
Polygon myPentagon = Body2D::create(Polygon::create(pentagon, 5)));
```

The basic 3D shapes (volumes) supported[1] by the library are;

- SPHERE - A sphere
  Specified by a blank constructor

  ```
  Sphere mySphere* = new Sphere();
  ```

  Specified by a position & a radius

  ```
  float radius = 1f;
  Sphere mySphere* = new Sphere(Vector3D(3, 6, 1), radius));
  ```

  Specified by a radius

  ```
  float radius = 1f;
  Sphere mySphere = new Sphere(radius)));
  ```

- BOX - A cube or cuboid
  Specified by two position vectors (any of the 4 opposite corner pairs)

[1]More shapes could be used by implementing the CollisionGenerator interface and extending support for new shapes like cylinders or capsules

```
Box myBox = new Box(Vector3D(1, 2, 5), Vector3D(3, 4, 6)); //or
```

Specified by a position vector (centre of mass), width, height, depth, and an orientation

```
Box myBox = new Box(Vector3D(4, 7), 1, 1, 1, 0.75f)));
```

Compound shapes can be created in the form of *Bodies*, discussed below.

A.3.4 Bodies

A body is composed of one[2] or more Shapes. They may optionally contain one or more collision listeners.

Simple bodies

Using the code from the Shapes section above:

```
body2D = new Body2D;
Body2D->add(myCircle);
```

for 2D, or for 3D:

```
body3D = new Body3D;
Body3D->add(mySphere);
```

The same thing can be expressed more succinctly:

```
myWorld3D->add(new Body3D(new Box(Vector3D(0, 0, 0), 4, 8, 1.5)));
```

for 2D, or for 3D:

```
myWorld2D->add(new Body2D(new Rect(Vector2D(0,0), Vector2D(1,1))));
```

[2]Bodies comprised of zero shapes will not throw errors and can be added to the world. By default they will be static and massless, and so will not take part in the physics pipeline. You could specify them to be non-static and have a mass, but even though they'd be affected by gravity they would not collide with anything.

More complex bodies

You can also create multiple-shape (compound) bodies or bodies with various different attributes

- Cross shape with a circle on each end

```
body = new Body2D();
body->add(new Circle(Vector2D(10, 6), 0.4f));
body->add(new Circle(Vector2D(12, 6), 0.4f));
body->add(new Circle(Vector2D(10, 8), 0.4f));
body->add(new Circle(Vector2D(12, 8), 0.4f));
body->add(new Line(Vector2D(10, 6), Vector2D(12, 8)));
body->add(new Line(Vector2D(10, 8), Vector2D(12, 6)));
world->add(body);
```

- See-saw

```
body = new Body2D();
body->add(new Line(10, 5, 20, 1));
body->add(new Line(15, 3, 14, 1));
Circle* c = new Circle(Vector2D(14, 1), 0.2f);
c->setDensity(1);
body->add(c);
world->add(body);
```

- Dense circles

```
body = new Body2D();
c = new Circle(Vector2D(15, 13), 1.5f);
c->setDensity(1.5);
body->add(c);
c = new Circle(Vector2D(14.5, 11), 1.5f);
c->setDensity(1);
body->add(c);
world->add(body);
```

Body methods and member data

- add
  - Shape - Bodies have one or more shapes added to them. The shapes need not be connected, and can be intersecting.

  - Collision listener - See A.3.4.

- contains - Returns true if the specified position vector is within the area of the Body
- intersects - Returns true if the specified Axis-Aligned Bounding-Box (AABB) overlaps the body.
- create
  Alternative syntax to `new Body();`.
- createStatic - As above, but the resulting body is static
- isPinned() - Pinned bodies have a fixed position, but may rotate
- isRotatable() - Non-rotatable bodies have a fixed rotation vector of 0. (i.e. they cannot spin).
- isStatic() - Static bodies with have a fixed position vector. Their mass is treated as infinite for the purposes of collisions. They are pinned and not rotatable.)

Collision Listeners

Collision listeners allow the application using the sandbox to receive notifications when a body collides. This is necessary to perform application logic around the collisions. For example, a bullet hitting a player. To receive such notifications the application must implement the virtual method `notify`;

```
// Return true to process contact
// Return false to ignore contact
// index is 0 or 1 depending on which body registered the listener
notify (Contact* contact, int index)
```

Every frame that the registered body is in contact with something else the library will call the application notification method listed in the appropriate listener.

A.3.5 ForceGenerators

- Joints Specified by Pendulum

```
Joint<TwoDee>* myJoint = new Joint<TwoDee>();
Body2D* fixed = new Body2D(new Circle( Vector2D(0,11), 2) );
Body2D* pendulum = new Body2D(new Rect(Vector2D(9,11), 1, 1) );
fixed->setStatic(true);
myJoint->set(fixed, pendulum, Vector2D(0,11));
world->add(fixed);
world->add(pendulum);
world->add(myJoint);
```

- Springs

```
Body2D* springbox = new Body2D(new Rect(Vector2D(20,45), 2, 2) );
AnchoredSpring2D* spring = new AnchoredSpring2D(Vector2D(20,50),
springbox, Vector2D(), 6, 3);
world->add(springbox);
world->add(spring);
```

A.3.6 Settings

- Sleeping (on/off, linear/angular velocity thresholds, and more)
  useSleeping = true;
  linearVelocitySleepEpsilon = 1.5f;
  angularVelocitySleepEpsilon = 0.01f;
  sleepRWABias = 0.5f;

- Air resistance parameters
  airResistance = false;
  linearDrag = true;
  linearDragCoefficient = 0.01f;
  quadraticDragCoefficient = 0.01f;

- Default density/coefficient of friction/coefficient of restitution for newly created shapes
  DEFAULT_DENSITY = 1.0f;
  DEFAULT_RESTITUTION = 0.4f;
  DEFAULT_FRICTION = 0.5f;

- Epsilon
  penetrationEpsilon = 0.005f;

- Default value of gravity (in y axis)
  GRAVITY = -9.8f;

- Default length of a simulation step
  `DEFAULT_TIMESTEP = 1.0f / 60.0f;`

- Contact options
  cacheContacts = true;
  findMultipleContacts = true;
  findSleepingContacts = false;

- Restitution (On/off, threshold)
  useRestitution = true;
  restitutionThreshold = 0.2f;

A.4 Gotchas, tips

- In Visual Studio, the `_SCL_SECURE` and `_HAS_ITERATOR_DEBUGGING` macros must be set to 0 in all code being linked together. This includes external libraries. If you don't do this it will compile and link without errors, but crash at runtime.

- The `Orientation::transform` method takes local coordinates and outputs world coordinates, not the other way round. Probably you should always use the `worldToLocalDir`/`localToWorldDir` methods of the `Body` and `Shape` classes, which are far more intuitively named.

- Setting the default value for gravity (found in `Globals::DEFAULT_GRAVITY`) will apply only to new `World` instances created: existing `World`s will not be modified. An alternative method of setting gravity is to call `world->setGravity` on an existing `World`.

- Similarly, the default values for density (`Globals::DEFAULT_DENSITY`), restitution (`Globals::DEFAULT_RESTITUTION`) and friction (`Globals::DEFAULT_FRICTION`)are only applied to new shapes.

- There is no way to adjust gravity on a per-body basis. You must instead set it to 0 and then manually apply gravity as an impulse to those bodies which require it.

- There is no way of explicitly setting the coefficient of restitution or coefficient of friction for any given contact. Both bodies have their own value, and the actual value used is an average of the two. In addition, two different types of average are used: the coefficient of restitution is taken to be the arithmetic mean of the two values, whereas the coefficient of friction uses the geometric mean.

- Circles and spheres which are exactly aligned (at creation) will stack rather than roll off one another. Technically this is mathematically correct: there are no horizontal forces acting. However, since in real life this cannot happen, preventing this behaviour may be desired. In this case, the shapes should be created with a small random offset added to the position. (N.B. care should be taken with randomness if a deterministic simulation is desired.)

Appendix B

Sandbox user guide

B.1 Compiling & Installation

First, follow the installation instructions for the library (see Appendix A).

B.1.1 Windows

Visual Studio

- Download SDL
  - Go to the LibSDL webpage: `http://www.libsdl.org/download-1.2.php`
  - Download the latest development library for Visual C++ (named something like SDL-devel-1.2.13-VC8.zip).
  - Extract the zipfile to somewhere sensible (for example C:\Program Files).
- Setup FreeType2 (2.3.7)
  - `http://sourceforge.net/project/showfiles.php?group_id=3157`
  - Extract somewhere (C:\Program Files\freetype-2.3.7\)
  - Download the headers and precompiled win32 debug .lib from `http://projects.draknek.org/versions/download/3?attachment_id=5`
  - Unzip this either in your global lib/header folders (and then add those directories to "VC++ Directories") OR if you're lazy it should be ok to dump the .lib in the /lib folder of the project and the header folder (complete) in /include.

- Setup FTGL
  - `http://ftgl.wiki.sourceforge.net/`
  - Set it up in Visual c++ like the others
  - Download the headers and precompiled win32 debug .lib from `http://projects.draknek.org/versions/download/3?attachment_id=5`
  - Unzip this either in your global lib/header folders (and then add those directories to "VC++ Directories") OR if your lazy it should be ok to dump the .lib in the /lib folder of the project and the header folder (complete) in /include.
- Configure Visual C++
  - Start Visual C++ and go to Tools → Options → Projects and Solutions → VC++ Directories.
  - From the drop-down box in the top-right corner, choose 'Include files'. Add:
    - C:\Program Files\SDL-1.2.13\include
    - C:\Program Files\freetype\include
    - C:\Program Files\FTGL\include
    - (depending on where you installed them)
  - Now select 'Library files' from the drop-down box. Add:
    - C:\Program Files\SDL-1.2.13 \lib
    - C:\Program Files\Design\freetype-2.3.7\lib
    - (again, depending on where you installed them).
  - Now select 'Source files'. Add:
    - C:\Program Files\Design\freetype-2.3.7\src
  - Click OK and save these changes.
- Update your PATH variable. Add:
  - C:\Program Files\SDL-1.2.13\lib;
- Update your debug path.
  - For BOTH physics and sandbox projects:
    - Right click on project, → Properties
    - Configuration Properties → Debugging,
    - Set working directory to `../..`
- Open the project

  - Navigate to the physics\build\VS2008 directory.
  - Open the Physics Project.sln or Sandbox.csproj file in Visual C++.
  - Compile the sandbox.

B.1.2 Linux

- Download Boost
  - In Ubuntu, you can do this with `sudo apt-get install build-essential libboost-dev`.
  - If you do not have installation privileges, download the latest release from the Boost webpage: `http://www.boost.org/users/download/`. Then extract the boost subdirectory from the archive into physics/include/
- Download libraries
  - In Ubuntu, you can do this with `sudo apt-get install libsdl1.2-dev libgl1-mesa-dev libglu1-mesa-dev libfreetype6-dev libftgl-dev`.
  - If you do not have installation privileges, you will need to find and compile any of the following which are not already installed:
    * SDL
    * OpenGL
    * GLU
    * FreeType
    * FTGL

    Install the include directories to physics/include/ and the generated libraries (*.a files) to physics/lib/.
- Open a terminal and navigate to the physics directory.
- Run `make sandbox`

B.2 Using the sandbox

The sandbox is an application made using the Large Polygon Collider library to demonstrate some of its features with a simple user interface. The binary can be found in the `/bin/` folder after compilation.

B.2.1 Preset worlds

The sandbox has a number of preset 'worlds' that provide demonstrations of various features in the engine. You can switch between them using the numbers 0-9 on the keyboard (See B.2.3 for a full list of controls). The worlds are:

1. (default) A variety of supported object types, scattered near the centre of the world, including: a circle, a rectangle, a capsule, conjoined circles, arbitrary polygons (a regular pentagon and a triangle), and a number of shapes joined by lines to form more complex shapes.

2. Two large towers, one of circles, one of squares.

3. A house of cards: a fragile triangle structure built with many thin rectangles.

4. A single box, which starts slightly elevated off the ground. This was used to test the correctness of box-box collisions.

5. A triangular stack of many boxes that starts slightly elevated off the ground.

6. A larger triangular stack of many boxes, this time created with a small distance between each layer, resulting in a more stable construction.

7. Pinball: Many small circles, poised to fall through an array of static pentagons and free-rotating spinners into hoppers below.

8. Joints and constraints demo: a pivoted rectangle (used as a seesaw); a pendulum, a double pendulum and a 'Newton's Cradle'; a rudimentary wheeled vehicle on a rope bridge, and a free-swinging chain.

9. Dominoes: A line of long rectangles balanced close together on their thin end, with a circle poised to knock over the leftmost one.

10. Friction demo: four small rectangles on a large slope, each with differing friction values.

11. A more complex dominoes scenario.

There is also a world 0, which is empty except for the four bounding walls, and is useful when running automated routines. World 10 is called when the 0 key is pressed and worlds 0 & 11 can only be accessed through the command-line (see B.2.2).

Parameter-free options	
`2D,2d`	Loads 2D sandbox (assumed by default)
`3D,3d`	Loads 3D sandbox
`-i`	Disables interactive mode
`-p`	Profiler: Unlimited logging, enable file writing/stdout output
`-g`	Profiler: Unlimited logging, enable file writing/stdout output, enable graph plotting
`--no-sleeping`	Disables sleeping
`--help`	Displays usage notes

Options that require parameters		
`-S`	Director script	Must be a number 0-2
`-s`	Scene	aka preset. Must be a number 0-9
`-o`	output file name	filename for profiler output

Table B.1: Parameters

B.2.2 Command-line parameters

Command-line parameters

Sandbox

```
Usage: sandbox [2d|3d] [-p | -g] [-i] [-o outputfile] [-s scene] [-S script]
```

1 is used if parameter is invalid. If conflicting options are specified the ones towards the end of the parameter string are used.

B.2.3 Sandbox Controls quick reference

World manipulation

Drawing and Spawning

Mouse camera control

The camera can be panned at any time[1] by holding the right-mouse-button and dragging.

[1] world paused/unpaused, whilst drawing shapes, whilst zooming, whilst carrying a body

World manipulation	
Key	Effect
`0-9`	Load preset 0-9
`F1`	Save world
`F2`	Load world
`r`	Reload last preset
`p`	(Un)pause world
`x`	Clear world of bodies
`d`	Delete selected body/bodies (or under cursor if nothing selected)
`w`	Wake all bodies up
`PageUp`	Zoom in
`PageDown`	Zoom down

Drawing and Spawning	
Key	Effect
`c`	circles
`b`	boxes
`v`	static circles
`n`	spawn random polygon (degree 3-8)
`o`	fireworks at cursor
`f5`	toggle meteor shower
`f6`	toggle rain
`f7`	toggle fireworks

Miscellaneous	
`s`	static
`m`	merge
`u`	unmerge
`i`	cycle verbosity
`l`	stats logging

Mouse body control

Bodies can be picked up by the mouse with the left-mouse-button.
When the world is paused, carried bodies are attached to the cursor at the point where they were clicked on, and can be positioned anywhere (including intersecting other bodies) with a high degree of precision. When the world is unpaused their interpenetrations will be resolved as normal. In normal operation, the body is attached to the mouse by a spring, allowing you to apply a variable force to it by increasing the speed at which you move the mouse away from it. This approach avoids applying infinite forces to the carried body. Note that whilst dragging you can press `CTRL` to kill rotation, and `CTRL+Mousewheel` to rotate.

Appendix C

Software License

Large Polygon Collider – a real-time physics engine (and its demo sandbox) version 1.0, April 2009

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Appendix D

Minutes

This appendix contains all minutes taken at group meetings. They are brief notes of the agenda and outcome of the meetings, intended to act as an *aide memoire* for group members, rather than a detailed account of proceedings.

September 28th

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley.

Organisational ToDo

- Officially submit project
- UML class diagrams
- Critical path analysis of what components depend on each other
- Planned development timeline
- Leigh: Research potential problems with using templates
- Alan: Host a workshop introducing C++, templates, and the existing code

Development ToDo

- Begin design of system that can handle 3D.
- Greater levels of code modularity
- Text-drawing in the visualiser: research GLUT and alternatives

Long-term development

- Shock propagation- not implemented in any existing engine (other than Box2D-Lite, which is just experimental)
- Fluid simulation- Leigh said he found a simple way of implementing this in 2D; may be extendable to 3D

Specification/other documentation

- Due in by the end of week 4
- Some of the content may be covered in System Lifecycle Management
- We need to document as we go: ideally full writeups that can be placed directly into the final report
- Paperwork and stuff is likely to take significantly more time than development

Doxygen

George has researched Doxygen; it should definitely be useful. He added a config file to the repository: in Linux you can generate the documentation by running `make doc` or `make doxygen`. How to build it on Windows currently unknown. The documentation is created in the directory doc/html.

Monday October 10th

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Splitting team into 2 development groups

- Result: 2 lead programmers, 3 programmers (2 assigned to one lead, 1 to the other)

Final features we want in the project

- Discussed; nothing new concluded.

Objectives to be reached before specification

- Progress text rendering. Note infeasibility of using glut, due to the effect this would have on the 3D component.

Relative merits of bodies defined in world vs local space

- Discussion of Alan's rework of the `Shape` class
- Adding Shapes to the world now wraps them in a body.
- More sensible static shape support. Also replaces `colour()`.

Restructuring of CollisionManager

- Mostly to facilitate work on broadphase collisions.
- Concluded that we would not support on-the-fly changing of collision code; a restart would be required.

Other issues

- Discussed addition of a "Settings object" to be passed into the world with such information as which algorithms the world should be using
- Alan gave a talk on the code structure of his third year project, and overview of hierarchies.

Tasks assigned

- 2D broadphase collision - George
- Profiler (run times of components, with possiblity for memory usage analysis) - Ben
- Text rendering/Sandbox - Rich
- Leigh/Alan continuing with early core physics, delegation to come

Roles assigned

- Leigh and Alan designated team leaders.
- Richard designated documentation co-ordinator. (inc UML class diagrams and Critical path analysis)

Wednesday October 13th

Present: Richard Falconer, Ben Hallett, Alan Hazelden

Gantt chart, spec documentation

- Gantt chart/spec to be discussed further at a later date.
- All outstanding features listed on Torque issue tracker are on the gantt chart.

Other issues

- Profiler progress
- TGL Text rendering progress

Conclusion

Please have a think about what we will be implementing. This will be discussed and added to the spec/timetable at next meeting.

Wednesday October 16th

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Documentation

- Gantt chart
- Specification
- Profiler progress

Profiler

- Where in the implementation should the profiler reside?
- Multiple worlds per profile not supported

Specification

- Explanation of what a gantt chart is
- Problems with scheduling of algorithms and algorithm modifications; many of the issues are mutually exclusive

Organisation

- Richard is going to break spec into components, and designate people to do each part and monitor progress.

Project rename

- agreed on: Large Polygon Collider

Extentions to features and objectives

- Blender saving and loading
- Increased empahsis on drawing conclusions from algorithms
- PAL - Physics abstraction layer
- Scythe Physics Editor
- 2D engine shifted to be more like Phun
- Collada / TinyXML file saving

We envisioned the use of Scythe to work as follows:

- 3D Benchmarking/Profiling
    1. File specifying 3D scene is made in Scythe Physics Editor/Blender, in COLLADA format
    2. This file is loaded into PAL.
    3. Our physics library implements (a subset of) methods required by PAL
    4. PAL makes calls to our physics engine for the scene we specified
    5. Algorithms to be used are selected, either via UI or command line switches
    6. Scythe Physics Editor runs, shows result
    7. Our profiler generates performance data on our individual algorithms
    8. Performance benchmarks are generated by PAL
    9. We analyse PAL and Profiler output
- 2D Benchmarking/Profiling
    1. (Optional) File specifying 2D scene is made in 2D sandbox on a previous run
    2. Algorithms to be used are selected, either via UI or command line switches
    3. 2D sandbox runs, shows result

4. Our profiler generates performance data on our individual algorithms
5. We analyse profiler output

- 2D playing
  1. (Optional) File specifying 2D scene is made in 2D sandbox on a previous run
  2. (Optional) File is loaded by the 2D sandbox, by parsing with TinyXML
  3. Algorithms to be used are selected, either via UI or command line switches
  4. Sandbox runs, user can do Phun-like things with it
- 3D playing (Complex, probably a late-extension)
  1. (Optional) File specifying 2D scene is made in Scythe Physics Editor/Blender, in COLLADA format
  2. (Optional) File is loaded into Blender
  3. Algorithms to be used are selected, either via UI or command line switches
  4. Scythe runs, world bodies added/removed/manipulate

Our library API

- Look at the APIs of other similar projects to get ideas for the API we are going to build

Wednesday October 22nd

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Specification

- Specify core & optional objectives
- Explain risks and potential issues
- Project management:
  - Timeline: we should aim to implement the hard stuff early, while there's still time to experiment
  - Make backup plans
  - We must improve communication! Project management website?

Specification responsibilities

To be done before tomorrow, Thursday 23rd:

- Section 1: Leigh
- Section 2: Alan
- Section 3: Ben
- Section 4: George
- Section 5: Rich

Friday October 31st

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Management

George elected Poster Co-ordinator

Development

Priorities are:

- 2d algorithms
- The profiler
- The 3D camera

Aiming to halt development 20th December to focus on poster. Other development notes:

- Rather than have a 3D sandbox we're going ahead with the "one generic sandbox" approach.
- New 3D camera class to be added to the sandbox to accomplish this.
- Leigh aims to have box-box and box-plane naive collisions working for the poster presentation.
- Profiler needs to to be able to run simulation x n-times, average results, and optionally output results
- Post-poster Richard is going to focus on the sandbox UI.

Poster

- To be made in OpenOffice[1]
- Aiming to finish 1 week before deadline, and print off multiple copies (depending on price); printing closer to the deadline may be more expensive.
- Ideally, OpenOffice poster file to contain only layout until we collate everything at once; allows subversion to work with the plaintext and avoids the conflict issues we had with specification.

Tuesday 13th January 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Organisation

Insufficient development achieved so far:

- We need to use Redmine's ticket system more.
- We must set deadlines on Redmine (and actually stick to them)
- Bi-weekly meetings now development sessions:
  - Monday 10:00
  - Wednesday 12:00

Monday 19th January 2009

Development

Tasks for Monday 26th:

- Rich: work through list of UI features from email
- Ben: investigate GnuPlot for graph output; add commandline options to sandbox (Alan to draw up list); show more statistics at the top of the window
- George: add option in `NaiveBroadPhase` to use bounding circles instead of bounding boxes; start on sort and sweep
- Leigh: commit work done so far; work on a very basic 3D implementation with just spheres; plus more to be split between Leigh and Alan

[1] Retrospective note: this was later changed to Photoshop

- Alan: Create 3D primitives & extremely naive contact detection; make it compile in Windows; plus more to be split between Leigh and Alan

Monday 28th January 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Development

- Discussion of 3D stuff between Leigh and Alan
- Diagnosis of Richard's save/load code by Alan, resulting in its completion.

Monday 3rd February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, George Stanley

Discussed

- Implementation of callbacks: the only thing not yet on the issue tracker. Reasonably simple, but large.
- Core physics work behind schedule: may need to delegate more physics stuff to Rich/Ben/George.
- GUI: aedGUI inappropriate; we don't use SDL for drawing; only use it for mouse/keyboard IO. Now looking at GTK+, Qt, and "open GL library".

Wednesday 4th February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden

Bug fixes

- Whilst testing load/save of body comprised of a low-density shape and a high-density shape, Richard found a bug in the bounding-circle code, which was fixed by Alan.
- Significant time spent discussing peculiarities with camera object on windows. Currently unresolved; camera update methods simplified as temporary debug measure.[2]

[2]Retrospective note: The bugs in the camera position have been fixed by Alan.

Profiler & Director

- Ben discussed with us the required functionality of director w.r.t benchmark scenarios.
- gnuplot graphical output of "explosion" benchmark was met with warm reception; everything appears to be performing as expected.

GUI

Following research from George we're looking at Crazy Eddie's GUI. Semi-setup in fork.

Tutorials

Alan gave Richard a brief but enlightening introduction to OpenGL, following some discussion.

Tasks assigned

- Richard: UI
- Ben: Director, fixing graphs, sleep system
- All: Redmine to be updated

Monday 9rd February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Meeting with Abhir

- Assuming new UI is semi-functional we should arrange a meeting to show progress.[3]
- Should happen sooner rather than later; Week 6 and we've not seen him yet.

Development

- Large discussion between Alan and Leigh on feasibility of various todo physics stuff.
- Rich working on UI & added arbitrary circle creation.

[3]Retrospective note: This meeting was never set up

Monday 16th February 2009

Present: (Not Richard; no minutes taken in his absence)

Monday 9rd February 2009

Present: Richard Falconer, Alan Hazelden

Development

- Rich: UI state development
- Alan: Miscellaneous development

Monday 23rd February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, George Stanley

Tasks assigned

George

Sort and sweep algorithm research and development

Ben

- Multiple graph generation
- Air resistance tweaking

Alan

- Stability
- Branch wrapper for Box2D

Richard

- Selection rectangle code. (Logic, drawing, and intersection tests)

Wednesday 25th February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden

Tasks assigned

Richard

- Intersection test for AARect to circles works
- Re-worked the safety states for drawing polygons/circles/boxes. Fixes bug #65.
- Hopefully by end of day will have separating axis theorem check done for AARect to rect intersections.

Monday 2nd Feb & March 4th Feb

Development sessions; no minutes taken.

March 4th February 2009

Present: Richard Falconer, Ben Hallett, Alan Hazelden, Leigh Robinson, George Stanley

Final report

- Collation of report documents completed over Easter
- Tweaks and alterations
- Implementation section expanded