

Sam Stokes

ss496

**Collision detection in the
simulation of rigid body
motion**

Computer Science Tripos Part II

Robinson College

September 29, 2005

Proforma

Name: **Sam Stokes**
College: **Robinson College**
Project Title: **Collision detection in the simulation of rigid
body motion**
Examination: **Computer Science Tripos Part II, May 2005**
Word Count: **11955**
Project Originator: **Sam Stokes**
Project Supervisor: **Richard Southern**

Original aims of the project

To create a physical simulation of rigid body motion, and to investigate methods of collision detection. The program should read in a scene description in a textual format, and animate its behaviour in a realistic and physical way, simulating the physical laws governing the motion of rigid bodies. Finally, the program should be efficient enough to simulate moderately complex scenes, containing at least several hundred objects, in real time.

Work completed

A rigid body motion simulation has been implemented. The program can read in a scene description, render the scene in 3D and simulate its behaviour. Object types supported are planes and uniform spheres and cylinders. The project simulates gravity and collisions between objects, and uses a sophisticated collision detection algorithm which allows it to handle scenes containing over a thousand objects in real time. In addition, this document contains theoretical and empirical comparisons of two collision detection algorithms used in the project.

Special difficulties

None.

Declaration of Originality

I Sam Stokes of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1	Introduction	1
2	Preparation	3
2.1	Requirements analysis	3
2.1.1	Change of project title	4
2.2	Resources used	4
2.3	Theory and algorithms	5
2.3.1	Quaternion arithmetic	5
2.3.2	Collision detection	6
2.3.3	Baraff's algorithm	8
2.4	On clean-room practices	9
3	Implementation	11
3.1	General features	11
3.2	Data structures	12
3.2.1	Object types	12
3.2.2	Mathematical classes	13
3.3	Frontend	14
3.3.1	Program structure	15
3.4	SDL parser	15
3.4.1	Generating scene files	16
3.5	Renderer	17
3.6	Simulation	18
3.6.1	Scene animation	18
3.6.2	Collision response	19
3.6.3	Intersection tests	20
3.6.4	Naïve collision detection	21
3.7	Baraff's algorithm	22
3.7.1	Axis-aligned bounding boxes	22
3.7.2	Algorithm details	23

4	Evaluation	25
4.1	What has been achieved	26
4.2	Example scenes	27
4.2.1	Pool break	27
4.2.2	Brownian motion	28
4.2.3	Spheres and pyramid	28
4.2.4	Spinning cylinder	28
4.2.5	Table	31
4.3	Performance evaluation	31
4.3.1	Real-time performance	31
4.3.2	Comparison of CD algorithms	34
5	Conclusions	39
	Bibliography	40
A	Scene description language	41
B	Sample code	43
B.1	Baraff's algorithm	43
B.2	Quaternion class	44
B.3	Object creation	45
C	Project Proposal	47

List of Figures

3.1	Class hierarchy for object types	12
4.1	“Pool break” scene (<i>a</i>)	29
4.2	“Pool break” scene (<i>b</i>)	29
4.3	“Brownian motion” scene (<i>a</i>)	29
4.4	“Brownian motion” scene (<i>b</i>)	29
4.5	“Pyramid” scene (<i>a</i>)	30
4.6	“Pyramid” scene (<i>b</i>)	30
4.7	“Spinning cylinder” scene (<i>a</i>)	30
4.8	“Spinning cylinder” scene (<i>b</i>)	30
4.9	‘Table’ scene	32
4.10	Frame-rate scaling: pool breaks	32
4.11	Frame-rate scaling: Brownian motion	33
4.12	Frame-rate scaling: spheres and pyramid	33
4.13	“Pool breaks” scene	35
4.14	Frame-rate scaling with naïve CD algorithm	35
4.15	Simulation time – algorithm comparison	37
4.16	Simulation time trace for naïve algorithm	37
4.17	Simulation time trace for Baraff’s algorithm	38
4.18	Simulation time trace for Baraff’s algorithm	38

Chapter 1

Introduction

The modelling and simulation of physical systems is an important and widely used application of computer technology. Physics simulations are used in scientific research to study systems and test theories; engineers and designers find them useful for building ‘virtual prototypes’, which allow early designs to be evaluated more economically than building a real prototype; and they are incorporated into many game engines and graphics demos to give the impression of a ‘realistic’ virtual environment.

While designing and implementing a physical simulation involves a great deal of physics and applied mathematics, there are also significant computer science issues to be addressed. One of the major tasks for almost any physical simulation is detecting when objects in a scene collide (particularly to enforce constraints such as nonpenetration of rigid bodies as the system evolves). This involves finding all pairs of objects that intersect and performing some action on them, typically forcing the objects apart or deforming them. Unfortunately, the obvious method for performing collision detection – naïvely checking every pair of objects in the scene – is in general too slow to be useful, so one of the most important components of an efficient physical simulation is an efficient collision detection system. There has been much active research in this area, and solutions often involve powerful techniques from computational geometry. Some recent papers surveying developments in the field are [JTT00, LG98].

Many physical systems are most conveniently modelled by *rigid body motion*. A rigid body is simply one whose shape does not change in the course of its interactions: thus the only transformations it may undergo are translations and rotations. Rigidity is a modelling assumption, and as such, it is an ideal: real objects are not perfectly rigid. However, many behave largely as if they were: for example, billiard balls are well modelled as rigid, and the kinetic theory of gases models an ideal gas as composed of many rigid spherical particles colliding elastically. The simulation of rigid body motion is an important example of a physical simulation.

The project described in this report is a rigid body motion simulator. There are clearly a large number of physical phenomena that may be modelled by such a simulator: inelastic collision, friction, elastic forces and gravity are examples. One also might desire an arbitrary degree of physical accuracy. However, achieving these largely involves the application of mathematics or physics rather than computer science (and moreover requires some fairly sophisticated modelling techniques), and I consider that it is most educational to focus mainly on the collision detection system. I therefore consider only a restricted set of physical phenomena: elastic or inelastic collision of rigid bodies, and motion under uniform gravity. I have also chosen to restrict the class of objects modelled to contain only spheres, finite cylinders and infinite planes: these are relatively simple geometrically, but serve as demonstrations of the principles involved in a collision detection system. The program has a modular design which makes it relatively straightforward to add additional primitive types.

Many applications of rigid body motion simulation, such as game engines and computer animation, require or benefit from real-time or interactive simulation rates. My project, in particular its collision detection system, is therefore designed to be efficient enough to produce animation in real-time. In addition, it is designed to scale well with scene complexity, so that it can fulfil the real-time requirement for scenes containing several hundred objects on desktop-class hardware.

Chapter 2

Preparation

2.1 Requirements analysis

The aims of the project are to produce a real-time simulation of rigid body motion, and to investigate methods of collision detection and their impacts on performance. Physical accuracy is desirable but not a principal goal; however, the accuracy of the simulation should be sufficient to appear realistic to a human observer. Therefore, the key requirements for the project are:

1. It must be able to read in a textual description of a scene.
2. It must display and animate the scene and simulate the behaviour of the objects therein.
3. It must perform this simulation in real-time on desktop-class hardware. (The figure often used as a rule of thumb for how often an animation must be updated in order for the human eye to perceive it as smooth is 30 frames per second: I therefore take this figure as the minimum frame-rate which will be considered to satisfy this real-time requirement.)
4. It must scale well with the number of objects in the scene. It should be possible to fulfil the real-time requirement for scenes containing several hundred objects.
5. If possible, two methods of collision detection should be implemented and compared regarding their suitability for the project.
6. The simulated objects should behave in a way that appears realistic and physical to a human observer.
7. The operation of the project should be demonstrable on several simple test scenes, and some more complicated scenes, in which various rigid bodies collide.

The development of the project will follow an iterative model similar to the Spiral development model.

2.1.1 Change of project title

It should be noted that the title of the project and of this dissertation – “Collision detection in the simulation of rigid body motion” – is somewhat different from that given on the Project Proposal (“An approximate graphical simulation of rigid body motion”). This represents a change of project emphasis, though not of subject matter.

It was decided during the preparation and analysis phase that it would be pedagogically more worthwhile and academically more interesting to concentrate on the issues relating to collision detection (rather than, for example, the sophisticated simulation of physical phenomena or the pursuit of greater physical accuracy, which involve more mathematics and physics than computer science). The change of title reflects this decision, and the new title more accurately conveys the project’s intent.

2.2 Resources used

The project was developed and tested on my desktop PC (AMD Athlon 64 2GHz, 1GB RAM, Nvidia Geforce 6800GT graphics card, Microsoft Windows XP).

The project source code and accompanying files, as well as the \LaTeX source of this document, were kept under the Subversion version control system during development, keeping the repository on a server provided by my College. This provided numerous advantages, including a form of backup (since the repository server was located away from my main work machine), easy rectification of errors, the ability to easily track the progress of the project, and to maintain multiple versions of the code.

The need to keep a backup in case of system failure was taken seriously and addressed from the beginning of the project. I set up a script on the server hosting my Subversion repository to automatically backup the repository on a daily basis to Pelican, the University backup server, and to my home server. In addition, the project working directory was backed up regularly to the same locations, and to a portable USB Flash drive.

Because of its reputation for high performance and its support for object-oriented programming, I chose C++ as the implementation language for the project. The C++ Standard Template Library (STL) includes efficient implementations of several key data structures and algorithms, of which use was made where appropriate. The OpenGL graphics library was selected for the 3D renderer due to its straightforward API and tried-and-tested C++ bindings. The Python scripting language is also used to perform some aux-

iliary functions, such as generation of sample input and automated benchmarking.

The project was developed on the Windows platform as this was the most readily available to me at time of development. Although I originally planned to use the Microsoft Visual Studio development environment, it was decided that greater familiarity with UNIX tools such as `vim` and `make` made the Cygwin Linux emulation environment a more suitable choice. The compiler used was the GNU C++ compiler `g++`.

When the project began I was familiar, but not proficient, with C++ and unfamiliar with OpenGL and the Win32 API, so part of the preparation stage was to better acquaint myself with these systems. This was done with the aid of books (particularly [Str00, AH04]) and by writing small example programs.

2.3 Theory and algorithms

2.3.1 Quaternion arithmetic

A rigid body simulation must keep track of the state of the scene it is modelling, and in particular the configurations of the objects in the scene. For a rigid body the configuration consists of the linear position of the object's centre and a representation of the object's orientation. The position of the object can trivially be stored as a 3D vector, but it is less obvious how to represent its orientation. Several approaches are common, including storing a 3D vector of Euler angles and using a 3×3 rotation matrix, but these have disadvantages.

A major disadvantage of the commonly used Euler angle representation is that for certain orientations, the Euler angles are singular. This can be observed in some 3D applications which use this representation: if one points the 'camera' straight up or down along the world vertical axis, a small movement of the controls – i.e. a small change in the Euler angles – results in a very rapid 'spin' around the vertical. This also leads to the phenomenon known as *gimbal lock* whereby one axis of rotation becomes aligned with another, causing a degree of freedom to be lost. A widely used and efficient technique proposed by Ken Shoemake in [Sho85], which avoids both of these problems, is to store the orientation as a *quaternion*.

A full discussion of quaternion arithmetic is beyond the scope of this report, but in essence a quaternion is an extension of a complex number. A general quaternion is given by

$$q = w + xi + yj + zk \quad w, x, y, z \in \mathbb{R}$$

with $i^2 = j^2 = k^2 = -1$, $ij = -ji = k$, $jk = -kj = i$ and $ki = -ik = j$, or equivalently

$$q = w + \mathbf{v} \quad w \in \mathbb{R}$$

with \mathbf{v} a vector. The *conjugate* of a quaternion is given by

$$q^* = w - \mathbf{v}.$$

The rotation of a vector can then be represented by quaternion multiplication. For a vector \mathbf{x} (represented as a quaternion with $w = 0$) and a quaternion $q = \cos(\theta/2) + \mathbf{v} \sin(\theta/2)$, the expression

$$\mathbf{x}' = q\mathbf{x}q^* \tag{2.1}$$

represents the rotation of \mathbf{x} by angle θ around axis \mathbf{v} (with $\|\mathbf{v}\| = 1$).

Quaternions require less storage than rotation matrices, and allow rotations to be computed more cheaply. They also do not suffer from the singularities and ‘gimbal lock’ problem that Euler angles are prone to. Their major advantage in animation, however, is that computing their evolution over time is very simple. If $\hat{\omega}$ is the quaternion representation of the angular velocity vector ω , then [vdB04] gives the time derivative of a quaternion q as

$$\dot{q} = \frac{1}{2}\hat{\omega}q. \tag{2.2}$$

2.3.2 Collision detection

The core of the project, and the main factor determining its efficiency, is the collision detection (CD) subsystem. The collision detection problem can be stated as follows: given the geometric configurations of all objects in a scene, find the set of all pairs of objects in the scene that are intersecting. The problem is usually extended to require returning some geometric information about the colliding pairs (such as the greatest penetration depth).

It can be seen that if the scene contains n objects, then the CD problem must have $O(n^2)$ complexity in the worst case. The number of object pairs in the scene is $n(n-1)/2$, and in the case that all objects are simultaneously colliding, all pairs will have to be detected and added to the result set. Clearly this worst case is also a highly uncommon case for most scenarios, and it is possible to do considerably better in the average case. Moreover, in applications where CD is used, most algorithms operating on the scene work in better than quadratic time, so an $O(n^2)$ CD algorithm becomes a bottleneck for large scenes.

In order to improve on the quadratic bound, it is necessary to take advantage of some assumed properties about the system being modelled. Some useful properties hold in most common cases:

Spatial coherence is the property that objects occupy only a relatively small proportion of the space, and that collisions are relatively infrequent. Collisions are usually resolved rather than maintained. This means that at any given time, the number of colliding object pairs is likely to be much smaller than the total number of pairs.

Temporal coherence means that the scene changes relatively little over small time intervals. This is similar to a ‘continuity’ property, but also implies that objects should not have very large linear or angular velocities. Since physical simulations usually approximate continuous time by a series of discrete ‘frames’, temporal coherence can usefully be expressed as *frame coherence*: this means that algorithms can typically reuse calculations from previous frames in order to avoid unnecessary recomputation.

A naïve solution to the collision detection problem is simply to iterate through all pairs of objects, testing intersection and adding them to the colliding set if the test returns true. This algorithm has the advantage of simplicity, and so is suitable (and indeed was used in this project) as a prototype algorithm while testing other aspects of a program. However, it clearly does not scale well with the number of objects: since it always considers every object pair, the time complexity of the algorithm is $\Theta(n^2)$ in all cases.

The naïve algorithm suffers from a second problem: it performs poorly when the geometry of the objects themselves is complex. While detecting intersection between spheres requires only a few arithmetic operations, for more complex shapes intersection testing becomes a significant expense. For convex polyhedra, for example, the time complexity of intersection testing depends on the number of vertices. Algorithms are known that perform this test in $O(\log^2 v)$ time, where v is the number of vertices, by preprocessing the polyhedron model into a hierarchical boundary representation: see [JTT00] §3.1 for more details.

A common technique for improving performance for collision detection between complex shapes is to use *bounding volumes*: this exploits the spatial coherence property that objects occupy a small extent of space by associating with each object a bounding volume of a shape that is cheaper to test for intersection, such as a sphere or cuboid. When performing an intersection test between two objects, their bounding volumes are tested first; if the bounding volumes are disjoint, then the objects must also be so, so the more expensive test is only performed if the first test returns true. The assumption of spatial coherence means that most bounding volume intersection tests will return false, so the number of expensive tests performed is greatly reduced.

Clearly the tighter the fit of the bounding volume to its object, the better performance improvement it will give. A tradeoff is required between tightness of fit and other factors, such as the amount of storage required, ease of computation (and recomputation if the configuration of the object changes, e.g. it rotates) and of intersection testing. One important and widely used bounding volume is the axis-aligned bounding box or AABB, which is a cuboid aligned with the principal axes of the scene. This can be stored using only six scalars – the endpoints of the box’s projections on each of the

principal axes – and tests for intersection are similarly cheap; computation effort compares favourably with other common bounding volumes (such as spheres and arbitrarily oriented bounding boxes). AABBs do not give the tightest fit, but the cheapness of the required operations makes them highly suitable for many applications.¹

2.3.3 Baraff’s algorithm

The CD algorithm used in my project is an incremental ‘sweep-and-sort’ algorithm due to Baraff [Bar92], as presented in [vdB04] §§5.4.1-2. It keeps track of all pairs of intersecting AABBs in a scene with a nearly linear amortised time complexity.

For Baraff’s algorithm, the bounding box endpoints are maintained in three sequences – one for each axis – sorted by the endpoint’s coordinate. The sorting is initially done as a preprocessing step before the simulation begins. Thereafter, each time the scene evolves and the objects move, their AABB endpoints will be updated, which will cause some endpoints to be in incorrect positions within their sequences. However, the assumption of frame coherence means that the sequence will still be *nearly* sorted, and the ordering can therefore be restored in linear time by the application of *insertion sort*.

Of course, as well as being numbers in a sequence, the endpoints also represent intervals in space; for two AABBs to overlap, it is necessary and sufficient for their intervals projected onto each of the three axes to overlap. Insertion sort proceeds by iterating along the sequence until an element less than its predecessor is found, whereupon it is repeatedly swapped with its predecessor until it is greater than all elements before it; sorting then resumes from the position above that which was pushed down. The key to Baraff’s algorithm is that if the lower-bound endpoint of one AABB and the upper-bound endpoint of another are swapped during the sort, then the overlap status of those two intervals must have changed: they have either begun or ceased overlapping. When two intervals on one axis begin to overlap, their boxes are tested for intersection on the other two axes: if they intersect, they are added to the set of intersecting AABBs (which is also effectively a set of possibly intersecting objects). When two intervals stop overlapping, and if their boxes overlap on the other two axes, then the box pair is removed from the intersecting set.

If n is the number of objects in the scene, k is the number of intersecting box pairs and c is the number of interval pairs whose overlap status change in a particular iteration, then the time complexity of Baraff’s algorithm can be analysed as follows. Each pass of insertion sort will perform $O(n)$ swaps. At most c of those swaps will result in a box pair being inserted

¹ See [vdB04] §5.3.1 for a discussion and empirical comparison of several bounding volume strategies, including AABBs.

into or removed from the result set. If the set is represented by a balanced binary search tree then these insertions or deletions will take $O(\log k)$ time on average, and $O(k)$ time in the worst case. The time complexity of the algorithm is thus $O(n + ck)$.

Once Baraff’s algorithm has updated the set of intersecting AABB pairs for a frame, it remains only to iterate through the set, performing exact intersection tests on the corresponding objects. There are k of these tests, which is still $O(n + ck)$, so this is still the overall time complexity class of the collision detection algorithm.

In the worst case, when all objects in the scene collide simultaneously, c and k will both be equal to n , so this reduces to $O(n^2)$ as before. However, the assumption of frame coherence means that c will usually be much less than n , and spatial coherence means that k will also be much less than n , so that when these properties hold the algorithm should run in close to linear time.

It should be mentioned that for the *first* frame, where no assumptions can be made about how sorted the endpoints are initially, the initial insertion sort will in general revert to its average-case $O(n^2)$ -time performance rather than the linear-time best case. The run time of Baraff’s algorithm in the first frame will thus also be $O(n^2)$. This means that collision detection for the first frame will typically take significantly longer (for large scenes) than subsequent frames. However, since only the first frame is affected, this can be considered a preprocessing step, and since it does not affect the performance of the simulation thereafter, it is not considered a major disadvantage. If the initial delay is undesirable, it can easily be reduced by running an initial pass of a sorting algorithm such as heapsort or mergesort with a guaranteed worst-case performance of $O(n \log n)$.

2.4 A note regarding clean-room practices

Collision detection is a field in which a large number of high-quality implementations have already been developed, including a number of open source projects. As any number of recent lawsuits (notably SCO vs. IBM 2003) demonstrate, what might be termed ‘code pollution’ is a serious matter, so I take this opportunity to state – in addition to the signed Declaration of Originality at the start of this document – that I have not read any of the code from any of these projects or incorporated any material from them into my work.

Of particular note is the SOLID collision detection library. The book [vdB04] is a general discussion of matters relating to collision detection, but it is also a description of the design of SOLID. While this book was very useful in an inspirational and didactic role during the development of my project, I have not read the penultimate chapter (containing details of the

implementation of SOLID), and have not even broken the seal of the book's accompanying CD which contains the SOLID source code. All work on the project remains my own.

Chapter 3

Implementation

3.1 General features

When designing and implementing this project, I had a number of goals in mind. Primary was, of course, to produce a program that would function according to the project requirements. I also attempted wherever possible to write code that was elegant and readable: partly because this facilitates debugging, refactoring and writing up, and partly because this is an important skill when developing in industry where others will need to read and understand one's code.

The project has a modular design: to a large extent its subsystems are independent and can be considered (and tested) on their own. Besides having comprehensibility benefits, this approach helps to make the program more easily extensible: for example, to add a new feature to the scene description language should not require making changes to any parts of the code besides the parser (unless of course the feature requires new behaviour, such as simulating a new physical phenomenon). Object-oriented design principles have been adhered to where appropriate.

While performance is a concern for this project, I have been mindful during development of Hoare's maxim¹ regarding premature optimisation, and have concentrated on writing code that works, and works correctly, before worrying unduly about efficiency issues. The modular nature of the design means that optimising and refactoring where necessary could be done with a minimum of disruption. I did, however, design and develop with an eye to efficiency, and so have avoided the use of wastefully expensive algorithms and data structures.

¹ "Premature optimisation is the root of all evil." – C. A. R. Hoare. Quoted by and often misattributed to D. Knuth.

3.2 Fundamental data structures

Inevitably in a modular design, certain data structures are fundamental enough to the nature of the problem that they need to be shared by several of the modules. In this case the main such data structures were the types of object in the scene, the Scene datatype itself and some utility mathematical classes.

3.2.1 Object types

I decided to model the types of scene object using an object-oriented class hierarchy (see Figure 3.1), to reflect the large extent to which different object types behave similarly. The base of the hierarchy is an abstract class `Object`, which defines methods common to all scene objects, such as `draw` (specifying how the object should be rendered), `evolve` (specifying how the object's state changes during one time step), and some associated with the object's state and physical properties (e.g. position and elastic coefficient of restitution). Some methods, such as `draw` and `evolve`, are *virtual*, so that concrete object subclasses can specify (using C++'s virtual dispatch facility) how they should behave.

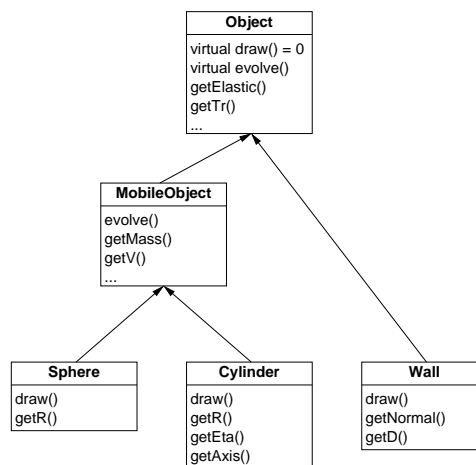


Figure 3.1: The class hierarchy for scene object types.

The simulation distinguishes between mobile and immobile objects: for example, it is useful to model a wall or floor as being stationary and immovable. The abstract class `MobileObject` derives from `Object` and implements common functionality for movable objects. Making this distinction at the class level obviates the need to store dynamical information for immobile `Objects`.

The specific shape classes inherit either from `MobileObject` or directly

from `Object`, and contain implementations of the mathematics, physics and graphics required to create, model and render them. In particular, a shape class will store geometrical parameters for its shape type. The implemented shape types are:

Sphere Represented by `Sphere`, which inherits from `MobileObject`. Its only geometric parameter is its radius.

Cylinder Represented by `Cylinder`, which inherits from `MobileObject`. Its geometric parameters are its radius, its halfheight (the distance along the axis from its centre to one end cap, conventionally denoted η) and its axis vector.

Plane An infinite, immovable plane, represented by the `Wall` class, which inherits directly from `Object`. Its only geometric parameter is its normal vector, but the `Wall` class additionally precalculates and stores the plane's least distance to the origin, and the position vector of the point for which that least distance is achieved, for use in calculations.

3.2.2 Mathematical classes

Many operations in the project required the use of mathematical entities not natively supported by the C++ language or standard library: the most prominent cases are the vector and quaternion (see §2.3.1) arithmetics. These therefore needed to be implemented as part of the project. C++ is designed to allow user-defined datatypes to be treated as much as possible like built-in types ([Str00]), so it was decided to implement vectors and quaternions as concrete classes; overloaded operators were provided to allow use of the classes to correspond more closely to mathematical intuition.

The `Vector` class supports the obvious mathematical operations ($+$, $-$, multiplication or division by a scalar), equality testing, scalar inner product $\mathbf{a} \cdot \mathbf{b}$ and vector cross product $\mathbf{a} \times \mathbf{b}$. Functions for finding the length $\|\mathbf{a}\|$ and squared length $\|\mathbf{a}\|^2$, normalising and pretty-printing a vector were also provided.

The `Quat` quaternion class similarly provides the obvious mathematical operations, and functions for finding the norm $|q|$, squared norm $|q|^2$ and multiplicative inverse q^{-1} . The scalar/vector representation of a quaternion $q = w + \mathbf{v}$ is used, in which arithmetic operations on quaternions can be represented in terms of scalar and vector arithmetic (for example, calculating the product of two quaternions can be done in terms of scalar multiplication and vector inner and cross products), so `Vector` is used in the implementation of `Quat`.

3.3 Frontend

The frontend module handles interfacing with the operating system and windowing system. It also encompasses the main function (in fact the `WinMain` function, as required by the Win32 API), the entry point of the program, and outer-level error handling. Because of its need to make use of fairly low-level system calls, this module is the only part of the project that is platform-specific (in this case, it is tied to the Win32 platform).

While cross-platform support was not an explicit goal of the project, the ability to run on multiple platforms is clearly a desirable feature (at least in the absence of market forces). To this end, care has been taken that *all* Win32-specific code should be encapsulated within the frontend module and, further, that the frontend should contain as *little* platform-independent code as possible: these measures ensure that porting the application to a different operating system should require very little in the way of rewriting or reimplementing.

Functions performed by this module include:

- Processing command-line arguments. Supported arguments are the name of a scene description file to read (if omitted, the program reads from standard input), and a `-n <num>` option that causes the program to exit after *num* frames have been rendered (by default it runs indefinitely until the user terminates it).
- Outputting timing statistics on each simulation run, to facilitate performance analysis at the testing stage. Times recorded include the render and simulation times for each frame, the total render and simulation times, and the total run time for all frames. Since a large degree of timing accuracy is required in order to time individual simulation and rendering steps, the `QueryPerformanceCounter()` Windows API call is used.
- Creating the window in which the OpenGL rendering is displayed.
- Handling exit conditions (the ‘Escape’ key is pressed or the ‘close window’ control is clicked).
- Error handling and reporting (the program exit codes are defined in a separate header file). Errors in the frontend (such as failing to create the application window or initialise OpenGL) are reported to the user via a message box; the application then terminates.

The frontend module also provides the top-level logical structure of the program.

3.3.1 Program structure

The `WinMain` function first sets up auxiliary facilities, such as opening input and output files and initialising the program timer. It then passes control to the parser module, which reads the input and builds the scene data structure. Next it initialises the rendering module, and opens the program window. Finally it enters the main program loop, where the program remains until it is terminated.

The main loop maintains the simulation and animation. The scene is evolved and rendered at discrete intervals, separated by a fixed virtual time step. The main loop first invokes the simulation module, causing it to update the positions of objects in the scene, detect and resolve any collisions, and move on one time step; it then invokes the renderer module to render the frame to screen.

3.4 Scene description language parser

This is the program module responsible for parsing the scene description language (SDL) and building the scene from this input. This module was implemented first, partly because it helped to clarify how some of the main data structures should be designed, but mainly because without an efficient method of input it would be difficult to test any other part of the program.

The SDL was designed to be simple, easily human-readable and -writable and fairly intuitive. It is specified as a context-free grammar in BNF form, as given in Appendix A. Informally, a scene description consists of zero or more object declarations, each of which has several attributes. A consequence of the language's simplicity is that it contains no control structures: for example, there is no way to express "two hundred spheres" besides explicitly declaring each one. This kind of behaviour can be easily produced by using an external script to generate the scene description, as described in §3.4.1 below.

The SDL parser is implemented using the freely available `flex` and `bison` tools, for lexer and parser generation respectively. Scene description parsing is done in a single, static pass before the main simulation loop begins. It is not possible to add or remove scene objects once the simulation has started. The parsing and scene-building process operates as follows:

1. The scene is represented as a list of objects, initially empty.
2. Each object declaration's parameters are read into an associative array (implemented using the C++ STL `map`). In fact two `maps` are used – one for scalar-valued parameters and the other for triple-valued parameters.

3. The declaration name (e.g. ‘sphere’) and parameters (e.g. ‘radius 1.0’) are passed to an object creation function `mkobj` which attempts to build an object from the declaration (see below). Object building will fail if the declaration is of an unknown type, or if some parameter values are missing or invalid.
4. If the object was successfully created, it is added to the scene.
5. If object creation fails, an error message is printed warning the user of the error in the scene description. Parsing then continues, omitting the erroneous declaration. I chose to take this approach, rather than halting the application on a parsing error, because omitting a declaration cannot cause the scene to be initialised in an invalid state, and because the user may be more interested in seeing the behaviour of the correctly specified parts of the scene than in tracking down syntax errors.

The object creation process is slightly complex because of the need to simplify adding new object types: it is based on the Abstract Factory Design Pattern. The module holds a mapping of declaration names to (concrete) factories, which themselves each contain definitions of what parameters that object type supports; for each parameter the factory also stores whether that parameter is required, a default value if it is not, and any constraints on the values it may take. Object creation proceeds as follows:

1. The `mkobj` function searches for a factory associated with the declaration name it was passed. If none is found, `mkobj` throws an ‘unknown declaration’ exception.
2. `mkobj` passes the declaration’s parameters to the factory that was found. The factory validates that all required parameters are present and that all parameter values satisfy the specified constraints. If any errors occurred, it throws an exception listing them.
3. The factory creates an object of the appropriate type from the information in the parameters and returns a pointer to it.

To add a new object declaration type to the language, one needs only to define a concrete factory (and thus the parameters) for the object type, then add the factory with an appropriate name to the declaration-name-to-factory mapping. Adding or modifying parameters for an existing object type is easier still, as only that object’s factory definition need be modified.

3.4.1 Automatically generating scene description files

Since the project reads in its scene description as simple text, there is no reason why this text should not be generated dynamically rather than being

read from a static file. In order to create some larger and more complex scenes on which to test the program, I use some simple Python scripts to generate larger scene descriptions than are practical to write manually. This approach brings a number of benefits.

First, it becomes easy to generate scenes with randomly varying initial conditions: for example, the outcome of a break in pool depends critically on the positions of the balls and on exactly where the cueball strikes, and by ‘jittering’ these quantities slightly it is possible to produce quite different (and, since in real life it is all but impossible to achieve precisely the same setup twice, arguably more realistic) results. The Python standard library includes facilities for pseudo-random number generation, including the generation of floating-point numbers from uniform and Gaussian distributions.

It is then easy to iteratively produce scenes containing many objects – such as a large number of initially randomly-positioned spheres – simply by using the control structures of the scripting language. It is also possible to modify the generated scenes by supplying command-line parameters to the scripts: for example, a script called with the parameter `-n 150` might generate a scene containing 150 spheres. This technique was very useful when benchmarking the project.

3.5 Renderer

3D scene rendering and animation was implemented using OpenGL. Care was taken to separate the rendering logic from the simulation logic, to ensure that each could be tested independently. High visual quality was not a main goal of the design, but certain graphical features (particularly lighting and Gouraud shading) were deemed necessary as making it much easier to observe the behaviour of the scene. Fortunately the OpenGL API provides these features in a convenient manner.

Rendering the scene requires rendering each of the objects within it. Each object class defines a `draw()` method specifying how that object type should be rendered:

- Walls are drawn as large quadrilaterals (`GL_QUADS`). Here ‘large’ means “extensive enough to appear indistinguishable from an infinite plane within the scope of the rendering viewport”.
- Spheres are drawn using the GL Utility function `gluSphere()`. Originally sphere drawing was implemented using a recursive tetrahedron vertex subdivision algorithm, as mentioned in the Part IB Graphics lecture course, but it was found that for small recursion levels the visual quality of the resulting approximation was poor, and with a higher depth of recursion the algorithm became very slow. The GLU

alternative uses a simpler ‘latitude-longitude’ subdivision scheme, and produces both visually superior and faster results.

- Cylinders are divided radially into a fixed number of segments, and drawn using `GL_TRIANGLE_FANs` for the end caps and a `GL_QUAD_STRIP` for the curved surface. The polygon vertices are precomputed and stored, rather than being recomputed each time a cylinder is drawn.

When the rendering module is initialised, it also initialises resources such as the GLU ‘quadric object’ required by the `gluSphere()` function and others, and the precomputed vertices of the cylinder approximation.

The rendering subsystem was substantially completed before work began on the simulation module. In particular, the renderer was tested on static scenes before implementing support for animation, to ensure the graphical display reliably reflected the internal state of the scene.

3.6 Simulation

The simulation module encompasses several aspects of the project: animation of the scene, simulation of physical phenomena (such as gravity and collision response), and the collision detection subsystem.

To start with, in order to more fully test other modules (particularly the parser and renderer), a very simple prototype simulation module was implemented which neglected collisions between objects and simply animated them. Gravity was included in this simple model, as it was very simple to implement (a constant downward acceleration of 9.8ms^{-2}) and constituted a test of the general framework for physical simulation. Since the renderer’s viewport is fixed, this caused objects in the scene to rapidly fall out of view, so an arbitrary ‘floor’ constraint was imposed to keep objects above a certain height and in the viewing frustum. This constraint was of course removed when the simulation module was implemented more fully (an equivalent constraint can now be achieved by modelling the scene’s floor by a horizontal plane).

3.6.1 Scene animation

Each scene object maintains its own internal state: all `Object`s store their position \mathbf{x} and orientation q , and `MobileObject`s also have mass m , velocity $\mathbf{v} = \dot{\mathbf{x}}$ and angular velocity ω . `MobileObject`s additionally keep a record of any forces (such as gravity) acting on them. The `draw()` methods defined by each object type take the position and orientation values into account when calculating how to render the object.

The animation strategy is defined by the virtual `evolve` method declared in `Object`. Since immobile objects do not evolve over time, `Object`’s own

`evolve` method is empty. `MobileObject` overrides this to enable mobile objects to be animated by using the objects' dynamical quantities and state in the current frame to calculate their values in the subsequent frame, so that the next rendering pass will render the objects in their new positions.

Calculating the values of these dynamical quantities in the next frame amounts to numerical solution of a differential equation for each object. Numerical integration is a major topic of computer science in its own right and many extremely sophisticated techniques are known and used for applications where numerical accuracy is critical. For this project, as previously stated, numerical accuracy is not a primary goal; realistic-looking results can be obtained by using the simple Euler method for numerical integration with a small enough time-step h . The simulation therefore calculates the new position of each object using the equation

$$\mathbf{x}(t+h) \approx \mathbf{x}(t) + h\dot{\mathbf{x}}(t) \quad \text{for small } h.$$

The object's new orientation is calculated using Equation 2.2 and another application of Euler integration:

$$\begin{aligned} q(t+h) &\approx q(t) + h\dot{q}(t) && \text{for small } h \\ &= q(t) + h\frac{1}{2}\hat{\omega}q(t). \end{aligned}$$

A quaternion representing orientation must be a unit quaternion (i.e. one for which $q^*q = 1$), and care must be taken numerical drift caused by the above angular velocity integration does not corrupt this property of the orientation. To prevent this, after the integration step, the new q is divided by its norm ($q' = q/\|q\|$).

3.6.2 Collision response

The simulation models collisions between bodies as either elastic or inelastic; bodies are considered rigid and atomic (i.e. do not deform or disintegrate as a result of a collision). The user specifies as part of the scene description whether a particular object behaves – i.e. whether objects colliding with it will rebound – elastically or inelastically by means of the `elastic` object parameter: a value of 0.0 represents a perfectly inelastic ('sticky') body, and a value of 1.0 (the default) represents perfect elasticity. In the case when both colliding objects are mobile and perfectly elastic, momentum and kinetic energy are conserved; elasticity values other than unity are modelled by scaling the resultant velocities of the objects.

Collision response is implemented as a general routine which is independent of specific object geometries: it therefore must be passed information regarding where the collision took place. Specifically, it is passed an `IntersectInfo` object, which gives the position vectors of *witness points* to

the intersection: these are points on each object which penetrate the other, typically the points of greatest penetration, thus serving as ‘witnesses’ that the intersection of the two objects is non-empty. The `IntersectInfo` object also records the greatest penetration depth.

The procedure to calculate the collision response is as follows:

1. Treat the vector joining the witness points as the *line of action* – that is, the direction in which an impulse should be applied.
2. Calculate the *incident velocity*: this is that component of the relative velocity of the two objects which is parallel to the line of action. If the objects are rotating, then their rotation may alter their effective linear velocity at the point of impact, so the quantity $\mathbf{r} \times \omega$ (where \mathbf{r} is the vector from the object’s centre of mass to the point of impact, and ω is the object’s angular velocity) is added to the objects’ linear velocities before the incident velocity is calculated.
3. If the incident velocity is *negative*, then the objects are already moving apart, so no further response should take place. (This prevents a problem whereby interpenetrating objects can become ‘stuck’ to each other if their intersection is detected twice in each frame.)
4. Displace the objects apart, by half the penetration depth each, along the line of action to enforce the nonpenetration constraint.
5. Calculate the new velocities and angular velocities for the objects. This is done by calculating the solutions to vector equations derived from the principles of conservation of momentum and energy: the results will therefore depend on the incident velocity and the masses of the two objects. The changes in velocity and angular velocity for the first object are scaled by the second object’s elasticity value, and vice versa.
6. Update the objects’ velocities and angular velocities with the new values.

The above procedure is used when both objects are mobile. If one object is immobile, the procedure followed is similar, but simpler (effectively that object’s mass can be considered infinite, which simplifies most of the equations), and only the mobile object is displaced.

3.6.3 Intersection tests

Collision detection requires identifying which pairs of objects are intersecting in a given frame. Moreover, since the general collision response function used requires to be passed an `IntersectInfo` object recording witness points of the collision, these witness points must also be calculated and returned.

The approach taken for intersection tests was to use explicit algebra. For each pair of shapes, the equations defining them were solved simultaneously, to give closed formulae for certain quantities which can easily be used to check for intersection or return the closest points. The program then simply uses these formulae to calculate the quantities and obtain the required information.

The equations and their derivations are fairly tedious and similar, so the procedure for sphere-sphere tests is given as a representative example:

1. Let the two spheres have centres $\mathbf{c}_1, \mathbf{c}_2$ and radii r_1, r_2 . Let $\mathbf{v} = \mathbf{c}_2 - \mathbf{c}_1$ be the vector joining the centres. The spheres intersect if the sum of their radii exceeds the distance between their centres, that is if $r_1 + r_2 > \|\mathbf{v}\|$.
2. If they do intersect, the required witness points are found on the line joining the centres. Let $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$; then the witness points are $\mathbf{p}_1 = \mathbf{c}_1 + r_1\hat{\mathbf{v}}$ and $\mathbf{p}_2 = \mathbf{c}_2 - r_2\hat{\mathbf{v}}$.

If the objects are disjoint, a *separating axis* is also calculated: this is the normal to a plane which separates the two objects. The separating axis is not currently used in the project, but a well-known technique for improving the efficiency of intersection tests is to cache the separating axis for each object pair, and then to use this as an initial ‘quick-reject’ test in future frames. If the objects’ projections onto the axis do not overlap then the objects cannot intersect (and the axis is still a separating axis), so the explicit intersection test can be skipped; this axis-overlap test is typically much cheaper than the full test for complex shapes.

3.6.4 Naïve “all pairs” collision detection algorithm

The $O(n^2)$ “all pairs” algorithm, as described in §2.3.2, was used as a prototype collision detection algorithm in order to test the project modules working as a whole. Its implementation is very simple (which is, of course, the reason for choosing it).

As described in §3.4, the scene is represented by a list of objects. The naïve algorithm simply performs two nested iterations through the list, testing each resulting pair of objects for intersection (care is taken not to test an object for intersection with itself). If the test returns true, the object references and information from the test are passed to the response function.

It can be seen that this procedure will end up testing each pair of objects twice. One way of preventing this would be to store a table of all object pairs and mark which have already been tested; however, with n objects in the scene this table would contain n^2 entries and use an extravagant amount of memory for large scenes. The test in the collision response function for a negative incident velocity will prevent any unphysical behaviour resulting

from detecting the same collision twice, and in any case since the response function displaces the objects to enforce nonpenetration, the second intersection test on a given pair of objects should always fail. Since this flaw introduces no incorrect behaviour, it was decided acceptable: it is hardly time-efficient, but efficiency is not the goal of this algorithm.

3.7 Baraff's algorithm

Once sufficient testing had been carried out using the “all pairs” algorithm to verify that all modules were working as expected, work began to replace it with the more sophisticated ‘sweep-and-sort’ algorithm due to Baraff (see §2.3.3). The implementation used here is based on the discussion of Baraff's algorithm in [vdB04].

3.7.1 Axis-aligned bounding boxes

Baraff's algorithm operates on axis-aligned bounding boxes (described in §2.3.2), so it was necessary first to implement an efficient AABB representation. While it is technically valid to simply assign all objects an infinite AABB, this defeats the point, and to achieve tight spatial bounds a routine must be provided to calculate the AABB for each supported shape type. Clearly for mobile objects the AABB must move with the object, and for objects other than spheres the AABB will also need either to adapt as the object rotates, or be large enough to contain the object in any orientation (here the latter approach is used).

Because the AABB depends on the shape geometry in this way, AABB calculation is implemented as a pure virtual method in the `Object` class, overridden by all shape classes. The AABBs for the three supported shape types are calculated as follows:

- For a `Wall`, if the plane's normal is parallel to one of the world axes, the AABB occupies the interval $\pm[-\infty, d]$ along that axis (where \pm signifies that the interval is reversed if the normal is antiparallel to the axis, and where d is the projection of the wall onto the axis), and is infinite along the other two axes. Otherwise, the AABB is necessarily infinite.
- For a `Sphere` with radius r and centre \mathbf{x} , the AABB is simply a cube of side $2r$ and centre \mathbf{x} .
- For a `Cylinder` with radius r , halfheight η and centre \mathbf{x} , the AABB is a cube of side $2\sqrt{r^2 + \eta^2}$ and centre \mathbf{x} : this is the AABB of the smallest sphere fully enclosing the cylinder, and ensures that the AABB contains the cylinder in any orientation. The cylinder's diagonal $\sqrt{r^2 + \eta^2}$

is calculated during scene building and does not change as the cylinder moves or rotates, so need not be recalculated each frame.

The sorting stage of Baraff's algorithm requires the AABB endpoints to be held in sequences, one for each axis. To avoid the complication and overhead of copying endpoints in and out of these sequences, [vdB04] suggests having these endpoint sequences *be* the AABB representation. AABBs then store the indices of their endpoints in these sequences. As a further optimisation, [vdB04] suggests an extra level of indirection: the indices into the endpoint sequences are themselves stored in a sequence, in a standard order

$$x_{0,min}, x_{0,max}, y_{0,min}, y_{0,max}, z_{0,min}, z_{0,max}, x_{1,min}, \dots$$

so that the index of an AABB's first endpoint index is equal to $6i$, if i is the index of the AABB itself.

The endpoints themselves are represented by a C++ `struct` containing the endpoint value, a flag indicating whether the endpoint is a lower or upper bound, and a pointer to the endpoint's AABB.

The sequences of endpoints and endpoint indices are implemented as STL `vectors`: this data structure is optimised for efficient random access to elements. Endpoints are stored in three `vectors` `AABB::xendpts`, `AABB::yendpts` and `AABB::zendpts`, and their indices are stored in the `vector` `AABB::indices`. Each AABB itself then stores only its own index and an `owner` pointer to the object whose AABB it is. To hide the complexity of this structure, an `AABB` class was defined with a simple interface and operations such as `minX()` and `maxZ()` to retrieve the endpoints, and `intersect()` to test for intersection with another AABB.

Because this representation of AABBs is somewhat nonintuitive, a small program was written to run a set of tests on the AABB class independently of the rest of the system. This verified, for example, that the AABB operations gave the correct results and that the AABB representation behaved correctly when the endpoint `vectors` were sorted.

3.7.2 Algorithm details

The first step in implementing Baraff's algorithm was to implement the modified version of insertion sort that it requires. The main noteworthy aspect of this is the procedure when swapping adjacent elements. First the endpoints themselves (in, say, the `xendpts` vector) are swapped; then their respective indices in the `indices` vector are also swapped, so that the affected AABBs will still refer to the correct elements.

The `swap()` procedure also checks whether the two endpoints being swapped are of different types – that is, one is a lower bound and the other is an upper bound. If this is the case, their associated AABBs are tested for overlap on the other two axes. If they are found to overlap, then the AABBs

must have either started to overlap or ceased to overlap. If the smaller of the endpoints just swapped is a lower bound of an interval, then the AABBs now overlap; otherwise they are now disjoint. In the former case, the pair of objects owning the AABBs is added to the set of possible collisions; in the latter, it is removed from the set.

The set of possible collisions must observe set semantics – i.e. it should never contain more than one copy of a given pair of objects. The case that both pairs (O_1, O_2) and (O_2, O_1) might be added to the set is dealt with by ordering the pair elements by pointer comparison of the AABBs: since their location in memory does not change during the course of the program, and exactly one object owns each AABB, this ensures a canonical ordering on objects. The set itself is implemented using the STL `set` data structure, which uses a balanced red-black tree representation to ensure fast – $O(\log n)$ time – element access, and enforces the uniqueness property on its elements.

The collision detection phase of the main program loop then simply sorts each of the three endpoint vectors using insertion sort (with the modified swap semantics described above), then iterates through the set of possible collisions, performs exact intersection tests on the object pairs, and initiates the response routine if the test returns true.

Chapter 4

Evaluation

The project evaluation was carried out against each of the requirements specified in §2.1. Because the nature of the project requirements varies, different methods of evaluation were used for each requirement.

The functional requirements #1 (reading in a scene description) and #2 (displaying an animated simulation of the scene) were implemented early on: these functions are essential to the operation of the project, and so were tested independently and rigorously from the early stages of implementation. The scene description language recognised by the parser module is adequate to express some fairly complex scenes (sometimes aided by a generating script: see §3.4.1). The scene rendering is visually appealing and its 3D view effectively conveys the behaviour of the objects in the scene. Thus I conclude that requirements #1 and #2 have been met.

It is worth noting that the user interface to this project is fairly minimal. There is no requirement for user interaction while the simulation is running, so the sole user input (besides terminating the program) is to supply the scene description. No graphical user interface for this process is provided: this is partly because designing a usable GUI for 3D scene modelling would be a substantial project in itself, and omitting it made it possible to concentrate on fulfilling the main project requirements, but also because it leaves the user free to choose his or her own method of input. The scene description can be entered directly into the program via the console; created in the user's choice of text editor and saved as a file; or even dynamically generated by a script, as discussed in §3.4.1.

Regarding requirement #6 (realistic behaviour), it is important to recognise that certain types of behaviour are altered because of the modelling decisions made to simplify the project. For example, in the “pool break” scene, the front ball in the triangle jumps slightly when the cueball hits: this is because the cueball is slightly smaller than the coloured balls, and the behaviour is entirely consistent with the model of frictionless interaction. On a real pool table, however, the cueball is *rolling* when it strikes the lead

ball, and the friction between the two causes the cueball to ‘climb’ slightly up the front ball. While not modelling this does represent a loss of realism, the ‘climbing’ phenomenon is quite hard to notice in real life, and the most obvious characteristic behaviour of the pool break – the scattering of the triangle – is faithfully simulated by the project.

Bearing this in mind, the method of evaluation for requirement #6 was to demonstrate the project operating on various sample inputs to several human observers, including both non-technical viewers and fellow computer scientists. There was a consensus among those demonstrated to that the behaviour of the simulation was realistic and physical, within the constraints of the model, for all simple scenes and many of the more complicated examples. (Some of the more complex scenes exhibit some unphysical behaviour, as will be described in §4.1 below.) The project is thus judged to have met requirement #6.

The evaluation of the remaining requirements will be discussed in the following sections. Section 4.1 gives an overview of the simulation features that have been implemented. Section 4.2 will describe some of the example scenes that were designed to test the project, and show results of their simulations, in order to evaluate requirement #7 (demonstrability on various scenes). Section 4.3 will explain how the program’s performance was measured and present the results of tests used to evaluate requirements #3 (real-time animation), #4 (scalability) and #5 (comparison of CD algorithms), which are essentially performance criteria.

4.1 What has been achieved

The project, as specified, implements a simulation of rigid body motion. It includes a parser to read in a description of a scene from a named file or from standard input, and a 3D renderer that produces an animation of the scene being simulated. Shape types supported are infinite planes, spheres and cylinders.

Intersection tests have been implemented per shape type pair. The following intersection tests have been fully implemented:

- Sphere with sphere
- Sphere with plane
- Sphere with cylinder
- Cylinder with plane

The intersection test between cylinders and cylinders has been partially implemented. In the case that the cylinders’ axes are parallel or antiparallel, intersection testing is fully supported (hence the ‘pyramid’ and ‘table’ example scenes described in §4.2), but the case of arbitrarily aligned cylinders

proved too difficult to implement within the time constraints of the project. However, the tests that have been implemented are adequate to support a wide variety of scenes.

The physical features simulated are:

- Gravity, simulated simply by a constant downward acceleration, rather than the full (and almost certainly unnoticeable) inverse-square attraction between all pairs of objects, since the objective is to simulate familiar scenes realistically.
- Spring forces were implemented, as a test for the general simulation infrastructure, but the scene description language does not currently support describing scenes containing springs.
- Linear collision response (i.e. the exchange of linear momentum of colliding bodies).
- Angular collision response (i.e. the effect of collisions on the colliding bodies' angular momenta) has been partially implemented, but a full implementation of angular collision proved too large a task within the scope of the project. As the “spinning cylinder” example scene demonstrates, the partial implementation still allows realistic scenes to be simulated.

The approach to simulating collision, and particularly the approximation of contact forces by repeated ‘microcollisions’, is similar to that used in [MC95]. The approach used here has the weakness that if an object is involved in multiple simultaneous collisions, unphysical behaviour may sometimes (but not always - see §4.2.5) occur: in particular the nonpenetration constraint on objects may be violated in the case that collisions occur simultaneously on opposite sides of an object. With more time this problem could be alleviated, for example by tracking all collisions in which an object takes part, then finding a displacement for each object involved that satisfies nonpenetration, but this would almost certainly incur a performance loss. The advantage of the approach used is its simplicity and efficiency.

Finally, two collision detection algorithms have been implemented: the naïve “all pairs” algorithm described in §2.3.2, and the more sophisticated “sweep-and-sort” algorithm due to Baraff, described in §2.3.3.

4.2 Example scenes

4.2.1 Pool break

This scene is a simulation of the opening shot in a game of pool, as depicted in Figures 4.1 and 4.2. Fifteen balls – seven red, seven yellow and one black – are set up on a horizontal plane (representing the pool table) in a

triangle formation, and the white cueball, slightly smaller than the rest, is fired toward them at high speed.

As in a real pool break, the scattering of the balls depends critically on their initial positions, and on precisely where the cueball strikes. To simulate this unpredictability, these quantities are ‘jittered’ slightly by drawing their values from a Gaussian distribution with a small variance.

4.2.2 Brownian motion

As mentioned in the Introduction, according to the standard kinetic theory of gases, the behaviour of an ideal gas can be modelled by a collection of rigid, perfectly elastic spheres. This scene simulates the behaviour of two such gases under gravity, enclosed at opposite sides of a perfectly elastic container (modelled by one horizontal and four vertical planes). Figures 4.3 and 4.4 show the scene containing 500 particles.

All of the particles are identical, and have randomised initial positions and velocities. This means they effectively constitute a model of *Brownian motion*, an explanation of how gases modelled by the kinetic theory expand and interact with other particles. As the particles move, the two gases gradually ‘diffuse’ and merge into each other. To make the random behaviour easier to observe, one of the particles is coloured blue: it can be seen to execute a 3D “random walk” as it collides with others.

4.2.3 Spheres and pyramid

This scene contains a ‘pyramid’ composed of vertical cylinders of various radii, and a large number (500 in the screenshots) of coloured spheres with random positions, velocities and radii, which fall onto and rebound off the pyramid: see Figures 4.5 and 4.6. Later a larger, more massive sphere drops onto the top of the pyramid, scattering the smaller spheres further.

This scene demonstrates the simulation of collisions between solids other than spheres. It also demonstrates the unwanted behaviour that can occur (see §4.1) when a given object is involved in several simultaneous collisions: if many of the spheres impact simultaneously, or if many are resting on one cylinder, one can occasionally see the cylinders sink slightly into the floor or each other.

4.2.4 Spinning cylinder

In this scene (see Figures 4.7 and 4.8) there are a rotating cylinder, a white sphere and a more massive red sphere. First the cylinder’s rotation causes it to strike the white sphere, propelling it away; then the red sphere’s velocity carries it into the cylinder, much reducing its angular momentum and also giving it a linear velocity to the left.

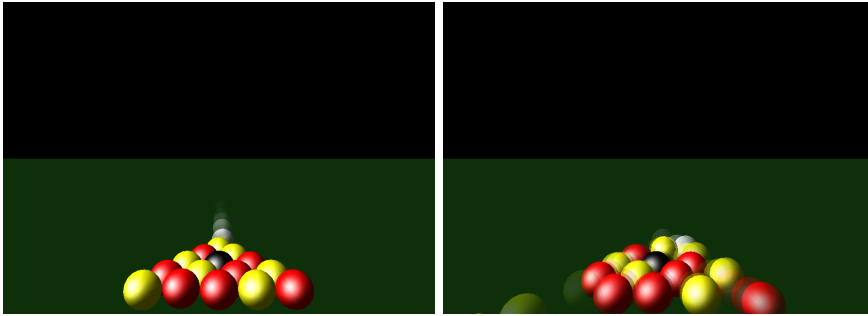


Figure 4.1: “Pool break” scene. (a): the cueball’s initial velocity carries it toward the triangle. . .

Figure 4.2: “Pool break” scene. (b): . . .and when it impacts, the red and yellow balls are scattered.

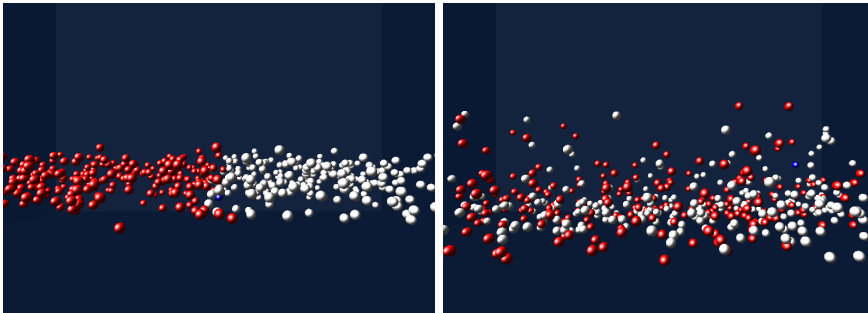


Figure 4.3: “Brownian motion” scene. (a): initially the white and red ‘gases’ are on opposite sides of the container.

Figure 4.4: “Brownian motion” scene. (b): as their particles move and collide randomly, the gases ‘diffuse’ into each other.

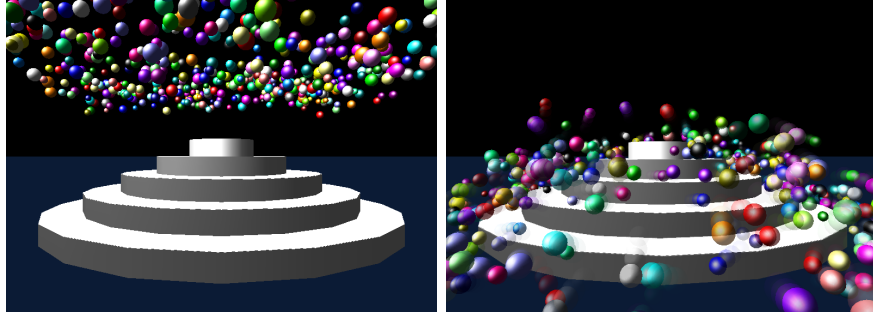


Figure 4.5: “Pyramid” scene. *(a)*: assorted spheres fall. . .

Figure 4.6: “Pyramid” scene. *(b)*: . . . and rebound off the pyramid.

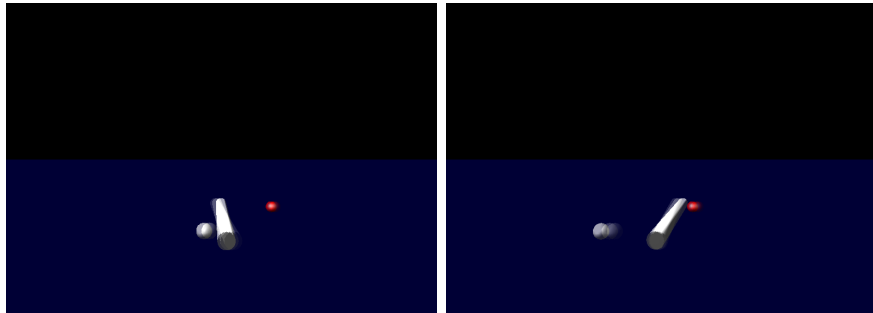


Figure 4.7: “Spinning cylinder” scene. *(a)*: the cylinder spins into the white sphere and sends it flying.

Figure 4.8: “Spinning cylinder” scene. *(b)*: the heavier red sphere hits the cylinder and counters most of its spin.

This scene demonstrates collisions involving both angular and linear momentum: it shows that a body that has no linear velocity, but is spinning, can impart (linear) momentum to another body, and that an object hitting another can change its angular momentum.

4.2.5 Table

This scene models a table, composed of five cylinders, and two spheres moving across the tabletop: see Figure 4.9.

4.3 Performance evaluation

In order to evaluate the project's performance, I carried out 'benchmark' tests, running the program on several representative scenes, each repeated with varying numbers of objects. Performance data were collected from the per-frame and all-frames timing statistics output by the program. To allow large numbers of tests to be run and their statistics collated conveniently, a Python script was written to automate the test process (including support for automatically varying test parameters, such as the number of objects in each test).

The benchmark results serve several purposes. They allow the overall performance of the program to be gauged, in order to evaluate the complexity of scenes able to be simulated in real time. They allow the two implemented collision detection algorithms to be compared empirically, to test whether their theoretical differences are borne out. By presenting detailed results from certain test runs, it is also possible to gain some insight into the workings of both algorithms.

For all of the benchmarks below, the project was compiled using the GNU g++ compiler, with options `"-s -O2 -mno-cygwin -march=x86-64 -mfpmath=387 -fomit-frame-pointer -ffast-math"`.

4.3.1 Real-time performance

As specified in item 3 of §2.1, I define real-time performance to mean that the simulation runs with a frame rate of at least 30 frames per second (fps). As will be seen, simply achieving this frame-rate is a fairly weak condition, since for simple scenes the performance is well above this figure: I therefore investigate what size of scene (in terms of the number N of objects therein) can be simulated within this real-time constraint. For these tests Baraff's algorithm was used as the CD algorithm.

For this test I use three scenes, each of which is generated by a script so that the number of objects can be varied. For each scene I give a graph of measured frame-rate against number of objects.

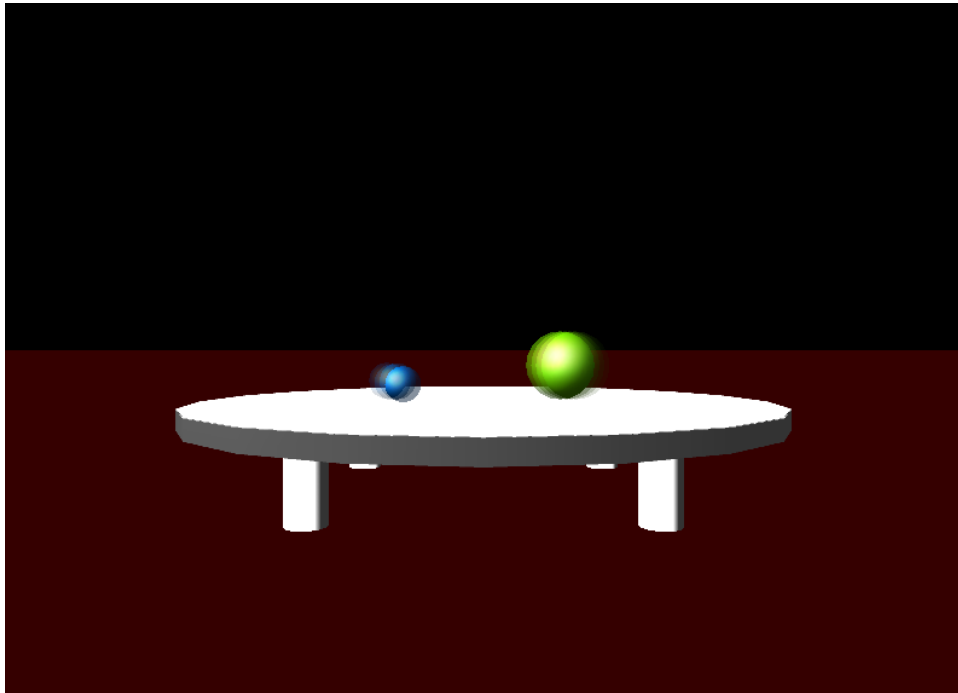


Figure 4.9: ‘Table’ scene. The two spheres land on the table, then move across it, colliding with each other in the process.

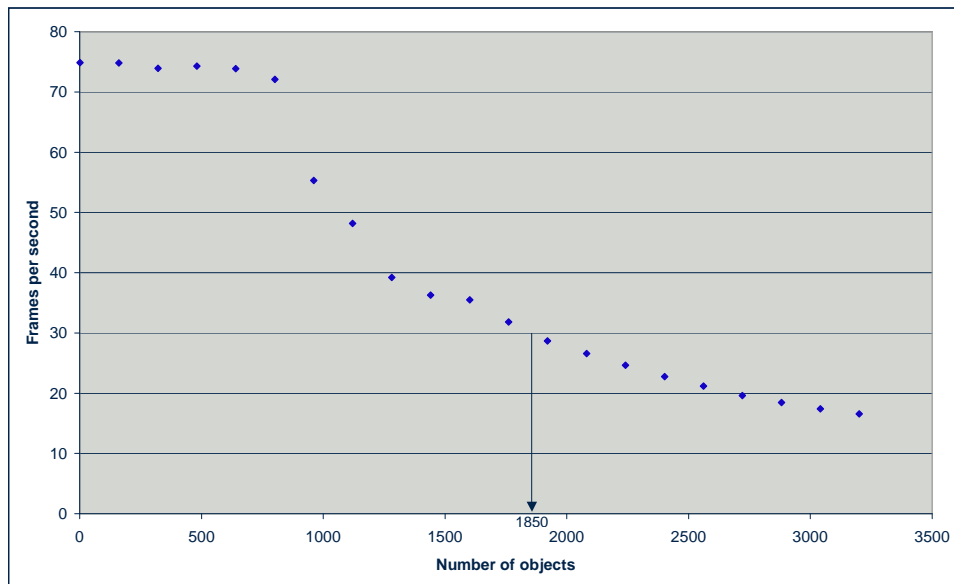


Figure 4.10: Graph of frame-rate against number of objects for the “multiple pool breaks” scene.

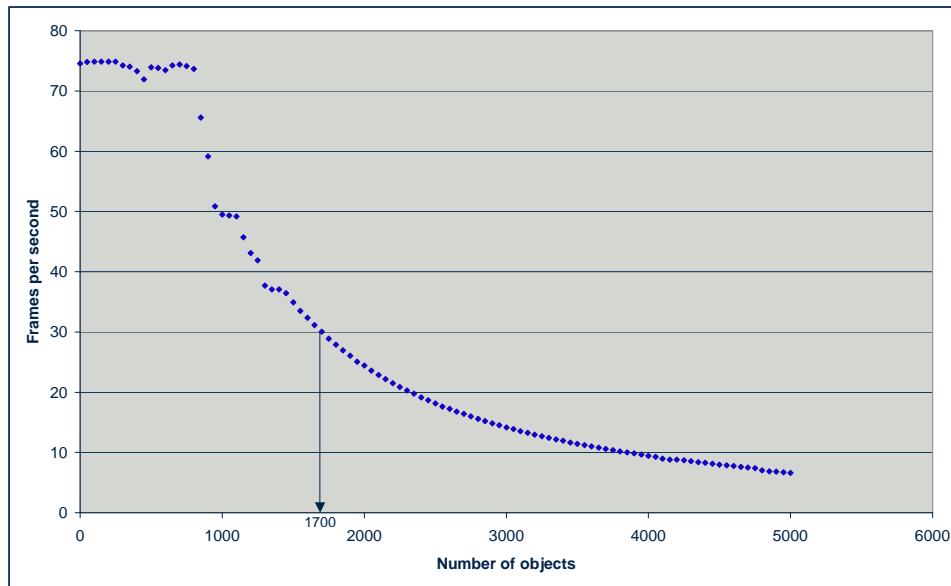


Figure 4.11: Graph of frame-rate against number of objects for the “Brownian motion” scene.

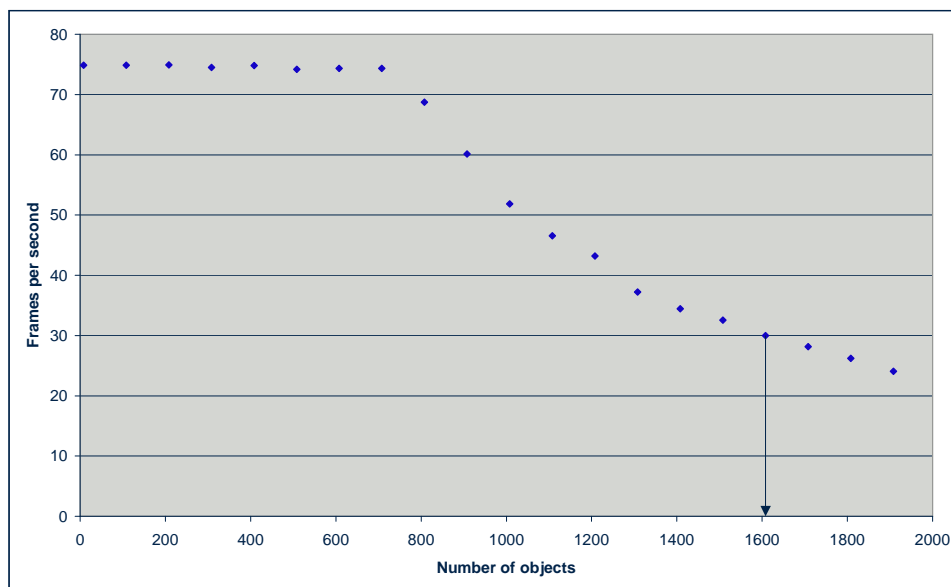


Figure 4.12: Graph of frame-rate against number of objects for the “spheres and pyramid” scene.

- **Pool breaks** (Figure 4.10): this scene is simply several copies of the pool break scene described in §4.2.1 distributed randomly over a large area (see Figure 4.13). Real-time animation is achieved for up to 1850 objects.
- **Brownian motion** (Figure 4.11): as described in §4.2.2. Real-time performance is achieved for up to 1600 objects.
- **Spheres and pyramid** (Figure 4.12): as described in §4.2.3. Real-time performance is achieved for up to 1600 objects.

A phenomenon worth noting is that all three graphs appear to ‘threshold’ at 75fps. No frame-rate limiting logic has been built into the program, so it is assumed that this effect is due to a limit imposed either by OpenGL or by the graphics card driver.

From these results it can clearly be seen that the real-time requirement has been satisfied. Not only is real-time performance achieved, in all three cases it is also achieved for scenes containing well over a thousand objects. The scalability requirement (#4) specifies several hundred objects as the threshold for acceptance: clearly this requirement is also more than fulfilled.

4.3.2 Comparison of collision detection algorithms

In this section I will compare the naïve “all pairs” collision detection algorithm (described in §2.3.2) with Baraff’s “sort-and-sweep” algorithm (see §2.3.3). The theoretical analysis suggests that Baraff’s algorithm should be the better performing of the two; this section tests that empirically.

For a quick answer to the question, see Figure 4.14. Comparing this with Figure 4.12 shows that, on the same scene, Baraff’s algorithm achieves real-time performance for over four times as many objects as the naïve algorithm can manage.

Figure 4.15 shows why. The simulation time is the total time spent in the “evolve scene” part of the main program loop (the time to render the scene is the other major time cost), and time spent performing intersection tests and collision detection is the dominant factor in this measurement. The naïve algorithm’s runtime grows far faster with the number of objects than does Baraff’s.

This graph shows the mean simulation time per frame. To achieve 30 frames per second, the program must spend no longer than $1000/30 \approx 33\text{ms}$ on each frame. Figure 4.15 shows that, for more than 400 objects, the naïve algorithm spends more than this on the simulation step alone.

Two trend lines are shown on the graph: both are parabolic curves, i.e. quadratic. The line fits the data for the naïve algorithm very well, as expected: theory predicts that this algorithm’s performance is $O(n^2)$ in all



Figure 4.13: “Pool breaks” scene.

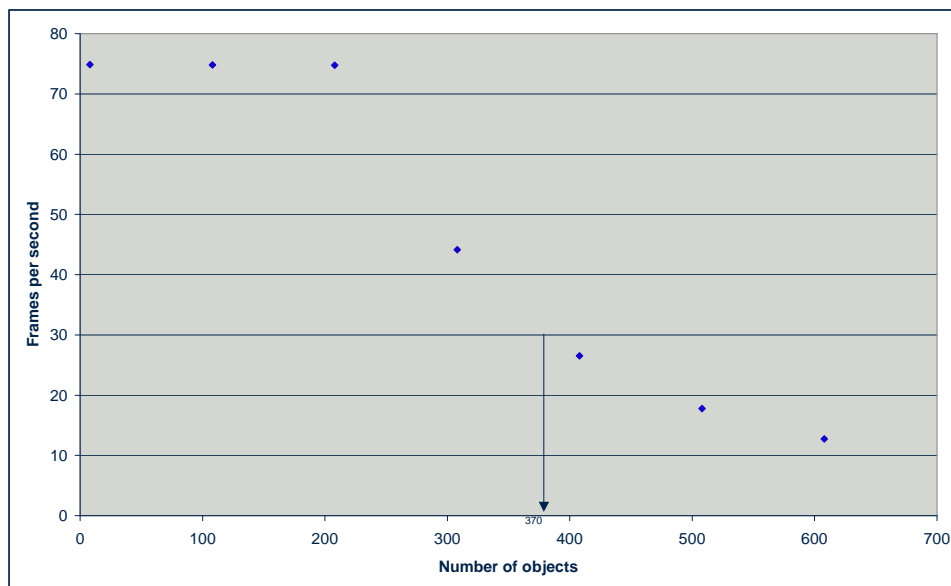


Figure 4.14: Graph of frame-rate against number of objects for the “spheres and pyramid” scene, using the naïve “all pairs” collision detection algorithm.

cases. However, since the analysis of Baraff's algorithm predicted *linear* time, the quadratic curve requires some explanation.

The average simulation time is produced by averaging over the *entire* simulation run. At the beginning of the simulation of this scene, a large number of collisions occur in a short space of time as the spheres fall onto the pyramid. There is another period of frequent collisions as the larger sphere falls onto the pyramid. Thereafter, the spheres scatter as most are deflected away, and collisions become much less frequent. The prediction of linear time performance depended on the assumption of spatial coherence – i.e. that collisions are *infrequent*. So for this period of frequent collision at the beginning of the simulation, the performance will be closer to the quadratic than linear. For most of the simulation run, spatial coherence holds and the performance is linear. Moreover, though the total time is quadratic, it is quadratic with a very small constant factor (since the periods of frequent collision last for only a short time), so the scaling is still *close to* $O(n)$.

For the purposes of benchmarking, I chose scenes where collisions were relatively frequent, so as to stress the collision detection subsystem and to ensure a rigorous test of the real-time requirement. In many real-life scenes, spatial coherence is likely to hold more strongly, so the performance of the algorithm should be much closer to linear.

Another aspect in which the two collision detection algorithms differ is in their performance over time. The naïve algorithm simply checks all pairs every frame, so its runtime for each frame should be constant. Figure 4.16 confirms this. Baraff's algorithm, however, keeps track of what it has done previously, and if it has already determined two objects' AABBs are disjoint, it will not check them again unless the last sweep of insertion sort shows they may have moved together: in other words, it should adapt as the scene evolves. Figure 4.17 confirms the prediction that Baraff's runtime for the first frame should be much greater than for frames thereafter; however, it is otherwise rather uninformative, so Figure 4.18 shows the same data suitably rescaled, with the first frame omitted. The graph shows several features: a peak around frame 150 when the small spheres fall onto the pyramid, then another around 300 when the large sphere hits. As predicted, the simulation time gradually falls off as the spheres disperse and collisions become less frequent. It is noteworthy that even the peaks are still an order of magnitude smaller than the (constant) time for the naïve algorithm.

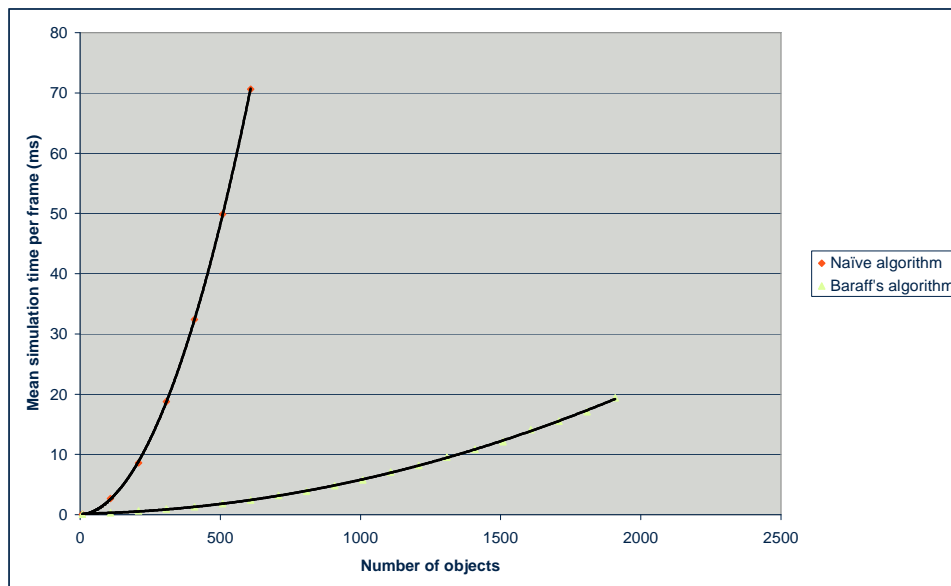


Figure 4.15: Graph of mean simulation time per frame against number of objects for the “spheres and pyramid” scene, comparing the naïve algorithm to Baraff’s.

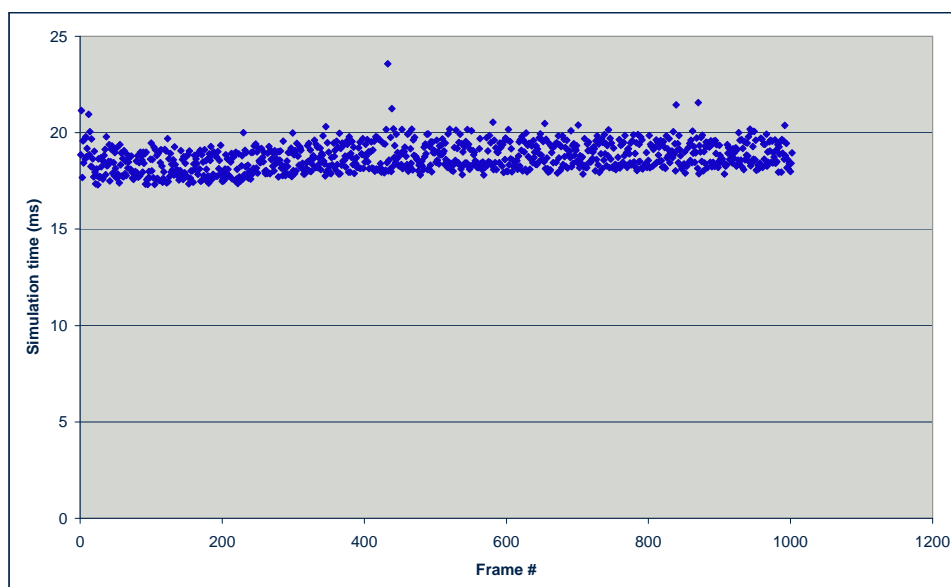


Figure 4.16: Trace of simulation time across a simulation run of the “spheres and pyramid” scene, using the naïve algorithm.

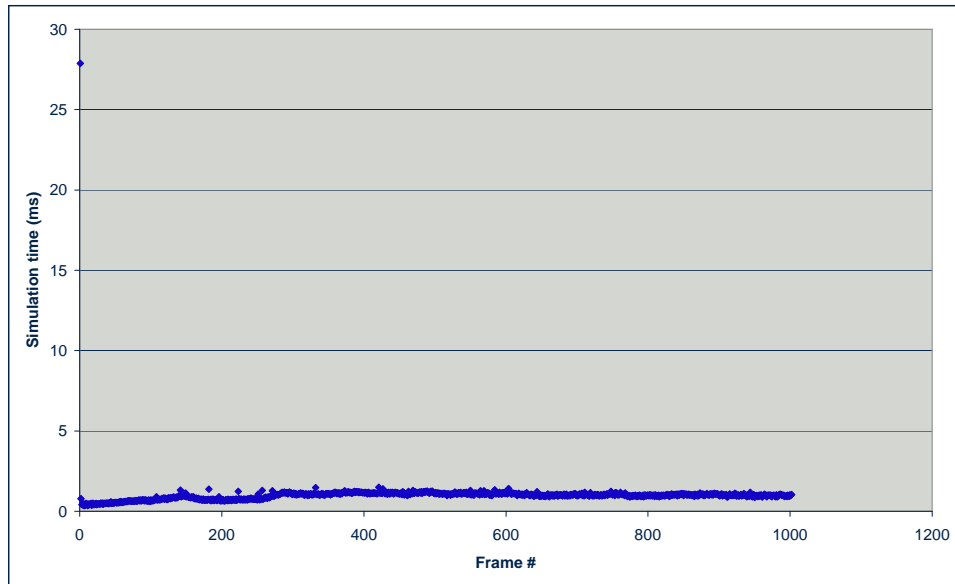


Figure 4.17: Trace of simulation time across a simulation run of the “spheres and pyramid” scene, using Baraff’s algorithm.

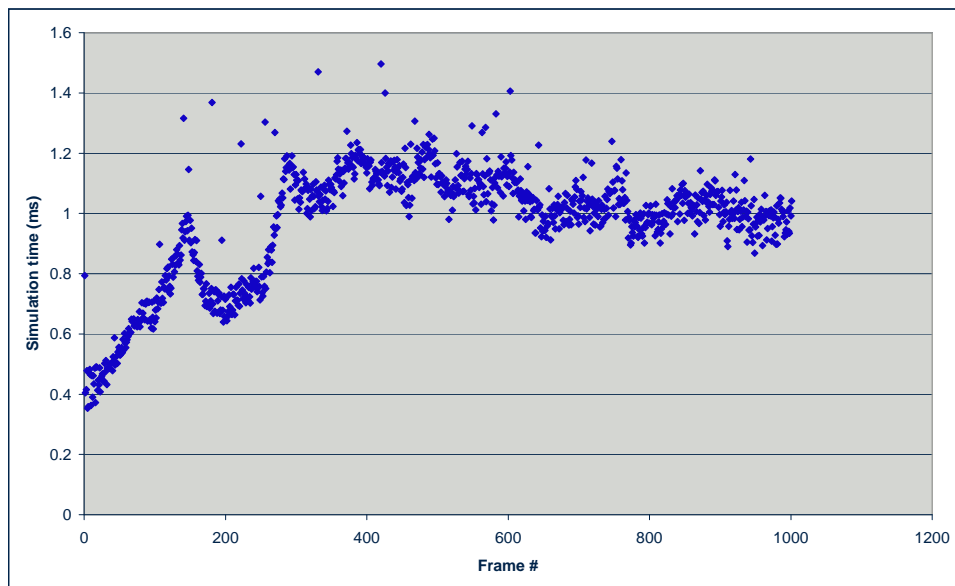


Figure 4.18: Trace of simulation time across a simulation run of the “spheres and pyramid” scene, using Baraff’s algorithm, omitting the first frame.

Chapter 5

Conclusions

As the previous chapter demonstrates, the project has achieved and surpassed its specified goals. I have learned a great deal about physical simulation and collision detection in the process of researching and implementing it; I have also gained valuable experience in using the various tools and languages involved, and in designing and building a medium-sized system.

Inevitably, some of what I learned in the process relates to how the project might be designed differently were I to start again from scratch. I would like to have been able to explore more aspects of collision detection (for example, there has been much interesting research into efficient algorithms for collision detection in highly complex scenes), and using third-party maths and graphics libraries might allow more time to investigate these.

The choice of explicit algebraic calculation of shape intersections may have been an unnecessarily limiting one. As well as rendering some intersection tests very difficult to design, it requires writing a different intersection test for each pair of primitive shapes, which is time-consuming. A more general approach is presented in [vdB04], where intersection queries are performed on any pair of convex shapes by the powerful and efficient Gilbert-Johnson-Keerthi algorithm, which starts from an initial ‘guess’ and uses a representation of the shapes’ geometries to converge iteratively on the solution.

Clearly, the issues to be addressed when implementing a simulation of rigid body motion are many and diverse, and most individually deserve as much time and attention as has been devoted to this entire project. To ensure the scope remained manageable, I have concentrated on building a complete, working system, and on achieving efficient collision detection. These aims have been achieved. My program produces visually impressive, physically convincing simulations of familiar scenes, scales to handle large numbers of objects, and does so in real time. I therefore conclude that the project was an enjoyable success.

Bibliography

- [AH04] D. Astle and K. Hawkins. *Beginning OpenGL Game Programming*. Thomson Course Technology, 2004.
- [Bar92] D. Baraff. Dynamic Simulation of Non-Penetrating Rigid Bodies. Ph.D Thesis, Cornell University, 1992
- [JTT00] P. Jiménez, F. Thomas and C. Torras. 3D Collision Detection: A Survey. In *Computers and Graphics*, Vol. 25, 2001.
- [LG98] M. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.
- [MC95] B. Mirtich and J. Canny. Impulse-Based Simulation of Rigid Bodies. In *Proc. 1995 symposium on Interactive 3D Graphics*, ACM Press, 1995.
- [Sho85] K. Shoemake. Animating rotation with quaternion curves. In *Proc. 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 1985.
- [Str00] B. Stroustrup. *The C++ Programming Language – Special Edition*. Addison-Wesley, 2000.
- [vdB04] G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, 2004.

Appendix A

Scene description language

This appendix describes the grammar of the scene description language (SDL). Note that the SDL is effectively specified at two semantic levels – the general structure of the language, and the specific allowed declaration types and required parameters. The former level will be described in the form of a context-free grammar; the latter will be described less formally.

The tokens output by the lexer are:

- NUM: an integer or decimal number, e.g. 10, -3.5.
- NAME: an alphanumeric string beginning with a letter, e.g. `sphere`, `Walrus42`.
- punctuation: the symbols `{`, `}`, `;`, `:`, `-`, `,` and `.`.

Whitespace is ignored, as are commented lines, denoted by `//` (as in C++) or `#` (as in Perl).

The grammar recognised by the parser is as follows.

```
exp      ::= NUM | -exp | exp, exp, exp
id       ::= NAME | NAME:      # the colon is optional
param    ::= id exp
params   ::= <empty> | params param
paramlist ::= ; | { params }
decl     ::= NAME paramlist
decllist ::= <empty> | decllist decl
scene    ::= decllist
```

The start symbol is `scene`.

The allowed declaration types are:

- `sphere`: creates a sphere. Parameters are:
 - `radius`: required, scalar, must be positive.

- **mass**: required, scalar, must be positive.
- **centre**: required, vector.
- **vel** (velocity): optional, vector, default $(0, 0, 0)$.
- **cylinder**: creates a cylinder. Parameters are:
 - **radius**: required, scalar, must be positive.
 - **height**: required, scalar, must be positive.
 - **mass**: required, scalar, must be positive.
 - **centre**: required, vector.
 - **axis**: required, vector, must be nonzero.
 - **vel** (velocity): optional, vector, default $(0, 0, 0)$.
 - **angvel** (angular velocity): optional, vector, default $(0, 0, 0)$.
- **wall**: creates an immobile, infinite plane. Parameters are:
 - **normal**: required, vector, must be nonzero.
 - **toOrig** (signed distance to the origin): required, scalar.
- In addition, all objects have the following two parameters:
 - **elastic**: optional, scalar, must not be negative, default 1.
 - **colour**: optional, vector, all components must be between 0 and 1 inclusive, default $(1, 1, 1)$ (white).

Appendix B

Sample code

This appendix presents a few illustrative samples of the project code.

B.1 Baraff's algorithm

The top-level collision detection algorithm:

```
typedef std::pair<base::Object*, base::Object*> objpair;
std::set<objpair> poss_collisions;
typedef std::set<objpair>::iterator pairsetit;
typedef std::set<objpair>::const_iterator pairsetcit;

// Detects and resolves collisions between the objects
// in the list. Uses Baraff's 'sweep and sort'
// algorithm, as presented as 'sweep and prune'
// in van den Bergen pp.210-5
int do_collisions(list<Object*>* objects) {
    // perform insertion sort on the endpoints for each of
    // the three axes in turn. tweak_possibles simply
    // adds or removes object pairs from the set of
    // possible collisions when passed a pair of AABBs.
    inssort(AABB::xendpts, xaxis, tweak_possibles);
    inssort(AABB::yendpts, yaxis, tweak_possibles);
    inssort(AABB::zendpts, zaxis, tweak_possibles);

    // poss_collisions should now contain all object pairs
    // whose AABBs intersect, i.e. all pairs of objects
    // which might be colliding.
    for (pairsetcit it = poss_collisions.begin();
         it != poss_collisions.end();
         ++it) {
```

```

Object* fst = it->first;
Object* snd = it->second;
if (fst != snd) {
    const IntersectInfo& info = intersect(fst, snd);
    if (info.intersect) {
        ncolls++;
        response(fst, snd, info);
    }
}
}
}
}

```

B.2 Quaternion class

The quaternion class declaration:

```

/** Quaternion class */
class Quat {
public:
    Quat(double a, double b, double c, double d)
        : w(a), v(b,c,d) {}
    Quat(double a, const Vector& s) : w(a), v(s) {}
    Quat(double a) : w(a), v(0, 0, 0) {}
    Quat(const Vector& s) : w(0.0), v(s) {}
    const Quat operator-() const;
    Quat& operator+=(const Quat& q);
    const Quat operator+(const Quat& q) const;
    Quat& operator-=(const Quat& q);
    const Quat operator-(const Quat& q) const;
    Quat& operator*=(const Quat& q);
    const Quat operator*(const Quat& q) const;
    friend const Quat operator*(double a, const Quat& q);
    friend const Quat operator*(const Quat& q, double a);
    Quat& operator/=(double a);
    const Quat operator/(double a) const;
    double sqnorm() const;
    double norm() const;
    /** multiplicative inverse */
    const Quat inv() const;

    double w;
    Vector v;
};

```

```
const Quat operator*(double a, const Quat& q);
const Quat operator*(const Quat& q, double a);
```

B.3 Object creation

The object creation function used when building the scene:

```
extern Factories* factories;

// attempts to build an object from a declaration with the
// given name and parameters.
Object* mkObj(string name,
              const pmap& paramfs,
              const p3map& param3fs) {
    factories->clearAll();
    if (factories->factories.size() < 1) {
        cerr << "Fatal error - no object declarations are "
              << "defined!\n";
        exit(1);
    }

    if (factories->factories.find(name) ==
        factories->factories.end()) {
        // invalid decl type
        throw UnknownDecl(name);
    }

    ObjectFactory* maker = factories->factories[name];
    maker->set(paramfs, param3fs);
    return maker->make();
}
```


Sam Stokes
Robinson College
ss496

Computer Science Tripos
Part II Project Proposal
Thursday 21st October 2004

An approximate graphical simulation of rigid body motion

Project originator Sam Stokes

Project supervisor Richard Southern

Director of Studies Alan Mycroft

Overseers Jean Bacon
John Daugman

Resources required

PWF facilities

My own PC (2GHz Athlon 64, 1GB RAM, 280GB hard disk, Windows XP Pro)

Introduction

The simulation of rigid body dynamics is a well-understood problem in computer science. For many applications – such as physical modelling for experimental purposes – it is necessary that the simulation give physically accurate results. This is usually achieved by means of finding numerical solutions of sophisticated constraint-based mathematical models; however, this approach is computationally intensive. When the target is to produce an animation of the motion convincing to a human observer, a simpler physics model can be used while still appearing sufficiently ‘realistic’. The primary challenges are then found in the areas of efficient collision detection and collision response.

Project description

I intend to develop a graphical simulator of approximate rigid body motion in three dimensions. The program will read in a scene description from a text file, render the scene, and evolve the objects within it according to the laws of dynamics, producing an animation of the motion. At least for simple scenes it should be possible for this animation to be displayed in real time.

The project will be structured as follows:

Scene description

Scenes will be specified and input in a text-based scene description language. This will be parsed by the program and converted into an internal representation suitable for modelling calculations and rendering.

Collision detection

A significant part of the simulation will be concerned with efficient collision detection (CD). I hope to be able to handle scenes containing a fairly large (20+) number of objects, so that it is possible to model scenes such as a ‘domino rally’ or a dry-stone wall being toppled. A naïve CD algorithm must check every object pairing for impending collision, which scales quadratically with the number of objects. While this is unavoidable in the worst case for any CD algorithm, techniques are known for reducing the average-case complexity using bounding volumes and techniques such as assumption of ballistic trajectories, which ensure that the algorithm will only check pairs of objects that are likely to collide. Recent survey articles in this field are [LG98] and [Jim01].

Dynamics modelling

Essentially this will consist of simulating the linear and rotational motion of various objects with time, and modelling interactions between them.

There are of course a large variety of physical effects and characteristics which one might desire to be modelled in a complete dynamical simulation: elastic and inelastic collisions, elastic forces, friction, gravity, deforming bodies, non-uniform density, objects of arbitrary shape, and so on. To ensure that the scope of the project remains manageable in the time available, I will focus on a restricted class of objects and of physical interactions. The allowed object shapes will be the traditional 3D graphics primitives – cuboids, spheres, cylinders, etc. – and bodies will be rigid and uniform;

interactions modelled will be (non-deforming) inelastic collisions and (simple) gravity. I hope to be able to extend this to include friction, rolling objects and spring forces.

The choice of approximate simulation is made for two reasons: it reduces the complexity of the implementation, and allows for better efficiency and scaling, while the effect produced is still perceived by observers as realistic. For example, although a complete physical model would include a gravitational force acting between every pair of objects, in most situations a human observer would not expect to perceive any noticeable motion due to gravity other than a constant acceleration toward the ground: the calculation of an inverse-square force for every object pairing can therefore be replaced with a simple constant downward acceleration, simplifying the dynamical calculations significantly. My simulation will use the method of calculating the instantaneous forces on an object at discrete time intervals, and applying these to change the object's velocity, and likewise applying the velocity to modify its position: this should make the calculations per object relatively simple, allowing work to be concentrated on the implementation of the simulation itself.

Success criterion

The project will be deemed successful if it is capable of producing an animation of the dynamical evolution of some simple scenes in which various rigid, uniform objects collide, to a degree of accuracy that appears realistic to a human observer.

Languages and tools

I will develop the simulator in C++: this language's object-oriented features will be valuable in designing and implementing the project in a maintainable fashion, and its ability to produce high-performance code will be important since efficiency is a goal of the project. For the 3D display I will use the OpenGL libraries, which are cross-platform, optimised for efficient display and animation of 3D scenes, and well understood in the graphics community. C++ also has a robust API for interacting with OpenGL.

Starting point

I have acquainted myself with the state of knowledge in the field of collision detection (in particular from [LG98] and [Jim01]) and have read some material on rigid-body dynamics simulation (especially [MC95]). I am familiar but not experienced with C++ and OpenGL.

Resources

This project will require no special resources other than the use of my own machine, so as to be able to continue work outside the computer lab and during the holidays. I shall employ several redundant backup strategies to prevent data loss due to hardware failure or accident: I will use a version control system to track all changes and backup regularly to Pelican, and also to CDR in case of network failure.

Plan of work

The following is the proposed timetable for the project.

1. 22/10/04 – 04/11/04

Prepare backup strategies for the project: e.g. set up repository for version control system, backup scripts.

Become more familiar with C++. Write some demonstration programs to aid and assess my progress.

2. 05/11/04 – 18/11/04

Become familiar with OpenGL. Write some demonstration programs to aid and assess my progress.

Research into efficient collision detection algorithms.

3. 19/11/04 – 11/12/04

Define the scene description language. Write a parser that reads in a scene description file and converts it to an appropriate internal representation. Create a simple OpenGL renderer for the scene.

Milestone: should be able to read in some simple scenes and display their initial setup in 3D.

4. 12/12/04 – 14/01/05 – Christmas vacation

Implement simple object motion (ignoring collisions) under gravity. Develop the OpenGL animation for objects in motion.

Implement naïve collision detection algorithm and physics model, and test this on some simple scenes.

5. 17/01/05 – 03/02/05

Select a more efficient CD algorithm, and begin implementation.

Write progress report.

Milestone: should be able to simulate and display the behaviour of some simple scenes with colliding objects.

Deliverable: Progress report – 4 February 2005

6. 04/02/05 – 17/02/05

Prepare progress report presentation.

Deliverable: Progress report presentation – 10-15 February 2005

Complete implementation of efficient CD algorithm. Test on some simple scenes.

7. 18/02/05 – 03/03/05

Compare the naïve and efficient CD algorithms and produce test report to be included in dissertation.

Create some more complex scenes to test CD (e.g. domino rally): may require writing scripts to automatically generate the scene description files.

8. 04/03/05 – 18/03/05

Investigate extensions such as spring forces and rolling motion. Attempt to implement.

Begin work on dissertation: plan structure and write introduction and/or abstract.

9. 19/03/05 – 22/04/05 – Easter vacation

Continue work on extensions.

Test the core project on a variety of scenes.

Write dissertation.

Begin revision for final exams.

10. 23/04/05 – 05/05/05

Finish work on extensions, and evaluate their success.

Update dissertation to reflect extension work.

Revision will take priority during this period.

11. 06/05/05 – 19/05/05 – Final touches

Final bug-fixing of project, putting final touches to the dissertation. As this period is only a few weeks before the Tripos examinations, no major changes to the project or dissertation will take place at this point.

Deliverable: Project dissertation – 20 May 2005

References

1. *Collision detection between geometric models: A survey* [Lin and Gottschalk 1998]
2. *3D Collision Detection: A Survey* [Jimenez et al. 2001]
3. *Impulse-based simulation of rigid bodies* [Mirtich and Canny 1995]