A FRAMEWORK FOR PHYSICALLY BASED MODELLING IN VIRTUAL ENVIRONMENTS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

March 2002

By

Mashhuda Glencross

Department of Computer Science

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

A framework for physically based modelling in virtual reality (VR) is introduced in this thesis. Currently many VR walk through demonstrations exist and the majority of these are static, containing limited interaction with objects. Simulating virtual world behaviour using real world physics is a worthy cause as it formalises a method for specifying consistent and plausible motion of objects. The problem with physically based models is that in general they are computationally intensive.

However, advances in the performance of desktop computers are making VR widely attainable. Modern desktop computer processors are faster than those present in graphics workstations used for research only three to five years ago, thus realising the prospect of physically based modelling in VR. This is not a trivial matter as most of the processor's power still needs to be devoted to rendering, so simulation computations must only use a small percentage.

In this thesis it is argued that the proposed framework provides a flexible and unified interface for physically based modelling in VR and addresses issues related to user interaction with such models. To this end the framework contains a general purpose simulator capable of dynamically restructuring scene graphs to support a mechanism for users to manipulate models, and a VR kernel. Both of these are under the control of a high level scripting language. The relative merits of this approach are demonstrated through a variety of case studies.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

This thesis would not have been possible without the help of numerous people, and it is not possible to list them all but I am grateful to everyone who commented or contributed in any way. I would especially like the pleasure to acknowledge the following people:

My supervisors Dr. Alan Murta and Dr. Steve Pettifer have been a constant source of advice, support and encouragement over the years. I owe much to my discussions with friends who provided insight over many a lunch hour and would especially like to thank: Gary Ng, Athina Christodoulou, Nadia Papamichail, Greg Wright, Daniel Kidger, Donal Fellows and James Marsh. I must also thank my MSc conversion course students who gave me faith in my ability to teach, and lots of laughs in the tea room. Special thanks also to Chris Kirkham and my husband Nicholas Glencross for reading drafts, their comments, discussions and support. In my life there are two very special people, my parents, who have always been there to bring me back to reality and so I dedicate this thesis to them.

Last but not least, I am indebted to the Department of Computer Science at the University of Manchester for providing me with a teaching post to finance my studies, and giving me real insight into the wonderful world of academia. I have gained much from the experience. This thesis was formatted using the LaTeX document preparation system, and the use of the wonderful GNU products is also acknowledged.

Chapter 1

Introduction

This thesis describes a proposed framework for physically based modelling in virtual reality (VR). In order to model and simulate flexible or rigid objects with which a user can interact, a number of core concepts from physics need to be implemented, often specifically for a given application. The aims of this thesis are to identify the types of physically based models required by examining literature within the subject of animation in conjunction with the types of models currently implemented in VR; to propose a novel framework for general physically based modelling in VR, and finally to implement a prototype middleware system which adheres to this framework as a proof of concept.

In our implementation there is an emphasis on simulations in which physical models can be dynamically restructured at runtime such that some of their physical characteristics, for example mass and shape, are altered. This approach has benefits in two main areas; firstly it provides a paradigm for interaction with physical models by specifying a clear interface through which a model's structure can be modified, and secondly a means for complexity management of simulations by restructuring models such that the amount of complex computations required to simulate them is minimised. Events which induce such changes can be invoked interactively by the user or choreographed in a program. It is attractive to add

complex behaviour to virtual environments, since it provides a richness in the types of user interaction possible with a computer generated model.

To set the scene, we present a clear and motivated statement of the problem to be solved. This is followed by a discussion of computer animation, virtual reality and physically based modelling in VR to set the context of this work within the subject of computer graphics. Subsequently, the goals of the research, a proposed solution to the problem, and the novel contributions made to knowledge are described. Finally a brief overview of the structure of this thesis and a summary conclude this chapter.

1.1 Problem statement

Virtual environments (VE), that is computer generated three dimensional worlds, tend to be static environments and there can often be few stimuli capable of keeping users interest for significant periods of time. Many VR applications, systems that create a real-time visual, audio and possibly haptic experience [Vin98], may present users with the opportunity to walk through an environment [MGH+98], pick up and put down objects, and possibly open and close doors. Often this type of interaction is event driven, the user may click a mouse button which invokes a script to animate the action of opening a door. Limited direct manipulation of such models may also be allowed often taking the form of rotating or moving an object. Simplistic interaction paradigms of this kind restrict the type of user participation possible because the VR system is activating the behaviour rather than it being an integral part of the world. In general, a user cannot participate in an interaction with complex entities within the world, such as a system of pulleys, ropes or manipulate deformable materials.

The current state of the art systems enabling physically based modelling

in VR are typically developed from animation systems, and much research effort has been expended in improving the performance of physical simulator engines [Fau99, KSZB95, KSB]. For example, rigid or flexible body simulation techniques may be implemented for specific applications such as VR surgery [HA95, Wat99], military simulation [ZPF+93] or entertainment [Col98, Woo99].

A survey of the literature and software libraries available for physically based modelling in VR reveals there is a lack of middleware systems which provide enough core functionality for applications developers to build suitable systems. In particular many toolkits such as MR [MRT] and MEME [MEM] only enable physical behaviour to be attached to objects via small self contained scripts. Physical laws are not an integral part of the environment.

At a high level, there are two issues which can be considered to be required to support physically based modelling in VR. Firstly, a simulator technology capable of real time computation. Secondly support for the key functionality necessary for VR such as navigation, input/output to and from VR peripherals and managing the complexity of large models. Traditionally these two areas are considered independently of each other due to the different emphasis of the subject domains and each area has its own unique set of problems which still need to be solved.

This thesis addresses the issue of supporting both simulator and VR services through a single application programming interface which is both extensible and customisable. Furthermore the issues of user interaction with physical simulations and management of complexity of simulations are also addressed.

To provide the reader with a background of the subject areas this objective covers we examine the history and use of physically based modelling in animation in the following section. Context and motivation for this type of modelling is described with its relevance to the field of animation. The discussion then leads on to a brief history of VR since this is the subject domain within which this research

is placed. An impression of some of the problems that have to be considered in order to adapt animation research for VR is imparted. Finally an overview of the state of the art in physically based modelling in VR is presented in order to convey the problems associated with current approaches.

1.2 Animation techniques

The fantastic animated films for which Walt Disney is famous, still charm adults and children world-wide. Disney's first full length animated feature film was "Snow White and the Seven Dwarfs" [Dis37] and proved to be more successful than he himself had ever imagined. Originally, it was intended to be a one-off, but it was to become one of many spectacular animated films. However, making Snow White come alive involved much painstaking work. Firstly, the images deemed to be the most important key frames were drawn by skilled artists. Then artists called "inbetweeners" drew the images between the key-frames to complete the animation sequences, and finally, all these frames were combined to produce the film. It would clearly have been more convenient to automate some of this effort and animate Snow White by computer.

However, computer animation is by no means trivial. For example Lucasfilm's "The Adventures of Andre and Wally B", had over 700 model parameters that were hand animated via key-frame interpolation [Bar92b]. More recently, the first fully computer animated feature film "Toy Story" took four years [LD95] to complete, and it required over ten thousand lines of code to animate the lead character Woody. The characters were represented as three dimensional models within the world and the animators were able to achieve shading and lighting effects that could not have been achieved using traditional animation techniques.

Early computer animation systems attempted to automate inbetweening using splines [WW92]. Skilled animators were still required to produce the key frames,

but now they required additional skills to be able to specify animation sequences, usually in some scripting language. This was not a desirable situation as it is unintuitive and difficult to animate anything remotely complex. Consequently much of the current research in animation has focused on tools which enable the animator to interactively specify at a high level the path of a particular motion sequence and more importantly the rules which govern it. Thereby the animator is relieved of the burden of dictating the detailed motion.

It is useful to compute the motion of an object using physical rules as the animator then has the freedom to try various sequences, none of which will violate fundamental physical behaviour. For example, in a traditional animation system a sequence of a tennis ball being thrown to the ground and bouncing back required a significant amount of data to be specified explicitly and carefully by the animator. However, using physical rules only a few parameters need specifying from which the path for the sequence can be computed and later used by the computer program responsible for generating the final images. Such physically based motion generation used to lie purely within the domain of simulation, but is now widely used in animation.

Many researchers have investigated interactive, physically based simulation for animation purposes [Gas92, Ove91, Ove94, WW90], and many of these approaches have been successful in the limited context of rigid body animation. Such techniques have been developed to help the animator intuitively produce animation sequences for use in the entertainment industry. They are responsive because animators want to see the results of changes to scenes instantaneously. These particular animation algorithms have similar frame rate performance requirements to VR systems but they do not necessarily have to be as robust or as detailed. An animator has the luxury of being able to regenerate frames which do not appear to be correct because the objective is to produce only one smooth

recording.

Aside from the consistency and interactivity concepts described earlier, VR systems differ from interactive animation in that they have differing end requirements. Animation systems are used to produce one recording of a given choreographed motion sequence which may be rendered in a more pleasing fashion later. Contrast this with VR systems which are real time interactive environments in which the user can have a profound effect on a simulation while a minimum level of graphical detail is maintained. This means that a general purpose physically based VR system has to be more robust than an interactive animation system. Bearing this point in mind, ideas developed for physically based interactive animation can still be adapted for use in VEs. To clarify the motivation and introduce context for the subject of physically based modelling in VR we outline in the following section the history and development of the subject.

1.3 The advent of virtual reality

The term virtual reality (also sometimes known as alternate realities, immersive environments or synthetic environments) is used to describe an interactive, three dimensional computer generated environment in which the user can become psychologically immersed, leading to a sense of involvement (*presence*). The definition of immersive experiences encompasses a wide range of human experiences from dreaming to being completely absorbed in a game like Tetris [GPP86]. Within the context of these experiences the sense of virtual presence may exist in varying degrees. While the factors that contribute to immersive experiences, in a VR context, are still not clearly understood, consistent behaviour is known to be important. Humans learn from observation [Ban86], and in doing so, build up a subconscious model of how the world around them works and how they interact with objects in it.

Research in the subject of VR can be divided into two distinct categories; firstly development of hardware and secondly software engineering. In the area of VR hardware much of the research has been directed by the work of Ivan E. Sutherland. In his quest for the sensation of being *immersed* in a VE Sutherland built the first Head Mounted Display (HMD) in the 1960s, using miniature cathode ray tubes as displays and mechanical links to relay the position and orientation of the head [Sut65]. Someone wearing this display would have found it to be very heavy and cumbersome. The Sutherland HMD started a boom of investigations into this technology and his work may have directed research into VR displays for thirty years. Although current HMDs are technologically far more sophisticated, the principle is the same as the original Sutherland version.

On the other hand in the field of VR software engineering, development has been comparatively slow and this opinion is supported by researchers such as Zyda et al. [ZPF+93] who propose a set of core requirements for VR software. In many current applications users can affect a VE by manipulating objects to achieve some simple objective, for example, fitting together two pieces of "virtual Lego™" [Guv99]. It is often difficult to predict what the user might choose to do to an object, so in general limited forms of manipulation are enforced. For more general modes of interaction with complex objects a physically based model may be required. Since these types of models are generally computationally expensive, few have yet been implemented in VR systems. Relevant VR systems are described later in §3.2.2.

Complex manipulation of objects can contribute significantly to the realism of a VE. However, very few systems exist which enable a user to pull, deform or interact with objects in such a way that some of their physical characteristics are temporarily or permanently altered. Greatly enhancing the level of interaction possible in VR can have a significant impact on the sense of presence that a user

may experience while inhabiting a virtual world.

To simulate real world behaviour requires a model representing the rules governing an object's reaction to manipulation. For example if we drop a cup onto the floor we might expect it to smash. How do we simulate the number of pieces the cup breaks into? How does one work out the time taken for the cup to travel the distance from where it was released to the floor? How should the impact of the cup with the floor be identified, and characterised? Will the cup bounce when it hits the floor? Is the floor hard or soft? The list of questions that can be asked about a simple everyday situation, indicates just how complex the problem of describing behaviour of virtual objects can be.

An intuitive method for modelling the behaviour of virtual counterparts to real world objects is to simulate the laws of real world physics. This opinion is supported in a survey by Logan et al. [LWA94]. Most observable real world behaviour can be explained using *Newtonian physics*, the branch of physics describing the mechanics of observable motion. It stands to reason that the same physics based models could be used to compute behaviour of virtual counterparts. Unfortunately the situation is not so simple, and often there still has to be a tradeoff between performance and the fidelity of a simulation.

The value of fast consistent modelling in a VR system is timely, especially with the significant developments in computer hardware. More powerful computers are being made available to the home market so it appears likely that the future will bring VR simulations in which people act and affect the outcome, in their own homes for work and leisure. Computationally intensive physical models can now realistically be used in VR applications, though many of these models are still too complex for simulations to be carried out on desktop computers. In the following section a number of current approaches are described together with their limitations.

1.4 Physically based modelling in VR

The core mathematics for physically based modelling has been around since Isaac Newton published his laws of motion [New86] and his ideas have been exploited in simulation, engineering and visualisation. An explosion of interest in the field of computer animation occurred in late 1980s. Traditionally this type of computation has been carried out as batch jobs on supercomputers due to the complexity of the computations involved.

In recent years however, physically based modelling in VR has gained momentum and a number or techniques have been successfully used in applications ranging from VR surgery and crash simulations to entertainment. Currently many physically based models are only able to compute either rigid object motion or deformation of soft objects. Ideally, users would expect to find both these types of behaviours but the few VR systems that have evolved reflect this disparity of models so material behaviours are simulated only in limited contexts. For instance, some researchers have tried to solve the problem of rigid body simulation in VEs [TJ, KSZB95, KSB] and others have focused on the problem of flexible body simulation [LWA94, HA95, Nab]. A problem with many of these applications is that they require implementation of core physics and simulator technologies from scratch.

A number of solutions have been put forward in recent years. Chapman and Wills present a unified approach to physically based modelling in VR [CW97, CW98]. A range of performance critical simulator libraries is also currently available or under development such as MathEngine [Woo99] and Tabule [Fau99] but in general these are implemented from an animation or physics perspective rather than a VR standpoint. As such the focus of research has largely rested on improving the performance of simulators which use techniques commonly found in animation [Fau99, Gas92, Ove91, Ove94]. The majority of these techniques are

not completely robust, meaning that the simulator may occasionally be unable to compute motion for the specified conditions. Furthermore the MathEngine library and Tabule currently have little VR support as they are essentially simulator libraries.

Consequently, the differences between interactive animation and simulation in VR have received little attention. VR differs from interactive animation in two very important and highly coupled ways: consistency and interactivity. By consistency we mean both a consistent world model and a consistent frame rate. Interactivity refers to both performance (frame rate) and complex user interaction with the model. These concepts are tightly bound because the provision of a consistent world model relates to the ability to provide complex interaction, while performance and consistent frame rate are also related. The danger of certain frames taking longer to compute than others is especially relevant in VR simulation. Hence, a simulator implemented for VR has to be able to compute results consistently quickly and be robust. This is a difficult task because of the very nature of physically based modelling techniques. Therefore a solid motivation exists for reducing the complexity of simulations to make the problem of computing complex physical behaviour more manageable for VR.

Moreover, few general purpose languages for complex scene description exist in current VR libraries or toolkits. Applications that use physical models in VR, due to their restricted scope, tend to use hard-coded underlying models which lack generality. Many programs are written to demonstrate a particular idea and thus are written at a relatively low level. Consequently, the programmer usually has to have a solid background in physics in order to implement a simulation usually in a language such as C or C++. A VR system for physically based modelling ideally requires a flexible and powerful means of describing a broad class of simulations for a wide variety of applications. Now that the context of the work described in

this thesis has been established as physically based modelling in VR, it is possible to give an overview of the nature of the proposed solution.

1.5 Proposed solution

In this thesis we advocate the use of a middleware framework of components for the development of applications which incorporate physically based modelling in VR. For now it is best to think of components as isolated "black box" parts of a software system responsible for providing a given functionality through a defined interface. Relevant concepts and motivation for this choice is detailed in Chapter three but briefly the philosophy of building a large software system by combining independent parts has many benefits. In particular, this type of framework is especially extensible, customisable and robust; this statement will be elaborated upon later in this thesis.

Furthermore management of the complexity of simulations is advocated in order to maintain robustness and real time interactive performance of the simulator engine. Consequently, the fidelity of simulations may be compromised in favour of performance. For this reason the aim is limited to computation of plausible motion only, and this is defined as being that motion which the average user will believe to be sufficiently accurate for a simulation to be judged correct. However, if the computational power is available, simulations may be computed to greater fidelity.

The specific implementation described in this thesis to illustrate the framework was developed using a high level scripting language (Perl) and the systems programming language C++. However there is nothing inherent in our approach which obliges us to use these particular languages, and any suitable alternatives with matching functionality would suffice. Moreover the implementation is a prototype developed as a proof of concept rather than a full product, and as such it

is not suitable for release to developers in general.

This thesis makes contributions in two main areas; management of complexity of simulations and a proposed architecture for a component based approach to physically based modelling in VR. More specifically, novel contributions include:

- First, a framework for physically based modelling in VR is proposed as a result of investigating the literature in the subject from which the core functionality required was identified. This framework is novel because of the architecture adopted and the combination of functionality supported through a single full and complete development language.

- Second, a method for dynamic restructuring of scenes supported by the simulator engine implemented for the framework. Dynamic restructuring is novel because of the way in which the scene graph is modified and analysed on the fly enabling the problem to be reconstructed for the simulator engine to solve. This approach provides a means for managing the complexity of simulations and a philosophy for user interaction and manipulation of physical models.

- Third, the concept of articulate domains is introduced, which relates to managing the complexity of physical simulations. This partitions the model into independent sub-problems which can be solved in isolation.

- Fourth, the framework's flexibility supports ad-hoc modelling within a physical context. The developer is not restricted by the need to conduct entire simulations to a specific fidelity or level of detail. This is novel because the simulator model is treated as an abstract representation of the virtual object and so can be as simple or as complex as desired, and this representation may vary as the simulation proceeds or as a result of user interaction. This means that a simulation can be as simple or as simulator intensive as

desired. The framework requires some conventions to be followed by the developer but are not intended to be restrictive.

- Finally, the use of general purpose physically based modelling in a novel context within a VE, which adds a uniqueness to the immersive experience due to the wide range of systems which can potentially be simulated and manipulated.

Now that these contributions have been described it is appropriate to address the question of the scope of the work described in this thesis. The scope of the framework is physically based modelling of articulated rigid body and particle based models in VR. The simulator implemented for use in the prototype framework may be replaced as long as the proviso of using *particle dynamics* (models based on point masses which are described in significant detail in §2.6 p. 49)is not violated. Potentially any appropriate simulator techniques or VR libraries can be incorporated into the framework.

The scope of the simulator engine implementation on the other hand has been limited to plausible motion computation. For this reason it can be argued that the simulator lends itself more to entertainment applications development and the case studies shown in Chapter six are chosen from within this domain. However, this is a consequence of implementation of specific case studies from the animation industry for comparison rather than a specific bias inherent in the simulator. Many serious applications can benefit from plausible motion computation as long as the simulation is used to provide a visual impression of realistic behaviour rather than detailed analysis of a particular phenomenon. For example, the realism of a VR walk-through of a large offshore installation can significantly benefit from simulation of ropes, pulleys or helicopters landing and taking off from helipads.

1.6 Thesis structure

Each chapter contains an introduction and summary, and an overview of the topic covered in each is listed here.

Chapter 2 discusses some fundamental concepts which the reader is invited to study before proceeding onto following chapters.

Chapter 3 discusses concepts relating to component frameworks and programming language choices to motivate the approach adopted for the prototype implementation of the framework (called Iota).

Chapter 4 describes implementation of the components which collectively form the Iota framework.

Chapter 5 illustrates the power and flexibility of the simulator and shows how it has been used in a number of different VR examples.

Chapter 6 presents suggestions for further work and concludes the thesis.

Appendix A contains miscellaneous algorithms developed during the course of the research described in this thesis.

Appendix B provides a background to the fundamentals of C and Perl callbacks.

Appendix C contains derivations for fundamental equations, together with a tutorial on the Newton-Raphson method for solving non-linear simultaneous equations.

Appendix D provides details on how C and C++ can be integrated with Perl.

Appendix E shows some simple examples of user scripts.

Appendix F contains comments regarding each case study from a sample of fifteen volunteers.

1.7 Summary

In this chapter we have presented the topics of animation, VR and physically based modelling in VR to set the context and illustrate the foundations for the ideas presented in this thesis. We show that there is a need to provide complex behaviour in VR to make environments more interesting and realistic. Animation systems use physically based models but their application in VR has been limited due to the performance demands imposed in such systems. Some of these approaches could be adapted for use in VR as long as the research is directed by VR requirements from the start and related issues of consistency, interactivity and generality are properly considered. Current VR systems tend to execute self contained scripts to activate specific event driven behaviour in a narrow context. This paradigm does not allow for much complex interaction with models. We suggest an alternative where the simulation environment drives the VR and the physical laws are an integral part of the application.

Furthermore we have indicated a need for a flexible framework within which to implement applications involving physically based simulations in VR. The components of this framework and motivation for them will be described during the course of this thesis.

Chapter 2

Physically based modelling

This chapter presents some of the laws, theorems and concepts which are relevant to forthcoming chapters. An elementary knowledge of terms and quantities is assumed. If the reader is familiar with basic Newtonian physics then it is possible to proceed directly to §2.5 where computer simulation of motion is discussed. Newtonian physics is the basis for all real-world behaviour, leaving aside situations such as relativistic or quantum level behaviour. For the purposes of simulation in computer graphics these extremes are rarely approached.

There are many types of models used in computer graphics, their nature depending on the application domain: some are accurate descriptions of real textbook physics, while other models are very loosely based on mathematical or physical models. This diversity of models encountered in computer graphics exists because realistic models are often too computationally intensive for interactive systems. Therefore, researchers often have to be inventive and develop models which produce plausible rather than physically accurate behaviour. For example, computational fluid dynamics models used to visualise fluid flow have to be accurate since they are often used in analysis and design of machinery. On the other hand, a distributed behavioural model for simulating flocking in animals [Rey87] only has to produce an overall impression that is visually appealing

and believable.

Rendering models also range from being highly physically based to being relatively ad-hoc. An example of a physically based atmospheric mirage model is presented by Glencross [GH97], where the emphasis is on realistic simulation of mirage phenomena. At the other end of the spectrum, a completely ad-hoc model to superficially render a rainbow is presented by Musgrave [Mus89].

The term "physically based modelling" has been used in computer graphics to represent any model which has an underlying mathematical model derived from physics. There is an inherent appeal in using established physical models within computer graphics because they are well understood and many have been successfully used in animation and simulation.

2.1 Newton's laws of motion

Newton's laws of motion and gravitation are the most important and fundamental laws in the subjects of mechanics and classical physics [New86]. It is these laws, proposed over 300 years ago, that have led to an understanding of the physical world without which developments such as space travel would never have been a possibility [Gri87].

Whilst Newton's ideas have remained the foundation of classical physics, they were later extended to encompass near-light speed travel and atomic-level behaviour by Einstein and others, bringing us towards modern-day physics. We do not concern ourselves with these extreme cases as they are beyond the scope of our work because the real world is observably Newtonian, so this is sufficient for simulating the behaviour of everyday objects in a virtual world. However other researchers have considered a variety of different approaches to computing motion; which are significantly different but each still based on the same core physics. For example recent work at NEC in Japan [Col98] used Newton's equations to

control the motion of molecules' nuclei and supplemented this with Schrödinger's equations to understand the motion of electrons as a reaction progresses whilst temperature and pressure vary.

The level of scientific growth which stemmed directly from Newton's work is nevertheless still very impressive. His view of the physical world is invaluable in construction engineering for analysing systems of forces on bodies. Buildings have to be designed with proper consideration of the forces they will have to withstand otherwise the consequences can be very severe. The classic construction engineering disaster at Tacoma Narrows, where the wind caused a bridge to oscillate at its resonant frequency, serves as a potent reminder to engineers of the value of analysing forces on materials used in construction. Newton's three laws of motion may be stated as follows [SB90]:

I. A particle remains at rest or continues to move in a straight line at constant velocity if there is no unbalanced force acting on it.

II. The rate of change of the momentum of a particle acted upon by an unbalanced force is proportional to the magnitude of the force and is in the direction of the force.

III. To the action of every force there is an equal and opposite (force) reaction.

These laws are discussed further in the following sections. There are two distinct types of motion: linear and rotational motion and they are presented separately for various configurations of point masses, otherwise known as particles. In the real world these two types of motion are rarely found in isolation as moving objects are usually subject to some combination of both. However in order to understand them it is simpler to consider each in isolation.

2.2 Action of force on a particle

When a particle at rest is subjected to an unbalanced force it begins to move in the direction in which it is compelled to move, due to the force being exerted on it. If the force is exerted by an attractor then it has the action of pulling the particle towards it. If the particle is subject to a repelling force then it will be pushed away from the repellor.

2.2.1 Linear motion

Linear motion results from an unbalanced force acting on an object. This motion of a particle is simple to calculate from Newton's laws of motion. Newton's Law II may be stated as

$$\mathbf{f} = k\frac{d(m\mathbf{v})}{dt} \tag{2.1}$$

where $\mathbf{f}$ is the unbalanced force vector on the particle

m is the mass of the particle

$\mathbf{v}$ is the absolute velocity vector of the particle

$m\mathbf{v}$ is the momentum vector of the particle

k is a factor of proportionality

If m is constant, the law becomes

$$\mathbf{f} = km\frac{d\mathbf{v}}{dt} = km\mathbf{a} \tag{2.2}$$

where $\mathbf{a}$ is the absolute acceleration of the particle. A suitable choice of units, for instance kg and ms^{-2}, enables k to become unity so that

$$\mathbf{f} = m\mathbf{a} \tag{2.3}$$

The rate of change of a particle's position vector, $\mathbf{p}$, with respect to a point is defined to be its absolute velocity. The absolute velocity $\mathbf{v}$ of a particle can thus be written as

$$\mathbf{v} = \frac{d\mathbf{p}}{dt} = \dot{\mathbf{p}} \tag{2.4}$$

In turn, the rate of change of a particle's velocity is defined to be the particle's acceleration.

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} = \dot{\mathbf{v}} = \frac{d^2\mathbf{p}}{dt^2} = \ddot{\mathbf{p}} \tag{2.5}$$

Since Equation 2.3 gives the instantaneous acceleration of a particle, it is necessary to integrate Equation 2.5 to obtain the path of a particle.

$$\mathbf{p} = \int \mathbf{v}\, dt + \mathbf{p}' = \iint \mathbf{a}\, dt^2 + \mathbf{p}'' \tag{2.6}$$

where $\mathbf{p}'$ and $\mathbf{p}''$ are arbitrary constants derived from the boundary condition.

2.2.2 Rotational motion

Rotation may also result from applying forces to objects. These rotations result from a 'turning effect', or more specifically, a *moment*. Moments are calculated by taking the force's perpendicular distance from the point about which the rotation will occur, and multiplying it by the magnitude of force.

A classic textbook example for testing this observation is a door. Where the perpendicular distance is small, such as near the hinge, the door becomes much more difficult to open and close, and finally at the hinge itself, the door cannot be moved at all. Similarly, the perpendicular distance can also be changed by varying the angle of the force, making it very difficult to influence the door if all

of the push is towards the hinge.

The moment about a particle can be illustrated if the particle can be affected by a force applied some distance away from it. Hence, we represent a particle subject to a rotation as a sphere with a rigid rod to show where the force actually acts to cause a rotation as shown in Figure 2.1.

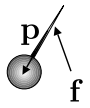


Figure 2.1: Rotation of a particle due to an offset force

The moment can now be calculated as described above, or using *the vector cross product*. The moment $\mathbf{e}$ is given by the vector product (denoted here as $\wedge$) of the force vector ($\mathbf{f}$) and the vector from the force to the turning point ($\mathbf{p}$)

$$\mathbf{e} = \mathbf{p} \wedge \mathbf{f} \tag{2.7}$$

The resulting moment $\mathbf{e}$ will have the following properties:

- It is perpendicular to both $\mathbf{f}$ and $\mathbf{p}$.
- Its magnitude is $\mathbf{pf}sin\theta$, where θ is the angle between $\mathbf{f}$ and $\mathbf{p}$.
- The action of the force will be to cause a rotation about $\mathbf{e}$ (and similarly $\hat{\mathbf{e}}$, its unit vector equivalent).

In practice, a moment cannot be applied to a single particle in isolation, since a force cannot readily be applied at a point some distance away from it. Situations where these principles can be applied will be presented in later sections.

2.3 Action of forces on collections of particles

The action of forces on a single particle can be extended to collections of particles. Forces which do not exert between particles are known as *external forces*, as in the case of wind flows or currents. This can be contrasted with forces between particles that interact with each other, for example mutual gravitational or electrostatic forces which are referred to as *internal forces*.

2.3.1 Linear motion

Consider a system to merely consist of a collection of particles which may be interacting or otherwise. The linear motion of such a system can be derived as follows [Lew71]. Figure 2.2 shows two interacting particles $\mathbf{p}_1$ and $\mathbf{p}_2$ (at some distance from an arbitrary origin O) which experience internal force vectors $\mathbf{f}_1'$ and $\mathbf{f}_2'$ and external forces denoted by $\mathbf{f}_1$ and $\mathbf{f}_2$.

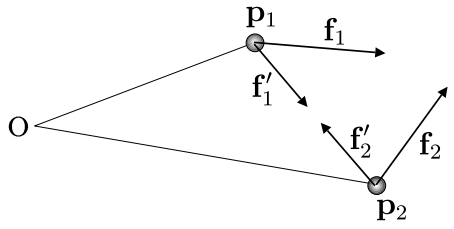


Figure 2.2: External and internal forces applied to two particles

Newton's second law applied to each particle gives

$$\mathbf{f}_1 + \mathbf{f}_1' = m_1 \ddot{\mathbf{p}}_1 \tag{2.8a}$$

$$\mathbf{f}_2 + \mathbf{f}_2' = m_2 \ddot{\mathbf{p}}_2 \tag{2.8b}$$

Since the total interaction $\mathbf{f}_1' + \mathbf{f}_2'$ is zero by Newton's third law (and so $\mathbf{f}_1' = -\mathbf{f}_2'$)

the two formulae may be added giving

$$\mathbf{f}_1 + \mathbf{f}_2 = m_1\ddot{\mathbf{p}}_1 + m_2\ddot{\mathbf{p}}_2 \tag{2.9}$$

The sum of the external forces is therefore equal to the sum of the mass accelerations i.e:

$$\sum \mathbf{f} = \sum_{n=1}^{N} m_n\ddot{\mathbf{p}}_n \tag{2.10}$$

Interestingly, Equation 2.10 works for any number of particles. In a system consisting of three particles for example, there could be as many as six internal forces, all of which sum to zero. This implies that the acceleration of any system of particles is not influenced by any internal forces there may be, although the motion of individual particles within the system certainly is. The macroscopic motion of a system is therefore determined by the external forces which are applied to it. Expanding Equation 2.10 gives

$$\sum \mathbf{f} = m_1\ddot{\mathbf{p}}_1 + m_2\ddot{\mathbf{p}}_2 + \cdots + m_n\ddot{\mathbf{p}}_n \tag{2.11}$$

which can be rewritten as

$$\sum \mathbf{f} = (m_1 + m_2 + \cdots + m_n)\ddot{\bar{\mathbf{p}}} = M\ddot{\bar{\mathbf{p}}} \tag{2.12}$$

where M is the sum of the masses in the system. Equation 2.12 shows that there is a unique point C associated with the system whose acceleration is $\ddot{\bar{\mathbf{p}}}$. This acceleration is the same as it would be if all the particles in the system were located at C. This tells us that the total external force is equal to the rate of change of linear momentum, which is in turn the same as the rate of change of linear momentum of a system whose mass is concentrated at C. Integrating

Equation 2.12 twice gives

$$\sum (m\mathbf{p}) = M\bar{\mathbf{p}} \tag{2.13}$$

from which we get

$$\bar{\mathbf{p}} = \frac{\sum (m\mathbf{p})}{M} \tag{2.14}$$

C is called the centre of mass (COM) (sometimes also called the mass centre or, somewhat misleadingly, the 'centre of gravity') and Equation 2.14 enables us to find its position. The definition and properties of this COM are summed up by the following quotation [Lew71]. "There is a mass centre C associated with a set of particles:

- Whose position is independent of the origin chosen,
- Whose motion is independent of any internal forces,
- It is typical of the system as a whole since, if every particle were to have the same motion, it would be the motion of C,
- C lies within the convex polyhedron bounded by the particles,
- Its motion is as if all the particles were concentrated at C with all the forces acting together at C on them."

2.3.2 Rotational motion

To obtain a model for the rotational motion of rigid bodies, the turning moments have to be taken into account. A similar derivation can be used for a rotation of a collection of particles, again starting with Figure 2.2 and Equations 2.8, moments

can be taken about O to give

$$\mathbf{p}_1 \wedge \mathbf{f}_1 + \mathbf{p}_1 \wedge \mathbf{f}_1' = \mathbf{p}_1 \wedge m_1 \ddot{\mathbf{p}}_1 \tag{2.15a}$$

$$\mathbf{p}_2 \wedge \mathbf{f}_2 + \mathbf{p}_2 \wedge \mathbf{f}_2' = \mathbf{p}_2 \wedge m_2 \ddot{\mathbf{p}}_2 \tag{2.15b}$$

Again, the internal forces will have a cancelling effect on each other,

$$\begin{aligned} \mathbf{p}_1 \wedge \mathbf{f}_1' + \mathbf{p}_2 \wedge \mathbf{f}_2' &= (\mathbf{p}_1 - \mathbf{p}_2) \wedge \mathbf{f}_1' \\ &= 0 \end{aligned} \tag{2.16}$$

since the interaction is parallel to $(\mathbf{p}_1 - \mathbf{p}_2)$, resulting in

$$\mathbf{p}_1 \wedge \mathbf{f}_1 + \mathbf{p}_2 \wedge \mathbf{f}_2 = \mathbf{p}_1 \wedge m_1 \ddot{\mathbf{p}}_1 + \mathbf{p}_2 \wedge m_2 \ddot{\mathbf{p}}_2 \tag{2.17}$$

The moment of the external forces is equal to the sum of the moments of the mass accelerations. Generalising the above equation gives

$$\sum_{n=1}^{N} \mathbf{p}_n \wedge \mathbf{f}_n = \sum_{n=1}^{N} \mathbf{p}_n \wedge m_n \ddot{\mathbf{p}}_n \tag{2.18}$$

Again this holds for any number of particles as all the moments of the internal forces will cancel in pairs as in Equation 2.17. This is a very important result because it tells us about the motion caused entirely due to the turning moments generated by external forces on the system. Maintaining the parallel with the derivation in the previous section, we can rewrite Equation 2.18 as

$$\sum_{n=1}^{N} \mathbf{p}_n \wedge \mathbf{f}_n = \left(\sum_{n=1}^{N} m_n \mathbf{p}_n \right) \wedge \ddot{\bar{\mathbf{p}}} \tag{2.19}$$

Where $\ddot{\bar{\mathbf{p}}}$ is the instantaneous acceleration, so at any one instant in time this equation will hold. While it is acknowledged that in general $\ddot{\bar{\mathbf{p}}}$ is unlikely to be

constant for any significant time, to keep the derivation simple this is assumed to be the case. A notable case where this is true is within a uniform gravitational field in which $\ddot{\mathbf{p}}$ is observably constant. Now we can rewrite this bringing it closer to Equation 2.12 with which we are trying to draw a parallel.

$$\sum_{n=1}^{N} \mathbf{p}_n \wedge \mathbf{f}_n = M\bar{\mathbf{p}} \wedge \ddot{\bar{\mathbf{p}}} \tag{2.20}$$

This can be rearranged into a more convenient form to give

$$\sum_{n=1}^{N} \mathbf{p}_n \wedge \mathbf{f}_n = \bar{\mathbf{p}} \wedge M\ddot{\bar{\mathbf{p}}} \tag{2.21}$$

This tells us that there is a torque acting on a unique position relative to the COM.

While this result is interesting, we cannot determine the motion of a disconnected system of particles without full knowledge of their interactions. Moreover, $\ddot{\mathbf{p}}$ has to be the same for each particle for the equation to hold. Hence it is not really meaningful to talk of the rotation of such a system but once they are connected, as in the case of a *rigid body*, their velocities and accelerations are related.

2.4 Action of force on a rigid body

In this section we describe how the motion of a rigid body is affected by the forces applied to it. A formal definition of a rigid body is given in [SB90], and is summarised by the quotation

> If in any displacement the distances between three reference particles P_0, P_1, P_2 and the body co-ordinates of each particles $P_i (i = 3, 4, \ldots, N)$ remains unaltered, then the body is described as a rigid

body.

Less formally, this means that a rigid body consists of particles which are locked together and hence move in unison. We deal with the mechanics of bodies in this section, and as such it is useful to add new shorthand conventions to the notation. Much of the mathematics discussed so far can be extended to rigid bodies as though they were 'large particles'. To maintain this parallel, capitals are used for body properties such as mass, force or position.

2.4.1 Linear motion

A body's linear motion is analogous to that of a theoretical particle located at its COM. This particle can be thought of as possessing the sum of all the masses in the body. So, in body notation, the total mass M_A and force acting upon the body $\mathbf{F}_A$ (only useful for linear motion) of a body A consisting of n particles are defined as

$$M_A = \sum_{i=1}^{n} m_{A.i} \tag{2.22}$$

$$\mathbf{F}_A = \sum_{i=1}^{n} \mathbf{f}_{A.i} \tag{2.23}$$

Now it is possible to write body A's linear acceleration $\mathbf{A}_A$ in this notation

$$\mathbf{A}_A = \frac{\mathbf{F}_A}{M_A} = \dot{\mathbf{V}} = \ddot{\mathbf{P}} \text{ (c.f. Equation 2.5)} \tag{2.24}$$

Maintaining the parallel between a particle and a rigid body whose mass is concentrated at the COM, $\mathbf{P}$ is

$$\mathbf{P}_A = \frac{[\sum_{i=1}^{n} m_{A.i}\ \mathbf{p}_{A.i}]}{M_A} \text{ (c.f. Equation 2.14)} \tag{2.25}$$

2.4.2 Rotational motion

It has been shown that the moment exerted by the external forces on a rigid body causes rotation but the shape, or more precisely the distribution of mass within the body, also affects it. An equation equivalent to Newton's Second Law exists for rotation. Compare

$$\boldsymbol{\tau} = \frac{d}{dt}\mathbf{I}\boldsymbol{\omega} \tag{2.26}$$

with

$$\mathbf{f} = \frac{d}{dt}m\mathbf{v} \tag{2.27}$$

$\boldsymbol{\tau}$ is called torque and is the rate of change of the total moment of momentum. The rotational equivalent to mass, $\mathbf{I}$, is the *moment of inertia*. Unlike mass, it is not a scalar, but is in fact a 3 by 3 matrix based at any one instant on the shape of the object in question. Imagine applying a moment to a plate: it will behave differently depending on the axis about which the rotation takes place.

The derivation of the relationship between the forces applied and the resulting angular velocity $\boldsymbol{\omega}$ is shown in Appendix C.2 [SB90], finally arriving at how to calculate the moment of inertia

$$\mathbf{I} = \begin{pmatrix} A & -F & -E \\ -F & B & -D \\ -E & -D & C \end{pmatrix} \tag{2.28}$$

where

$$A = \sum m(y^2 + z^2) \qquad D = \sum myz$$
$$B = \sum m(z^2 + x^2) \qquad E = \sum mzx$$
$$C = \sum m(x^2 + y^2) \qquad F = \sum mxy$$

The parameters x, y and z are the distances (in the corresponding dimensions) from the COM to each particle in the body. Only in the case of a body consisting of point masses can the moment of inertia be calculated in this way. To compute the inertia tensor of bodies which are more complex, calculus techniques are used to break the shape into an infinite number of points, the moment of inertia of each is computed, and then recombined to give the final moment of inertia. Tables of common shapes and their corresponding moments of inertia are widely available.

It is worth pointing out that in a world co-ordinate frame a body's inertia changes as it rotates. This can lead to rotational motion which appears quite complex, as in the case of a Frisbee thrown into the air off-axis. Many researchers maintain objects in local co-ordinate systems to avoid an inertia calculation for each frame.

2.5 Simulation of motion

Physically based simulation of motion involves integrating the equations of motion over a time step using some suitable method. Researchers normally classify their systems as *forward* and/or *inverse* dynamic systems. These terms refer to the nature of the question posed. For example in forward dynamics we are asking the question: given an initial state, what is the state at time t? Whereas in an inverse dynamics system the question being asked is: what forces are needed to ensure a certain outcome? When the forces are known, they are incorporated into the

model and then integrated over the chosen time step. This section covers these terms in more detail together with a discussion on two widely used integration methods.

2.5.1 Forward dynamics

Simulations which involve applying forces and/or torques to bodies and observing the outcome are known as "forward dynamics" problems. The equations of motion will relate subsequent positions to forces, current positions, accelerations, velocity and other state information, usually as a second order Differential Equation. All the state information is known from the previous iteration and the simulation is advanced by integrating the forces over the time interval. A number of numerical techniques exist to perform such computations, each varying in their degree of accuracy and complexity.

The general case of solving higher order ordinary differential equations (ODEs) can be reduced to a set of n coupled first order differential equations [PTVF92]. For the functions y_i, where $i = 1, \ldots, n$ these equations have the general form

$$\frac{dy_i(t)}{dt} = f_i(t, y_1, \ldots, y_n) \tag{2.29}$$

where the functions f_i are known. However, to solve these equations further information about the type of problem is required. The functions y_i in Equation 2.29 have boundary conditions which can be satisfied at particular points. The type of boundary conditions that exist usually provides a clue as to which numerical integration technique is most appropriate. The specific type of problem addressed in this thesis is called an initial value problem. In this case, all the values of y_i are known at some starting point and the values of y_i are computed at some desired time t. The two most popular numerical methods to solve this type of problem are called Euler and Runge-Kutta, and these are considered below.

Euler method

This method is the simplest and least accurate numerical integration technique available. The algorithm uses the following formula

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2.30}$$

This advances a solution over the time interval h from t_n to t_{n+1} using the derivative at the beginning of that interval. Naturally, less error is introduced if the step size is very small, meaning that the change in the derivative over the time step is negligible. Accuracy can be traded off against performance for VR applications particularly since we are interested in plausible motion. In this context the most important issue, aside from being responsive to user interaction, is believability. Does the result convince the user of plausible motion?

Runge-Kutta method

Runge-Kutta is an extension to the basic idea of the Euler method. The algorithm involves performing some trial steps to gain further information about the derivative in a given interval. The simplest type of Runge-Kutta method involves using the derivative at the start of the time step to find the midpoint of the interval. The new derivative at the midpoint is then used to integrate over the whole interval.

$$k_1 = hf(t_n, y_n) \tag{2.31a}$$

$$k_2 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \tag{2.31b}$$

$$y_{n+1} = y_n + k_2 + O(h^3) \tag{2.31c}$$

This is a second order Runge-Kutta because by convention the error term $O(h^{n+1})$ indicates the order n of the method.

The most frequently used Runge-Kutta method is the fourth order Runge-Kutta which requires four calculations of the derivative for each time interval.

$$k_1 = hf(t_n, y_n) \tag{2.32a}$$
$$k_2 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \tag{2.32b}$$
$$k_3 = hf\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \tag{2.32c}$$
$$k_4 = hf(t_n + h, y_n + k_3) \tag{2.32d}$$
$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \tag{2.32e}$$

In general, the fourth order Runge-Kutta is more accurate than the second order except when the time interval is too short.

Since the Euler method is the simplest to implement and can offer a performance advantage over Runge-Kutta it was considered to be a suitable integration method to use. We acknowledge that this method has some disadvantages; in particular the Euler method may result in an accumulation of error faster than Runge-Kutta over a simulation. We argue that ultimately most numerical integration methods only give an approximate result [PTVF92] so it is not possible to ensure complete fidelity of an integration method.

2.5.2 Inverse dynamics

Forward dynamics is sufficient to solve many types of problem, such as finding where Mars will be at any given date and time in the year 2010. However many problems have an interrelation between components of a scene which requires more information about the forces exerted before forward dynamics can be performed.

A classic example of this is a pendulum, where the pin both exerts forces on the pendulum (otherwise it would fall to the floor) and is itself exerted upon.

The force exerted on the pendulum and pin, if known, can be used in order to determine its behaviour. This calculation, computing forces from the current and desired state of a scene, is known as *inverse dynamics* because the equations used in forward dynamics are now used in reverse. In other words force is being computed rather than acceleration.

Finding forces which will meet a set of criteria is normally beyond analytical means, and so requires numerical methods which vary forces until optimal values are found. Once suitable forces have been computed, the time step is advanced by performing a forward dynamics calculation. A good discussion of dynamics techniques can be found in the SIGGRAPH 1990 course notes [Wil90].

2.6 General particle based models

A particularly successful class of physically based models for computer graphics are particle systems. Researchers have used particle based models for simulating a wide range of natural phenomena from fire, waterfalls and wind-blown leaves to steam rising from a cup of hot tea [Ree83, Sim90, WH91, SF93, MM99]. These often beautiful effects are achieved by rendering particles in different ways and often changing their attributes as some function of time. All particle based models have some type of force acting on the particles in a simulation. Particles can be either interacting or non-interacting. The former type of system is one in which internal forces are present, i.e. each particle exerts a force on other particles in the system. This type of system is shown in Figure 2.3, the dashed lines represent forces between pairs of particles.

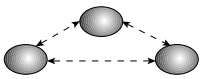


Figure 2.3: An interacting particle system

Newton's law of gravitation states that any two particles in the universe are attracted to each other by a force

$$F = G\left(\frac{m_1 m_2}{r^2}\right) \tag{2.33}$$

where r is the distance between particles with masses m_1 and m_2 and G ($= 6.672e^{-11}Nm^2kg^{-2}$) is the gravitational constant. Although this law can be used to compute inter-particle forces, the magnitude of this force will often be negligible between a significant number of particle pairs, due to their mass or distance, and can therefore be omitted [Gia84a, Gia84b]. For computational efficiency, most interacting particle system models are simplified by considering a particle's force to have a finite scope. This is a reasonable simplification because a particle's gravitational force is usually a rapidly decreasing function of distance.

Non-interacting particles are totally independent of each other. The motion in such systems is completely governed by external forces, such as gravity within earth's field. Figure 2.4 shows a particle system subject to the external force g.

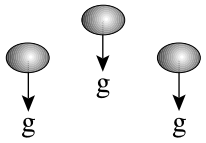


Figure 2.4: A particle system with gravity acting on it

This class of particle system is very attractive from a computational point of view because it is so simple. Some particle systems combine internal and external forces to achieve interesting effects at a moderate computational cost.

2.6.1 Simple particle models

We classify simple particle models as those which are manipulated exclusively by external forces. These types of systems have been used very successfully to

animate fire, fireworks [Ree83], smoke [SF93] and waterfalls [Sim90]. The stunning effects achieved by using these particle based models can be attributed to two important aspects; firstly, the natural evolution of the motion of particle systems under external forces and secondly the way in which they are rendered. William Reeves [Ree83] used a simple particle based model for the wall of fire (the "Genesis Effect") in the film "Star Trek II: The Wrath of Khan" [PMT82]. His subsequent paper was instrumental in launching particle systems as a useful simulation technique in computer graphics. Reeves considered a particle system to be a collection of non-interacting minute bodies which collectively represented a fuzzy object. In his simulations, particles were born with a particular state represented by attributes which define their colour, transparency, size and lifetime. The first three attributes may change over the life of the individual particles.

He concluded that a particle based approach was promising for simulating clouds, fire and smoke. Particles can be efficient to render when treated as point light sources. Also, rendering them as line segments could give the illusion of motion blur at low computational cost.

Non-interacting particle models of the type described above are excellent for use in special effects. However, as we shall see, they are applicable only in a limited context. Hence, many researchers have attempted to take the basic elements of such models and explore how they could be used to achieve more complicated effects.

2.6.2 Complex particle models

There are fundamental limitations to the effects achievable by simple particle systems. One particular problem associated with particle systems is the lack of control over the configurations that the systems finally settle into. For animation purposes, a certain outcome may be desired but it is often difficult to assert

sufficient control on a particle system to achieve that outcome. Many researchers considered the possibility of introducing added complexity to encourage particles to adopt certain desirable configurations, in preference to less desirable ones. Some of these approaches are classified here.

Interacting systems

This type of system may at first appear to be a slight variation on a simple particle system. However, the inclusion of internal forces adds a significant degree of complexity to the behaviour of the overall system. Each particle is now compelled to move in a number of different directions, shown by the dashed vectors in Figure 2.5. The vectors f and g represent external forces, and the dashed vectors show possible directions of motion resulting from the combination of internal and external forces acting on the particle.

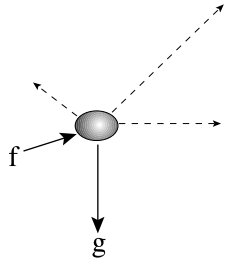


Figure 2.5: Which way do I go now?

The final direction the particle is influenced to move in is the vector sum of all the force vectors acting on it.

Possibly the best and simplest example of an interacting particle based model was presented by Reynolds [Rey87]. He simulated flocking behaviour of animals for animation purposes. Each flocking animal in the system was represented by a particle. The overall flocking behaviour was achieved by moving individual particle 'actors' according to rules which specified their motion relative to each other, and is an example of a distributed behaviour model. Interestingly, every

particle in Reynold's model has the same programmed behaviour. This is common to most interacting particle systems. The work described in this thesis takes the idea of specifying particle behaviour much further as will be shown in Chapter 6.

An interesting model was published by Yi Wu et al. [WTT95] in which deformable surfaces were simulated using particle systems. In this particular work, they achieved some very effective results at low computational cost by treating a deformable surface as being composed of many particles. Each particle was subjected to forces when it moved away from its rest position. The balance of these forces governed the extent to which particles were able to return to their initial positions. Using this method, they were able to create and animate skin (with or without wrinkles), cloth, paper, rubber and plastic sheeting.

Miller and Pearce [MP89] illustrated the value of interacting particles for animating viscous fluids. They defined objects called globules, which were the primitives of their system. Inter-globule forces were specified such that soft collisions could occur between them. This enabled their globules to flow over each other, within the limits they imposed on interactions. By varying the interaction behaviour they were able to simulate the flow behaviour of a wide variety of materials including powders, fracturable solids, solid-liquid transitions and jets or sprays of liquids. The important point to note about this particular work is the range of materials they were able to simulate at near interactive frame rates.

There are a number of other particle based models which illustrate the flexibility of the approach notably [LJR+91, Ton91]. In both models particles are animated separately and their motion is controlled by internal and external forces. By varying the internal force functions the properties of the materials are changed from solid to liquid or elastic to plastic. Moreover, topological changes are easy to effect since each entity in the model is discrete.

Oriented systems

The next stage of development, after making particles interact, was due to Szeliski and Tonnesen [ST91]. They introduced some simple attributes to their *Surfels*, an extension to the simple point mass representation of particles. Surfels are considered to be minute surface elements each possessing a radius and a spatial orientation vector. Consequently, the interaction between their surfels was preferential in certain orientations. Linear and angular accelerations of particles are computed using the standard Newtonian equations of motion. The particles are held together by long range attraction/short range repulsion forces derived from the Lennard-Jones potential functions. This type of function has an equilibrium distance where the attraction and repulsion between the particles are balanced.

To affect surface forming behaviour, they derive an interaction rule from the weighted sum of three potential functions which are zero when favourable conditions are met. Forces and torques arising from each potential attempt to place particles in desirable relative positions. These functions when integrated represent the total potential energy of the system during simulations. The potential energy of the system is minimised as this energy state imposes the least geometric strain on the system.

Szeliski and Tonnesen also developed a suite of tools for interacting with their surfaces; these tools could change the orientation of particles thus achieving creasing or tearing of their surfaces. In addition, they developed a particle growing algorithm which enabled stretching and reconstruction of surfaces from sparse data sets. This particular work was an interesting development in particle based simulation because, prior to it, particle based models were primarily volume based, and were not used to model and reconstruct surfaces.

Spring-mass-damper systems

The early development of particle systems research took two relatively different directions. Some researchers chose to develop the concept of interacting particles with sophisticated attributes and rendering methods, whereas others investigated the effects of connecting up point masses into organised meshes. In many of these approaches materials such as cloth [BHG91, BHG92, BHW94, CDR96] are typically represented as height mapped 2D grids connected by deformable units.

Terzopoulos and Fleischer [TF88] used a model consisting of a spring in parallel with a spring and *dashpot*, shown in Figure 2.6[1], where a dashpot is defined as a loose fitting piston in a cylinder containing a liquid of a given viscosity [Cow73].

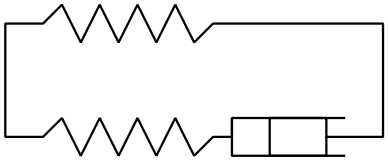


Figure 2.6: A mechanical model for deformation

The spring acts as a store of recoverable energy, since springs deform elastically and linearly to applied force. The dashpot on the other hand represents the loss of energy, in a form such as heat when a material is deformed, at a rate proportional to the applied force [FvFH90]. This type of mechanical model can be used to simulate the stress-strain relationship of materials capable of deforming. The combination of springs and dashpots in the model can significantly affect the deformation characteristics of the material.

The springs in Terzopoulos and Fleischer's units stretch gradually until the force on the dashpot reaches its threshold. At this point the dashpot slides and the spring attached to it compresses momentarily until the other spring takes up some of the load. Their results were very convincing and they showed their

[1] Adapted from Computer Graphics Principles and Practice [FvFH90] p. 1042.

materials draping naturally over obstacles and tearing.

In a subsequent paper Terzopoulos et al. presented a similar system in which they were able to simulate melting blocks of flexible material [TPF89]. The solid was represented as a network of mass points inter-connected by springs of variable stiffness. Each spring possessed a threshold 'temperature' at which it was destroyed. The world temperature was raised gradually in their simulations causing the gradual breakdown of the network. Force functions between particles were used to control the motion of particles that detached themselves from the network. Again they showed some very impressively rendered results.

Connected systems

Connected systems are similar to spring damper systems but we classify them in their own category because they use multiple meshes. For example in Wejchert and Haumann's animation "Leaf Magic", each of their wind blown leaves was modelled using primitives consisting of eight particle nodes connected by springs which represented the vertices of a flexible polygon mesh [WH91]. They defined the wind system as a set of basic 'airflows' which they combined to model complex wind patterns.

The technique of modelling objects by building them up from small meshes was used effectively by Miller [Mil88] for simulating the motion of snakes and worms. Each segment of Miller's creatures was modelled as cubes of masses connected by springs along the edges and diagonals of each face. Through detailed study of the locomotion methods of snakes, Miller was able to effectively capture the essence of this type of motion in his results.

More recently a similar method was used by Holton and Alexander [HA95] for modelling material behaviour by connecting up particles. Their soft cellular approach consisted of defining tetrahedral structural primitives called cells, each

comprised of mass points joined by deformable connectors. These cells were then used to build up larger objects such as muscles. In addition to deformable material behaviour, they were able to simulate fracture and breakage at low computational cost. They anticipated possible applications in virtual reality surgery training and their work represents a proof of concept of real time interactive connected particle systems. None of the above approaches used any means of inter-mesh interaction between primitives.

Implicit Surfaces

While the above systems have used particles which directly represent both the volume and surface of the materials, Desbrun et al. [DG94, DG95] treat particles as control points. The particle system is used to model large scale deformations. A body may be composed of any number of rigid skeletons, and a sizeable number are required to define the macroscopic shape of the body. Skeletons can interact with each other via force functions.

Local deformations of the surface are modelled by an implicit surface defined as some function of the underlying particle system. In other words the particle system implies the actual surface of the material. The implicit surface parameters are stored at each animation step, thus enabling subsequent high quality rendering.

Desbrun et al. were able to simulate the behaviour of soft substances such as clay or dough, particularly at boundaries where they tend to merge together when compressed. Moreover, they were able to compute animation sequences at interactive frame rates.

In general, these types of models suffer from the problem of unwanted blending between objects. This arises due to the fact that when two objects converge their implicit surfaces prematurely merge into one, and Desbrun et al. propose a

solution to this problem [DG95].

2.7 Summary of general particle based models

Each of the above techniques has been successful in terms of its ability to achieve the desired system behaviour. However, the techniques listed only use either particles and force functions, or spring dashpot type connections, as a means of limiting particle motion. They each have their particular strengths within the context of their application and illustrate the wide variety of effects particle based approaches can be used to achieve.

It is interesting to note that little research has been carried out on interacting articulated rigid body systems which enable body interactions to be defined using a particle based modelling approach. There are a wealth of effects to be achieved by experimenting with interacting structural primitives, one of the topics addressed in this thesis. In particular inter-body interactions between whole bodies or parts of bodies may provide a method of controlling and directing forces on bodies. To successfully generate articulated structural primitives with rigid connections involves looking into the domain of simulation of rigid body systems using constraint based techniques.

2.8 General constraint based techniques

Restrictions on many different types of systems are governed by rules which describe desired behaviour under a certain set of circumstances. Such rules can be called constraints and are often satisfied simultaneously to impose some condition. Constraint satisfaction enables the model to fulfil some goals specific to the nature and application of the model.

The idea of specifying constraints is frequently used in artificial intelligence

decision making applications. An example of this can be found in the case of simulation software used to aid the decision making process in nuclear emergencies [FPRS98]. Each countermeasure strategy has attributes which are weighted and the software minimises these, according to some initial criteria, to provide a ranking of strategies.

To algebraically solve these types of problems is very difficult except in the most simple cases thus, almost all constraint based systems use a suitable solver to find a solution to a problem subject to the given constraints.

Constraints can be satisfied in one of two ways, inexact or exact solution. An inexact approach varies the inputs to the solver to find a solution at which some measure of the state of the system is minimised or maximised. For example, the monetary cost of an action in a decision making system. The exact solution approach requires the solver to solve a set of equations to give a specific value.

The type of constraint satisfaction technique used gives an indication of the type of solver algorithm to use. We are interested in a subset of this subject area, in particular that of constraining the motion of bodies with an exact solution method. To provide context for our method we describe, in the following section, the types of constraints animators normally wish to apply and categorise a number of existing approaches.

2.9 Constraining particles or bodies

Typically animators constructing objects from bodies wish to somehow limit their motion while still obeying Newton's laws. Imagine trying to construct a chain from a number of bodies (links) and hanging it from a point halfway up a wall. To solve this type of problem involves keeping the joints between bodies intact, maintaining the position from which the chain hangs and computing its overall motion subject to the forces applied.

Common types of constraints applied to bodies involve either computing exactly, asymptotically or minimising values such as positions, velocities, accelerations, forces or energy. Some of the more common objectives which can result from these calculations are listed below.

A body to a fixed point This constraint (also called point-to-nail) ties a point on a body to a particular co-ordinate in space, as shown in Figure 2.7.

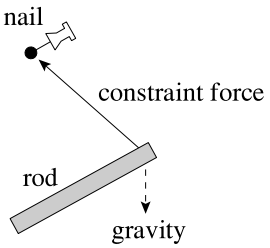


Figure 2.7: A body to a fixed point constraint

A body to another body Joints between two bodies are assembled using this type of constraint (also called point-to-point). Normally only two bodies can make up a joint. This type of constraint is shown in Figure 2.8.

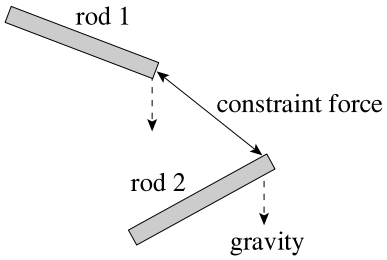


Figure 2.8: A body to another body constraint

A body to a path Also called point-to-path, it forces a point to follow an arbitrary path specified by the user as shown in Figure 2.9. For example constraining a *small* hoop to a wire superficially resembling the common fairground game. This is not the same as scripting a point moving along a

line or surface because the actual motion is determined by other events in the scene.

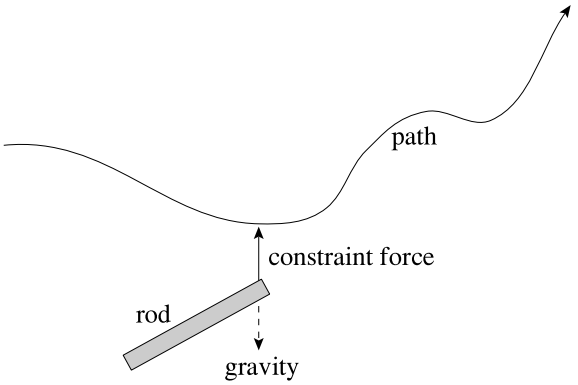


Figure 2.9: A body to a path constraint

The orientation of a body This type of constraint aligns an object by rotating it as in the case of posting a rigid envelope through a letter box.

Surface to surface This constraint can be compared loosely to a point-to-point constraint but with the distinction that the constraint is between any point on one body and any point on the other. This could be useful for animating a drop of water gradually progressing down the side of a glass.

Point inside surface A point on a body is maintained inside a particular surface but can move freely within, for example, a piston in an engine cylinder.

Collision and contact restrictions are not considered here, since while they are often talked about in terms of constraints they are generated as discontinuities in the simulation and have to be explicitly tested for.

There is a large body of research devoted to solving for the motion of constrained systems, from which a number of different methods for imposing the types of constraints described above have been investigated. These are classified below in terms of the class of solution method employed, according to Platt's classification [Pla92].

2.9.1 The projection method

This is the simplest method for enforcing constraints and usually involves exact solution of unknown constraint forces. The main benefits of this approach is that it is simple to program, is possibly the least computationally intensive and the constraints are fulfilled as soon as they are specified.

Drawbacks of this approach are that if two bodies are very far apart, the force required to instantaneously satisfy the constraint can be huge, thus leading to instabilities in elastic models. Secondly since the outcome is based on both forces in the system and applied forces, animators can find it difficult to achieve a specific outcome due to the difficulty in relating positions to forces. Models which lie in this category often have little in common with each other, because they use significantly different techniques to ensure the broad objective that all forces are computed to bring the system to an exact solution 'instantaneously'.

Isaacs and Cohen [IC87] implement a simulator where three methods of control are provided: "Kinematic constraints" for traditional keyframe animated systems, "behaviour functions" to relate the momentary state of the dynamic system to desired forces and accelerations, and finally "inverse dynamics" to determine the forces required to perform a specific motion. They are able to create joints with varying degrees of freedom allowing them to create pin joints, universal joints and ball and socket joints. This has allowed them to simulate a large repertoire of objects, including chains, trees, whips and a simplified representation of person on a swing.

Auslander et al. [AFP+95] experimented with articulated figures, primarily in 2D. Their simulations are different because equations are not solved to find forces as in inverse dynamics. Instead a searching algorithm applies inputs while trying to assess outcomes which are considered fit according to some criteria. This can take in the order of five minutes to evaluate 40,000 outcomes for a character to

perform a simple cartwheel. Clearly such simulations are too complex to run at interactive speeds.

Sohrt and Brüderlin [SB91] describe the application of geometric constraints to CAD/CAM, arriving at a very different type of simulator. The system attempts to maintain constraints between objects as a design is assembled. A novel feature is that the system will try to construct implicit constraints. So, for example, if a block is placed on a table it will try to retain this relationship, allowing it now to only move along the table's plane. Furthermore, when the table is moved, the object should be moved with it. Their constraint solver is used to restrict relationships to requested distances, slopes or directions. Fernando et al. use a similar technique in a VR context [FWT98].

These techniques avoid the disadvantages of the projection method by careful choice of application. For example by activating constraints only when they are appropriate, or by not considering elastic systems.

2.9.2 The penalty method

It is best to think of this method as one which introduces a restorative force to compensate for a body being pulled away from its desired position. Platt [Pla92] suggests that it is equivalent to adding a rubber band to a mechanical system, which attempts to pull the body onto a particular position.

The main advantage of this approach is that adding a rubber band to the model is simple, and so this type of model is considered to be easy to use but a number of disadvantages exist. Exact solution of the constraints is not guaranteed, but in general this is required because otherwise joints may appear to oscillate. Consider the example of animating an articulated figure such as an angle poise lamp. Unless specific Disney-like appeal is desired, it would be abnormal if the joints oscillated causing the lamp to jiggle.

Another important disadvantage is that as the restorative force increases, the differential equations representing the physical system become *stiff*. This means that there is little variation in the solution between time steps and so the solver spends a large amount of time computing solutions for each step without much to show for the computation. Finally, it is difficult to know how to increase the restorative force as the simulation proceeds.

Andrew Witkin et al. [WFB87] present this type of model. Their constraints are expressed as energy functions which behave like forces pulling the model into place and maintaining the condition. A simple energy function can be implemented as a spring connecting two points on different bodies. The motion of the system is computed by minimising the energy constraints. They supply a catalogue of basic constraints and corresponding energy terms.

In a later work Witkin et al. [WGW90] present a hybrid model which builds upon the penalty method by combining it with the *constraint stabilisation* method described in the following section. The objective of this work was to dynamically assemble objects by pinning together predefined parts with constraint forces. Naturally the reverse action could also be performed. Their constraint forces were a function of time so the constraint was met at a particular time t, thus generating a smooth animation. Using their model they were able to interactively add or remove constraints during a simulation.

2.9.3 Constraint stabilisation

Extra constraint forces are introduced to enforce exact solutions and can be considered as a variation of the penalty method. Essentially if the system starts to drift away from the constraint it is pulled back by an exact restorative function, the magnitude of which is called a Lagrange multiplier, rather than being pulled back by an approximate force exerted by a rubber band as in the penalty method.

Advantages offered by this technique are that constraints are exact and gradually fulfilled. Forces are added to the model but the animator is relieved of the burden of specifying forces to achieve a desired outcome, but the rate of constraint satisfaction is controlled by parameters which are themselves unintuitive. Worse still, for certain choices of these parameters, the system will oscillate multiple times around the point where the constraint is satisfied.

Platt and Barr [PB88] published an early approach using Lagrange multipliers to gradually find exact solutions for the motion of flexible models. They show examples of mouldable clay-like substances which maintain their new shape after strong deformation. Baraff [Bar89, Bar92a] also used Lagrange multipliers to impose collision and contact constraints between rigid objects. Witkin and Welch [WW90] use the stabilisation method to animate and control non-rigid structures.

2.9.4 Dynamic constraints

Barzel and Barr coined the phrase dynamic constraints in their publication [BB88] which presented one of the first approaches to modifying and applying stabilisation constraints in computer animation.

They wanted to control the speed at which constraints are satisfied so that they could animate assembly of models in a smooth and aesthetically pleasing manner. Thus they suggested a specific modification to the constraint stabilisation method which results in the system being pulled back asymptotically to the constraint with damped motion.

Dynamic constraints offer the same advantages as stabilisation constraints but with the added benefit that the rate at which a constraint is satisfied can be controlled. However the method can only be used with a limited set of constraints due to their constraint force formulation. Moreover, the constraints are

never really met exactly as the constraint is pulled back asymptotically towards zero [LKC93].

Other researchers have used dynamic constraints to constrain rigid models with articulating links. One example published by Peter Schröder and David Zeltzer [SZ90] contains images from their simulation of a car. The car is modelled as a body with four spring-damper shocks at the corners, together with two steerable constraint forces at the front wheels and brakes. During the simulation, the car was decelerated such that it would slide into a crossing, turning sideways by about 100 degrees. They interactively chose parameters for their spring-dampers which imparted a bouncy braking action.

Platt [Pla92] modified the constraint stabilisation and dynamic constraints methods to arrive at a hybrid technique which he calls *generalised dynamic constraints*. He uses this technique to assemble deformable computer graphics models in a smooth and aesthetically pleasing animation. He illustrates his method with an example of a sphere falling onto a trampoline and bouncing.

Finally another technique termed *sequential goal constraints* was published by Liu et al. [LKC93], to enable animators to set up sequences of transient constraints that can be activated and then released as soon as they are satisfied. With the dynamic constraints method, problems arise when the time in which a constraint is satisfied is small and a relatively exact solution is required. In this case, an object can appear to move rapidly towards its goal then remain near it for a long period of time before moving rapidly towards the next goal. The sequential goal constraints method deals with this problem while retaining the benefits offered by dynamic constraints.

2.10 Summary of constraint based techniques

It is surprising how much variation there is in the approaches taken to impose constraints on objects. Nevertheless, it must be said that although all the models described have been successfully used in animation and some of the behaviours achieved would be attractive to simulate in a VR system, most are still too computationally intensive. Even in the cases where constraint based techniques were used for interactive animation systems, often the authors published frame rates of simulations excluding rendering. In VR rendering the result is everything, and the physical model has to be simulated at a fast enough rate to be usable and viewable.

Another issue to consider is that animators find it difficult to script animations using constraints because the required programmed methodology can be unintuitive, and they often find it difficult to visualise the implications of applying certain constraints. However, currently this is not such a serious issue for a VE developer as most have programming skills. Furthermore specifying the exact motion of an object is often not critical in many VR contexts, for example if a pen is thrown along a desk the exact position it will end up in may be unimportant.

2.11 Hybrid models

Clearly a diverse set of behaviours can be modelled and animated using particle systems. Recall that in §2.6 particle based approaches for animating fire, waterfalls, flocking animals, soft substances, powders, cloth draping, and deformable material behaviours have been described. However, this range, albeit wide, does not cover the entire gamut of behaviours that would be desirable to simulate and interact with in virtual environments. For this reason another class of physically based models was described in §2.8. This approach is particularly successful for

animating articulated rigid body systems.

Since particle and rigid body approaches together cover a diverse range of possible physical behaviours it would appear sensible to be able to support both modelling approaches within a suitable framework. However, we have stated that many of these models are implemented for specific applications and often when these approaches are supported in a library they are found in isolation. For example Tabule [Fau99] and AERO [KSZB95] only support rigid body modelling. Other systems such as MathEngine [Woo99] and Barzel's 'fancy forces' approach [Bar92b] attempt to support both particle and rigid body models but their support for particle based modelling is currently limited. Barzel on the other hand provides his 'fancy forces' approach in order to script and attain some goal so although in principle he could simulate and model particle systems, he does not illustrate this.

A range of animation toolkits such as Alias/Wavefront [ALI], Softimage 3D [SOF], LightWave 3D [LIG] and 3D studio MAX [MAX] support a wide variety of different physically based modelling techniques [Mae96]. These systems are not directly relevant for discussion in this thesis, because they are toolkits aimed at users wanting to create and choreograph animations, and not middleware intended for use by developers building applications. Within the domain of VR a number of toolkits exist (described in Chapter three §3.2.2) which enable a developer to incorporate an often limited degree of physical simulation in applications, but they enforce many strict conventions which the developer must follow in order to implement an application. These toolkits generally have fixed application programming interfaces which are difficult to extend or customise.

For the reasons discussed above it would be attractive to support a hybrid approach which combines features of both particle systems and rigid body systems within a customisable and extensible framework for physically based modelling

in VR

2.12 Summary

In this chapter we have provided the foundation concepts necessary for later chapters. We have highlighted the point that Newton's laws are sufficient to describe most commonly observable behaviour making his model of reality well suited for many VR applications. Furthermore, we have presented a survey of existing particle and constraint based models. These are classified in terms of the models and methods used to solve the problem of specifying the motion of objects. In this survey we have shown that interacting particle systems alone are not sufficient for simulating complex behaviour because they are difficult to control. Researchers have investigated methods of imposing additional control on particle systems by restricting their motion through forces and connecting particles up. Rigid body simulation techniques share many concepts with particle systems and can provide a mechanism for simulating more complex constrained behaviours. However, few particle systems and rigid body systems are implemented to coexist in the same simulation so the benefit of this approach is highlighted. Finally note that in general both particle based and constraint based approaches have been successfully used in animation systems, but these do not necessarily have the same real time performance and graphical requirements as a VR system.

Chapter 3

Software engineering concepts

Since the objective of the research described in this thesis is to define and implement a prototype framework for physically based modelling in VR, a number of software engineering issues need to be discussed. In this chapter we describe the concepts of object oriented design, software frameworks and in particular motivate the need for a component framework with 'white box' (open source using the term in the broadest sense) components. Finally the use of scripting languages and motivation for using Perl to integrate components is described.

3.1 Object oriented design

Object oriented (OO) design is based on the idea that real world concepts and objects, such as an address or a person, may be represented as data structures in a computer program. These data structures incorporate data and functionality specific to the type of object or concept that they represent [Str87]. For example, a person may have some data to represent their name, age, gender and some methods to position and render them. Currently, OO design and implementation is a much favoured programming paradigm, and has a number of advantages and

disadvantages described below.

The OO approach lends itself well to modular design because of the way in which related data and functionality is packaged up together and this subsequently makes the programming task more manageable. The design of objects can be comparatively intuitive for certain applications, especially those which involve computer graphics. For example, a graphics window could be an object which may have some values (or attributes) to represent its internal state like its size, background colour and font metrics. To resize such a window may require a call to its resize function (or method). This association of a real world entity to a program data structure is relatively easy to conceptualise.

Another feature of object oriented programming is the ability to call a method on related objects for a similar end result, but with a very different implementation. For example, different shapes may all have a paint method which will cause them to be drawn on the screen, but the resulting code called would be different for circles and squares. This is possible due to dynamic binding, which enables the actual method call to be determined at runtime.

Many of the disadvantages are not directly associated with OO concepts themselves, rather with the amount of initial design required. Many programmers are unprepared for a substantial initial design effort and often may not actually know enough about the application to make the correct design decisions in advance.

Many OO languages suffer from a performance overhead caused by calls to dynamic methods. This can be seen as a serious disadvantage in any performance critical application such as a simulator with real time performance demands. Of all the most commonly used OO languages, C++ suffers the least from this problem as methods are bound at compile time by default. Another alternative, Java, is considered to be a safer language because the programmer is not responsible for memory allocation. However it can be argued that the output from current Java

compilers is not as optimised as that produced by C or C++ compilers. Ongoing research within the broad category of Java compiler, run-time optimisations and high performance computing testifies to this [AFG+00, Cox, FPS+00, Hay96]. In particular Alpern and Flynn-Hummel [AFH99] address issues related to high performance applications. They discuss potential sources of performance limitations in Java and survey existing approaches to solve these problems. Thus the advantages offered by using C++ does make it a suitable language to implement low level systems such as a simulator.

3.2 Software engineering frameworks

Broadly speaking, frameworks are reusable software architectural designs which specify an approach for developing applications within them. By their very nature they tend to be object oriented (OO) designs regardless of whether or not they are implemented in languages that support this particular programming paradigm. The reason for this is that they embody many concepts usually associated with OO, such as reuse of code, extensibility, encapsulation and data hiding. Furthermore a framework defines an application programming interface (API) for the problem domain that it is designed for.

Frameworks are useful because they provide core functionality at the middleware level to application developers. Consequently they are targeted at a particular subject domain. For example a graphics library which provides intermediate level functionality specific for VR developers could be considered to be a framework. The user (developer) will use functionality through predefined function calls, conforming to the prototypes supported by the library. Achieving a given action such as navigation is the main concern of the developer, while the technique used to achieve the action is unimportant. Clearly providing a

framework to developers is of significant value because they can be used to implement a wide range of applications all requiring the same core functionality. Consequently, robust and fully tested code can be reused in many different programs and the time required to implement large complex software systems can be greatly reduced.

From Chapter 2 we have seen that physically based modelling is a complex task and requires a large amount of core functionality to be implemented in order to be able to simulate soft body and/or rigid body motion. To support this physics in a VR context is valuable in terms of contributing to the realism and functional behaviour of a VE but currently a large amount of implementation is repeated for specific applications. In order to address this problem a framework for physically based modelling is presented in Chapter 4. However it is first useful to briefly discuss component frameworks in particular, describe some related frameworks which currently exist, and finally to consider implementation language and software engineering choices.

3.2.1 Component frameworks

A component framework is one composed of clearly defined separate software parts (or components). There is wide scale debate over what exactly constitutes a component and there is a large variation of granularity in the definition of the term. In other words a component could be a function, a complex data-structure, a software module or a complete software system. An accepted definition of a component was given by Szyperski and Pfister [SP96] and is quoted below:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Less formally, this means that a component has a clear and defined API through which it can be used, has limited dependencies on other components and can be used or developed in isolation without affecting any other components in the framework.

Components can be categorised into three different types; black box, white box and grey box [Szy98]. Black box components are those which can be used only through the specified interface and the code which carries out the operations is always hidden from the user. This is considered to be beneficial because the user only needs to know how to use the component not how it works, and such components can be further developed and replaced by new releases without updating programs that use them. It is widely believed that black box components lend themselves well to re-use and Szyperski [Szy98] supports this view. The argument for this case is based on two main considerations. Firstly, black box components are often developed commercially and as such significant effort is invested in support and ensuring that the product is robust. Secondly, such components cannot be developed or customised by third parties and thus the interface to the component can be maintained, thus applications which use them will not require further development in order to use upgraded versions of the components.

White box components on the other hand have their source code freely available. Szyperski [Szy98] states that this type of component is difficult to reuse for two main reasons. Firstly, developers are free to significantly change the implementation to suit their needs and secondly applications may rely on specific implementation details. However, he does acknowledge that there is an advantage to making a component's source code available in order to enable developers to better understand their functionality and how best to use it. Grey box components are a compromise in that part of their source code is hidden and parts may

be freely available.

We argue that white box components are more useful to applications developers for a number of reasons. Firstly, making the source code available allows the developer to better understand the functionality of the component. Secondly, the functionality of such components is much easier to extend. Thirdly, white box components are much easier to integrate into frameworks particularly because the types of problems developers will encounter cannot easily be foreseen so it is difficult to implement a perfect black box component. Finally, developers can exploit their knowledge about white box components to implement more optimised applications. We argue that these benefits more than adequately compensate for the reuse problems highlighted by Szyperski. Furthermore we argue that white box components can be reused over a broader class of application areas because they can be customised.

Component frameworks are particularly useful when implementing large complicated software systems which make use of a number of different techniques. For example, in order to provide a middleware system for physically based modelling in VR we can speculate that we would need at least three parts: a language to describe physical models and their behaviour, a simulator engine and a VR engine. The precise nature of these parts may be unclear at this stage and it may be difficult to see exactly what base functionality or techniques need to be supported, but a clear division of functionality can be seen. Furthermore once a prototype system is implemented, new components can be easily integrated into the framework using a clear prescribed technique. This has the benefit that a more sophisticated or updated component (perhaps the simulator engine in our example) can replace any existing ones in the system.

Component integration frameworks

Three main component integration frameworks are considered to be current industry standards. These are: Object Management Group's CORBA[1] [Gro99, Gro], Microsoft's COM/DCOM [MS] (Component Object Model [Box98]/Distributed COM [EE98]) and SUN Microsystems' JavaBeans [BEA]/Java RMI [RMI] (Java Remote Method Invocation). These approaches enable components to be glued together by implementing appropriate stubs (or wrapper functions). It is difficult to compare CORBA, COM/DCOM and Beans/Java RMI primarily because as functionality is introduced into any one standard it quickly becomes incorporated into the others, however a detailed comparison is presented by Szyperski [Szy98]. A problem with these frameworks as highlighted by Fayad and Schmidt [FS97] is that they currently lack the semantics and interoperability (the degree of ease in which components can be seamlessly made to co-operate) required to be effective over a wide range of different application domains. Furthermore as stated by Johnson [Joh97] most commercially available frameworks are specific to domains such as user interfaces development or networked applications.

The High Level Architecture (HLA) standard [U.S98] is possibly one of the most widely known component integration standards in the domain of distributed simulation and was developed by the US Defense Modeling and Simulation Office (DMSO). This standard prescribes the use of a 'simulation bus' into which a variety of different simulations can be incorporated. Each of these simulation components (called federates) are classed as members of a *federation* or distributed simulation. Each individual component computes a portion of the overall simulation and broadcasts updates to distributed clients. The problem with this particular standard is that it is very complex and currently supports over 125 different services. Furthermore it is designed for distributed simulation and as

[1] Common Object Request Broker Architecture

such is not particularly suitable for implementing the framework described in this thesis.

Designing for reuse and extensibility

The following quotation from Demeyer et al. [DMNS97] best describes the problems associated with designing component frameworks which can be widely reused.

> Unfortunately, the design of frameworks remains an art rather than a science because of the inherent conflict between *reuse*–packaging software components that can be reused in as many contexts as possible– and *tailorability*–designing software architectures that are easily adapted to target requirements.

Clearly individual applications each have their own unique set of requirements although these can be common for broadly similar application categories. For example a word processing application and a drawing application may use a common set of user interface components. However a drawing application will need some specific customisations unique to the purpose it is intended to serve. The problem of being able to customise components becomes even more severe when the application areas are very different; for example a word processor and a VR crash simulator.

In the business world there is an increasing tendency to use black box components. Evidence of this sentiment can be seen in the number of popular computing publications [Mey99, Szy99, Edw99] which feature articles describing the principles of component frameworks. However, although the idea of being able to plug together existing parts to implement a new system is attractive there are many reasons for wanting to be able to tailor or customise components. Using black box components can make it difficult for a developer to use knowledge specific to an application in order to develop a sophisticated or more optimised implementation.

We argue that it is helpful to consider reuse in terms of the degree to which it is supported. Core functionality can be reused across a range of applications while customised functionality can only be reused within the application domain. For example, vector and matrix mathematics components could be used across a range of computer graphics, drawing, and CAD applications. A customised vector component may then form the basis of a representation of a primitive in a particle system. When developing a vector and matrix component framework, it is difficult to see the complete range of functionality that will ever be required. Thus we believe the need for customising the core functionality of a framework for physically based modelling in VR is essential. There are two ways of achieving this, either by providing 'hooks' for user code or by making the source code freely available, and furthermore both approaches can be used in conjunction.

3.2.2 Frameworks in VR

A number of VR toolkits exist which may be classed as frameworks since they provide a middleware development tool for building general VR applications. Some of the well known VR toolkits DIVE [DIV], Meme [MEM], MR Toolkit [MRT], Tandem [Tan], MAVERIK [HKG+98] and DEVA [Pet99] are briefly described and classified as general VR frameworks.

General VR frameworks

The Swedish Institute of Computer Science's (SICS) Distributed Immersive Virtual Environments (DIVE) is a toolkit in which distributed VR applications can be developed [DIV]. It supports simple dynamic behaviour of objects that are described by Tcl scripts. Behaviour is executed as a result of events in the application, such as user interaction signals, timers and collisions. DIVE reads

and exports VRML together with several other 3D formats. Virtually no support for physical simulation is supported as standard but Anders Wallberg et al. [WHN+98] in a recent work describe spring mass damper models of deformable objects implemented using DIVE.

Immersive's Meme toolkit (Multitasking Extensible Messaging Environment) has an interpreted threaded language, allowing code to be typed at the command line while an application is running [MEM]. Meme supports articulated models but with only limited interaction and dynamic assembly or breaking.

The University of Alberta's Minimal Reality toolkit (MR Toolkit) provides a suite of low level software tools for implementing VR applications these include [MRT]; Object Modeling Language (OML), 3D modeler (JDCAD+), and Environment Manager (EM) to support multi-user networked applications. OML can be used to implement geometry and simple behaviour, there is no obvious support for physical simulations.

Tandem is a distributed interaction framework for implementing collaborative VR applications [Tan]. It uses CAVElib for VR projection display, and CAVERNsoft for networking and can be regarded as a component framework particularly because the architecture exploits existing patterns in collaborative VR. Tandem supports limited animation played upon events but no real support for physically based modelling in VR.

The University of Manchester's GNU MAVERIK [HKG+98] is a highly modular middleware C library for building VR applications also implemented in C or C++. As such it supports no physical behaviour but a complementary framework (DEVA [Pet99]) currently being developed is intended to support complex object behaviour over distributed applications. The DEVA framework supports behaviour through an environment hierarchy intended to represent the laws of each node in the virtual universe (or metaverse in the DEVA terminology). Child

nodes inherit fundamental rules which should be obeyed from the parent environment. All environments inherit from a root called the 'void' which has no properties of its own but acts as an aid to navigation around the metaverse. Currently applications may be implemented within the DEVA framework using C++.

Currently, all the frameworks discussed above do not support physically based modelling to any significant degree. This is largely due to the fact that most have been developed as general purpose VR toolkits. A few general libraries do exist for physically based modelling in VR and these can also be argued as being frameworks because they provide a middleware resource for implementing applications which use physically based modelling. A survey of the frameworks considered to be most relevant is presented in the following section.

Frameworks supporting physically based modelling in VR

MathEngine [Woo99] is one of the most general purpose C libraries for physically based modelling. It supports a wide range of physical modelling techniques and is able to simulate both rigid and deformable object properties. Applications which use this framework must be written in C and currently only limited support is provided for true VR as the display module is a simple OpenGL [OGL] implementation. MathEngine is unique in the sense that it does provide support for a wide range of physically based modelling techniques. Other systems generally support only deformable models or rigid body models. A number of examples in each of these categories warrant discussion.

Firstly, two systems for simulating deformable object behaviour in VR are described; Clayworks [CLA] and Chapman and Wills [CW97, CW98] unified system. Clayworks is library for supporting physically based modelling in VR and is currently under development. It has an as yet unfinished scripting language

and it is intended that the new version (3.0) will support a full object hierarchy, mesh deformation, bones, nurbs and metaballs.

Chapman and Wills [CW97] also present a proof of concept paper in which they discuss the applicability of modal analysis (an implementation of the finite element method) to physically based modelling of deformable objects in VR. In a subsequent work Chapman and Wills [CW98] demonstrate a technique for improving the performance of their model without loss of accuracy. Currently their system is not available under general release.

Systems which could be categorised as frameworks for constraint based modelling in VR include; AERO [KSZB95, KSB], Tabule [Fau99], Sced [SCE] and a system being developed as part of the Salford Centre for Virtual Environments' IPSEAM [FWT99] project.

AERO [KSB] was developed primarily for animation and uses a rigid body simulator. It contains a 3D scene editor for designing simple scenes based on bodies. Objects can interactively be placed and linked with joints or forces. AERO runs in two different modes called animation and batch. In animation mode, the simulations are carried out in real time but objects are rendered as wire frames. At each time step batch mode outputs the scene to a file in a format suitable for the 'POV Ray' ray tracing program [POV]. AERO is marketed as a VR system but it is really only able to play a simple predefined animation sequence in a VE. Little interaction with the animation is possible.

A more recent approach presented by François Faure considers the problem of physically based modelling in VR from an animation background. He has published a number of methods for speeding up the computation of constraints [Fau96, Fau99], which he illustrates through a simulator called Tabule. In general, he has optimised computations by performing pre-processing, where applicable, to convert the constraint solving problem into a more manageable one.

His algorithms are applicable to acyclic and cyclic structures and he can compute constraints for acyclic structures in linear complexity. The Tabule simulator is essentially a rigid body simulator which uses a Newton-Euler integration scheme and sparse iterative solution of a linear system of equations using the conjugate gradient method. His system performs best with scenes which are acyclic, and as such attempts are made to manipulate cyclic problems into a more suitable form before solving the equations. Scenes with a few or very many cyclic structures benefit little from this pre-processing, while with a moderately cyclic scene large improvements are seen. Furthermore he implements collision and contact constraints. Currently, the Tabule simulator is not used in a VR context and cannot simulate particle systems although Faure's intention is to incorporate these [2].

Sced [SCE] is a modelling package that utilises geometric constraints to edit objects in a virtual world and could be used at an intermediate stage from an application to create scenes and export them to a variety of rendering programs. As such it could be justified as a framework for modelling and editing geometric constraints. Editing a model within Sced is achieved by interactively indicating the conditions to be satisfied and then allowing the system to solve the constraint satisfaction problem. Furthermore indirect constraints may be applied through other bodies meaning that interdependencies may exist. A simple example of a stack of boxes is given in which each box must lay on the top of the other, and the adjacent planes must remain in contact. To achieve this, Sced constrains the position and orientation of each box in relation to the box beneath it. Thus when the bottom most box is moved, the stack follows as the system maintains the constraints between boxes. Sced does not allow full integration within a VR system, but instead supports the idea of outputting a scene graph to be rendered by an external graphics system.

Terrence Fernando et al. [FWT99] present a generic system for supporting

[2]Personal communications with Dr. Faure.

interactive assembly and maintenance tasks as part of a project called IPSEAM (Interactive Product Simulation Environment for Assessing Assembly and Maintenance). The IPSEAM system imports CAD models into a scene graph and allows the user to grab and manipulate objects in three dimensions via a virtual environment interface. Fernando et al. use a component based architecture to provide a framework specifically for interactive manipulation of CAD models. As such it can form part of a larger system providing services to a certain class of applications.

From the discussion above, it can be seen that relatively few systems support a wide range of physical behaviours in a virtual reality context. One reason for this is that it is a difficult objective to attain. This is particularly due to the complexity of physical models in general and the real time performance demands imposed by VR applications, an opinion supported by Chapman and Wills, Faure, Logan et al. and Pettifer [CW98, Fau96, LWA94, Pet99] amongst others. We argue that a component based approach is a better architecture to adopt for this type of system as this provides a large number of benefits in terms or code reuse, customisability and flexibility to incorporate a range of components into the framework.

3.3 Scene description

It is now appropriate to consider the problem of complex scene description. We believe a general framework for physically based modelling in VR should support simulation techniques for both rigid and flexible bodies, so scene description may involve construction of complicated models. For this reason we argue that an easy to use, full and complete language is necessary for scene description. Thus scene description language considerations and the choice of a high level scripting language as an application development language and component integration

standard are described.

In a complex physical simulation there may potentially be any number of different force functions acting on or between primitives together with a variety of different types of constraints in the scene. These types of scenes are difficult to describe in a low level language designed for systems development so it is valuable to use more appropriate tools for the task.

Any potential VR scene description language (SDL) should contain concepts capable of describing entities and behaviour from the real world in a virtual world. Since the physical models are composed of objects (in the OO sense), a language capable of supporting this programming paradigm would be desirable. Since the simulator engine is performance critical it should be implemented in an efficient language such as C++ however a SDL is not subject to the same efficiency demands. It is more important that the SDL is easy and safe to program in.

An ideal SDL should provide a flexible method for developing VR applications which use the simulator. Since the simulator could potentially be used in a wide number of applications, it is undesirable to impose too many restrictions on how it is used. The trend in animation has been towards interactive scene description, based on the argument that animators are artists not programmers [WW92]. This tactic widens the user base of the software and virtually anyone with a computer and the right software can animate characters [Mae96]. However, complex scenes consisting of hundreds of primitives and the same order of types of interactions would be tedious to set up interactively. Programmers on the other hand (as opposed to animators) want freedom to use systems in the most inventive and convenient manner, a philosophy adopted by MAVERIK which does not enforce specific data representations on the user. A good way to achieve this is by maximising customisability and extensibility.

Many graphics systems ranging from animation to rendering packages use

SDLs. A survey of such software reveals that the languages available vary considerably in their elegance and sophistication, but all share one common feature; they are often written from the ground up specifically for the application. A good example of such a language in the public domain can be found in Craig Kolb's raytracer RayShade [Kol92] which provides a powerful parser using the language generation tools `lex` and `yacc` [LMB92]. It would appear attractive to implement something similar in order to program simulations but this is a complex task. The language would ideally require support for OO concepts which are not easily implemented. Many suitable languages already exist so why "reinvent the wheel"?

This brings us to the important question: what criteria should a suitable SDL possess? In answer to this, it must be easy to learn, possess high level concepts, be extensible and preferably enable rapid development of applications. These criteria imply that a scripting language such as Tcl or Perl may be an appropriate choice.

3.3.1 Scripting languages

Many VR systems incorporate a limited degree of scripting in order to play a simple animation invoked by some event. In general these scripts are used purely to choreograph a particular motion sequence of an object. An elementary level of interaction is possible within this context because the script is driven by the VR system to perform a specific action, such as flexing an arm. The scripting languages discussed in this section are much broader in their scope than the more traditionally used VR languages. There are many reasons for using a high level scripting language in VR, but the most compelling is being able to control a scene through a full and mature language. In other words, the VR system is driven by the scripting language, this approach offers a much greater degree of flexibility in

the types of complex behaviours which can be simulated.

High level scripting languages are typically interpreted and are designed for gluing together components written in system programming languages (SPLs) such as C or C++. Being weakly typed they can easily be used to build connections between different software components thus enabling rapid development of applications.

Modern scripting languages are a far cry from their ancestors and are finding more wide-scale applicability. John Ousterhout [Ous96] discusses this general trend towards wider scale use, the merits of an untyped language and the applications for which a scripting language is suitable.

He suggests that "Yes" answers to the following questions indicate that a scripting language will work well for a given application.

- Is the application's main task to connect together preexisting components?
- Will the application manipulate a variety of different kinds of things?
- Does the application include a graphical user interface?
- Will the application's functions evolve rapidly over time?
- Does the application need to be extensible?

Scripting languages and SPLs complement one another because they are designed for performing different tasks. Complex data structures and functionality are best achieved by coding in a SPL whereas integrating two very different tools together is well within the capabilities of a modern scripting language.

The applicability of scripting languages has grown of late due mainly to the increasing performance of machines, better scripting languages, the need for graphical user interfaces and the growth of the Internet. Ousterhout argues a silent "scripting revolution" is occurring of which even the participants are unaware. He believes that scripting languages such as Perl and Tcl are going to bring about a new way of programming. They enable a significantly high degree of code reuse due to the high degree of interoperability supported by these languages.

Building an application by reusing components allows rapid development of applications provided good software actually exists. This feature of modern scripting languages is attributable to the weak typing that exists in these languages. Use of a weakly typed language is the easiest and most convenient way to glue together very different components each possessing their own data structures. Figure 3.1[3] correlates the capabilities of languages with the degree of typing found in them. Weakly typed languages are well suited to our SDL because they enable rapid construction of data structures and easy manipulation of data.

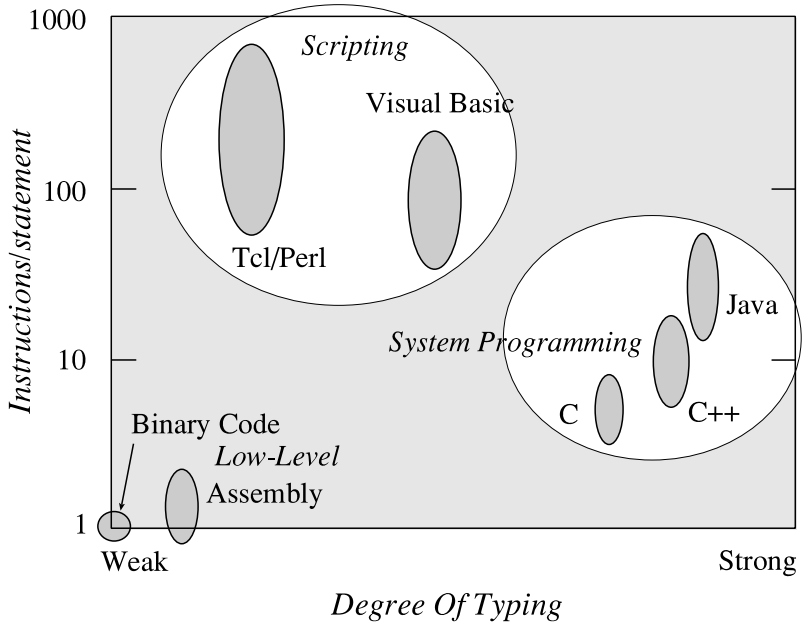


Figure 3.1: A comparison of various languages based on power and their degree of typing

It is generally accepted that there is an extra performance penalty incurred when using a scripting language. This is due to the fact that such languages are generally interpreted and each call in the scripting language often corresponds to at least one call in a SPL. The extra performance overhead is not considered to be a real concern here, as the performance critical simulator is written in a low

[3]Reprinted from Additional Information for Scripting - White Paper [Ous96].

level language and a relatively small number of scripted instructions will typically be executed per frame.

Scripting and objects

A convincing case can be made for using an object oriented language for scene description, because users building physical models to simulate may intuitively think in terms of the components which make up the scene. Therefore, it is natural to provide a direct mapping between real objects and conceptual objects. For example, if a user wants to create a body composed of ten particles, it would be quite natural to simply say

```
new body(10) ;
```

However, most object oriented languages are strongly typed and as such do not provide a simple means for using powerful components built at a lower level. A number of high level scripting languages provide support for object oriented programming together with the advantages offered by weak typing.

Choice of SDL

Ousterhout argues that possibly the two most powerful scripting languages readily available are: Tcl and Perl [Ous94, WCS96] We chose to use Perl because scripts are in fact fairly simple to write and the current version, Perl 5, supports 'Object Oriented' concepts which most importantly can be easily laid over C or C++ equivalent data structures.

The choice between Perl and Tcl really is an arbitrary one as they compare, feature for feature, very well with each other. If a new version of Perl has a unique feature then it is usually incorporated into Tcl and vice versa. Perl's main strength over Tcl lies in its object oriented capabilities.

Perl 5 provides an unusual mechanism for OO programming, but this is attributable to the desire to provide this facility with little additional syntax. Modular development already exists in the form of packages, which provide an environment for data and function hiding. To access them, they must either be exported from the package or an external function can make a direct call to the function. For example

```
chain::build_me() ;
```

The function `build_me` can be viewed as a method inside the package `chain`. These bear more than a passing resemblance to C++ static method calls[4], which can be argued as a benefit to programmers familiar with this syntax.

Furthermore Perl supports a powerful standard (called Perl XS) for component integration enabling us to extend the language and support our framework through a full and complete language. The merits of this choice could be debated at great length but the specific high level scripting language chosen ultimately comes down to personal preference.

3.4 Summary

To summarise, we have argued a case for using an OO approach to implement the low level simulator based on the belief that simulator concepts and building blocks for constructing physical models can be intuitively represented within this paradigm. The need for a component framework for physically based modelling in VR has been discussed and motivated; both in terms of the benefits offered by such frameworks and in terms of the current limited availability of middleware in this area. Finally the use of a high level scripting language to integrate and provide component functionality has been outlined.

[4] Also referred to as class methods in some languages.

Chapter 4

Iota: A framework for physically based modelling in VR

In this chapter the requirements for the Iota framework and an overview of the framework is given. The requirements for the framework are based on concepts and the state of the art research described in the previous two chapters. We take a top down approach to describing the framework, beginning with a section that presents a high level view of the architecture, followed by a detailed discussion of individual components in the system. Finally a discussion on the use of a high level scripting language to integrate the components is presented, with particular emphasis on the ease and limitations of integrating new components into the framework.

4.1 Requirements capture for the Iota framework

Requirements capture for the Iota framework was carried out largely through a survey of current systems available which support physically based modelling at

the middleware level. Furthermore, discussions with a number of researchers and developers in the field[1] helped to influence the final architecture for the system.

We argue that a wide range of rich and complex behaviours are valuable contributory factors towards the sense of presence that a user may experience while inhabiting a virtual environment. The following abridged quotation from Slater [SS00] addresses this sentiment:

> Imagine that you are in a park in the sunshine. You are walking through the park, perhaps admiring the trees. A particular tree is interesting, and you begin to move closer to it. As you get within a certain distance there is a moment when you become aware that in fact the 'tree' is flat - a virtual cardboard cutout. You recall that you're actually in the laboratory, wearing a head-mounted display (HMD), and it is the middle of the night.

The passage evocatively conveys the essence of immersion and the factors necessary to maintain the illusion. It implies that the richness of the park, the trees, the sunshine all contribute to the illusion until a point at which the image is shattered by the realisation that a particular tree does not appear to be real enough. Slater's scenario of a virtual park so enticingly real that the user forgets they are in a laboratory, focuses on three factors related to presence; richness of behaviour, image lag and disturbance from external stimuli. The first time his scenario is shattered as a result of a lack of richness in the environment. In later unquoted paragraphs the illusion is destroyed due to the image lagging behind the head movement of the user and later still it is due to an interruption from the building superintendent.

Although it is not easy to quantify how much rich behaviour contributes to the sense of presence in a VE, it is generally accepted that it is valuable to try

[1] Personal communications: François Faure, Stephen Pettifer, Daniel Kidger and Gary Powell

to incorporate complex behaviour in VR applications. Logan et al. [LWA94] present a survey of simulation techniques and conclude that VR represents a new application area for several techniques traditionally associated with engineering and non-real time animation applications. Pettifer [Pet99] proposes a framework for incorporating rich behaviour into distributed environments. Furthermore, there is evidence that rich behaviour can contribute to the sense of presence, for example the looming response described by Slater [Sla99] where participants know that there is nothing there but they still duck as a object flies towards them. Therefore we can conclude that it is justifiable to support rich behaviour in virtual environments and that a framework for physically based modelling is of value.

Recall from Chapter 2 where we highlighted the lack of systems which use a hybrid approach for physically based modelling in animation. This was further developed in Chapter 3 particularly with reference to the state of the art systems in virtual reality. We also argued that a hybrid approach is valuable because it enables a broad class of physical models to be simulated ranging from flocking behaviour, fluid behaviours, dust, powders, fire, firework displays, to soft materials and articulating snakes. Based on this we consider the first requirement of the Iota framework to be that it should support a hybrid approach to physically based modelling in VR. In order to achieve this, a purpose built simulator must be implemented.

Clearly for any VR system there are real time performance constraints, so we use this as a basis for the second requirement of the Iota framework: the ability to manage the complexity of simulations. This is justified by Barfield and Hendrix's [BH95] study of the effects of frame rate on presence in which they show that consistent frame rate is necessary. We argue that in order to maintain a consistent frame rate the complexity of simulations must be appropriately

managed otherwise certain frames could take longer to compute. Ellis suggests that an optimal degree of presence can only be maintained through trading off factors which contribute to presence against one another [Ell96]. We therefore consider it appropriate to trade off some degree of fidelity of simulations against performance, as long as the degradation in fidelity of the simulations cannot be perceived by an average user.

The third requirement for the Iota framework is the provision of adequate VR functionality. In Chapter 3 we described a number of systems or libraries which provide varying degrees of functionality for physically based modelling in VR. We also highlighted the fact that the more general systems have limited support for VR functionality largely because many were developed from an animation direction (cf. Tabule, MathEngine, AERO, Sced). The Iota framework should support the basic functionality (such as navigation and spatial management) necessary for VR because it is intended for use in this field.

A need for a full, complete and easy to use high level scripting language was also motivated in Chapter 3 together with the requirement that low level simulator and VR functionality should be supported through this language. This means that it should be possible to implement applications fully within this language. Schmalstieg and Gervautz [SG95] propose a virtual environment architecture inhabited by avatars which are controlled by users across a network. They use a high level scripting language (Python) to control and customise actors implemented in C++ and motivate this by stating that Python is more powerful for specifying customised behaviour of actors making the construction of applications fast and simple. In particular, they benefit greatly from the fact that Python is cross-platform and so can be used to pass messages (methods) between systems. Furthermore, they were able to modify the behaviour of actors at runtime, another trait of high level scripting languages. Although the context of the Iota

framework differs from theirs, the points they make to justify their approach are relevant and support our arguments for high level scripting.

An early system which used a limited degree of simulation in VR was NPSNET [ZPF+93]. Michael Zyda et al. describe this system within the general context of the software required for generation of virtual environments. NPSNET embodies some useful concepts, in particular Zyda et al. make a convincing argument to develop a strong foundation of software upon which applications can subsequently be built. They identify the six core elements of a VR system as being: navigation, interaction, communication, autonomy, hypermedia and scripting. This supports the idea of implementing a component framework which provides VR functionality together with scripting and simulation. Furthermore, the discussion on component frameworks in the previous chapter highlighted the benefits and disadvantages of the white box component based approach. It was argued that the advantages (customisation and extensibility) offered by this approach outweigh the disadvantages and so the final requirement for the Iota system is that it should be implemented as a white box component framework.

Now that the requirements of the overall framework have been identified it is appropriate to consider a high level view of the architecture of the Iota framework.

4.2 Overview of the Iota framework

The Iota framework consists of a tightly coupled simulator engine/complexity management module and a VR component. A high level view of this architecture can be seen in Figure 4.1. The simulator is purpose built and supports a hybrid particle-rigid body modelling approach. VR functionality is provided through MAVERIK and integrated into the framework. Above the low level component layer lie Perl XS stubs through which component functionality is made visible to the Perl scripting language (Appendix D contains a short tutorial which describes

how XS files are used). On top of this layer are a collection of Perl modules to support model construction, rendering customisations and any other user customisations. The user scripts are implemented in Perl and make calls to custom functionality from the Perl layer as well as low level functionality through the Perl XS layer. Interfaces to all components are supported through the Perl XS stubs. This protects the user by only making available functionality which should be accessed. Low level simulator and MAVERIK calls may make some callbacks into user code or custom modules which exist as Perl scripts.

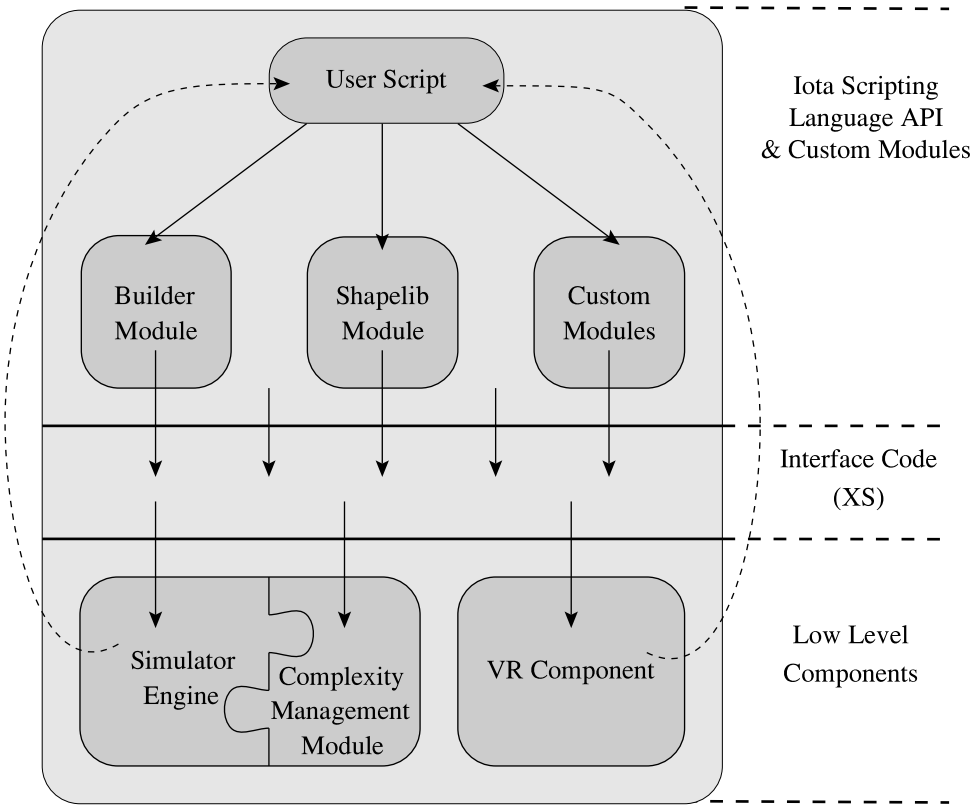


Figure 4.1: The Iota framework for physically based modelling in VR

A rich common data format is used within Perl to describe the simulator models together with their graphical representation. This has the benefit that the same data structure may be used to store data for both the simulator and MAVERIK rather than having to maintain different data representations. It is

helpful at this stage to examine each low level component in turn and then discuss the integration of these into the overall framework.

4.3 Simulator component

We have seen in Chapter 2 that many types of interacting particle and rigid body systems exist independently, but there is a missing link: there are few true hybrid models. The objective of simulation for general purpose computer graphics is clearly different to molecular modelling or fluid dynamics applications, so some optimisations can be made at the cost of accuracy, but the underlying theory is essentially the same. Particularly in terms of managing the complexity of physical models and reducing the simulation to that of the simplest model required to achieve the desired behaviour. This section describes the implementation of the simulator. Moreover, due to the interactive nature of the simulations, the scene structure may be very dynamic so various issues regarding connecting and disconnecting particles are highlighted [GM98]. Examples are presented throughout this section to help clarify the concepts.

We take a top down approach to describing the simulator. First an overview is presented together with a coverage of the notation required to understand the examples presented in this chapter. A specific example of a system one may wish to simulate is given and attention is drawn to the problems associated with computing its motion. This leads on to a discussion of the core mathematics required to simulate such systems. Finally specific implementation details and algorithms are described.

4.3.1 Overview of the simulator

A high level overview of the core simulator is shown in Figure 4.2. Essentially the core simulator consists of a block which advances a frame by computing required forces and advancing a time step in a simulation. If any events have caused a change in the scene graph then a scene modification block performs some pre-processing to reconstruct the new scene. The techniques used in the simulator are described later in this chapter. It is important to note that advancing a frame purely means moving forward in the simulation by an arbitrary time step, the duration of which can be adaptively modified.

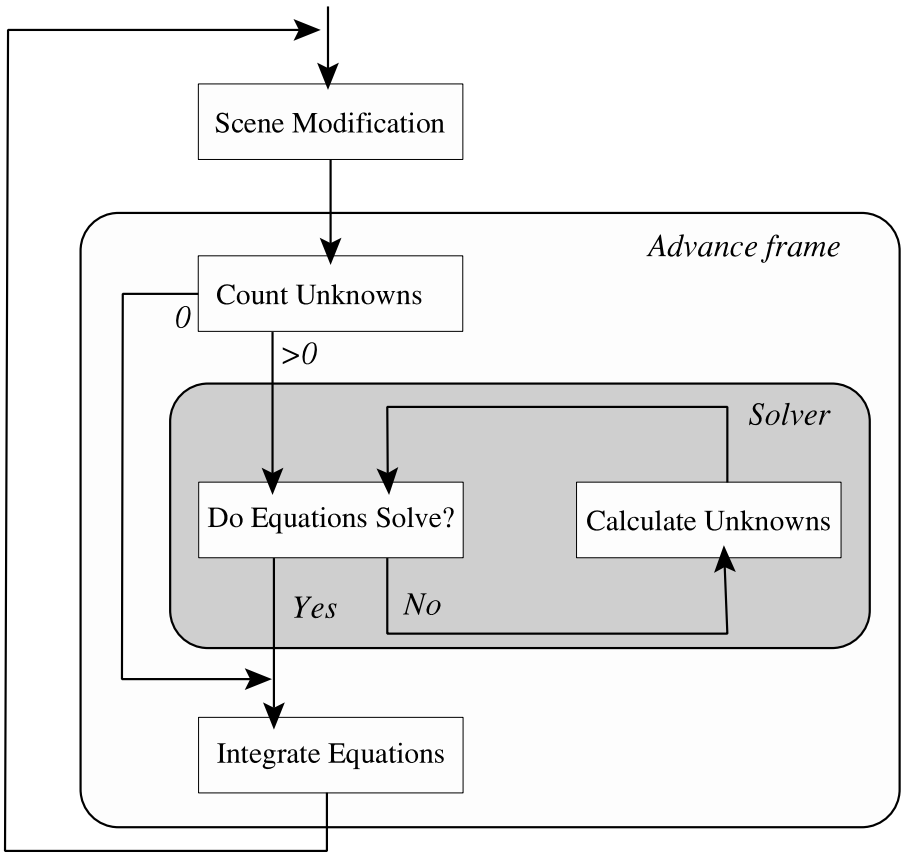


Figure 4.2: Overview of the flow of control in the simulator

Advancing the frame involves a number of different types of computation. Recall that the physical model incorporates a particle system and articulated

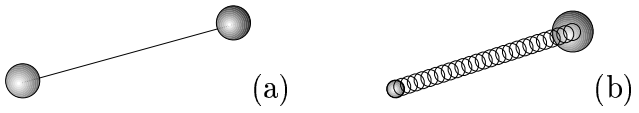


Figure 4.3: Two types of bonds

rigid bodies so a scene could consist of any number of interacting particles or articulated rigid bodies. Indeed, interactions may also occur between particles which coexist within articulated bodies.

At this point it is worth digressing a little to consider the conventions adopted to graphically represent various components in the scene. The conceptual building blocks used to construct scenes to be simulated, and the conventions adopted to represent them are described. We start by presenting point masses or *particles*, a number of which are shown in Figure 4.3, and are the basic building blocks available to the user. Although particles are technically 'point masses' and should therefore be represented as dots, this is unhelpful in terms of visualising them. Hence, the radius of the sphere is used to illustrate each rendered particle to give an indication of its mass in a simulation.

Particles may be connected together using *bonds*. Figure 4.3 shows two different types of bond. The first, shown in part (a), represents a *rigid bond* which cannot change length. The second shown, in part (b), is a *flexible* harmonic oscillator or 'spring-like' bond which exerts a force on the two particles to which it is connected. Such bonds can be modelled as force functions.

Now consider a *system*, a collection of entities which only interact with themselves, as shown in Figure 4.4. Hinge particles are represented by a black dot. These are connections about which rigid sections are free to rotate. The five particles P3, P4, P5, P6 and P7 are connected using rigid bonds and thus move in unison. Such an entity is referred to as a *body*; each separate body will be labelled by convention $A, B, C \ldots$ and so on.

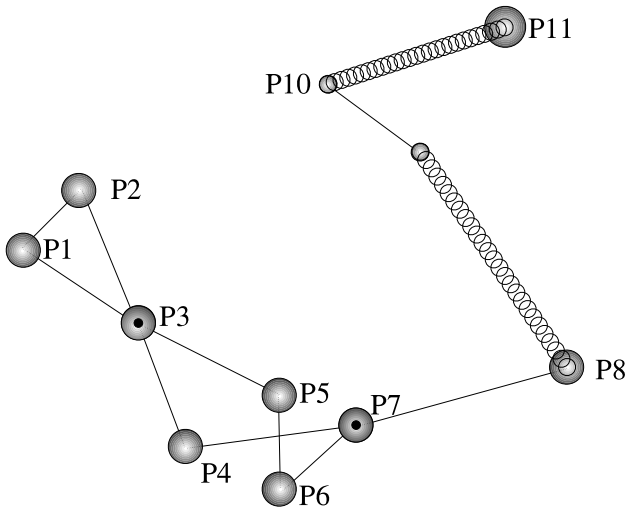


Figure 4.4: An example system of particles

Henceforth, visual clarification of body boundaries is added in forthcoming figures as shown in Figure 4.5. A particle always belongs to a body, even if it is the only particle in it. Particles within a body conform to the mathematics given in §2.4 (p. 43).

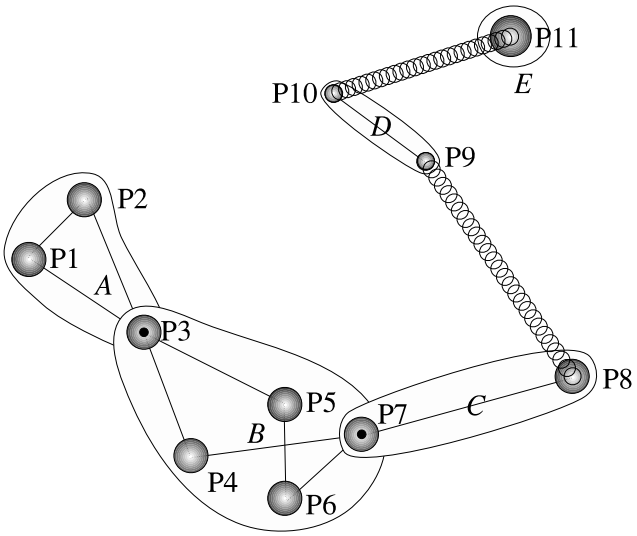


Figure 4.5: Boundaries used to delimit bodies

Now that the conceptual entities in the scene shown in Figure 4.5 have been established, it is possible to consider what needs to be computed to simulate the

motion of such an entity.

It is clear that the bodies A, B, C, D and E are rigid bodies and so their velocity vectors could be computed using the centre of mass as though they were big particles. However, this specific problem is not quite so simple, particles P3 and P7 are hinges so some special computation is required to keep them from falling apart. There is an inter-dependency between bodies A, B and C caused by the forces exerted on them by the hinges. The problem appears to require simultaneous solution. Moreover, between particles P8/P9 and P10/P11 force functions exist which must be taken account of too. What would happen if a force function also existed between P8 and P2? It soon becomes clear that it is not easy to describe the motion of such systems.

4.3.2 Core mathematics

Recall from §2.5 (p. 45) that a numerical integration is required to advance a frame in the simulation. An Euler algorithm was chosen because the trade-off between performance and accuracy was considered to be favourable. For interactive VR purposes the best possible performance is required, at the expense of some loss of accuracy. Given that the average user is unlikely to be able to notice any drift in the solutions and the overall motion remains plausible, the performance issue is of utmost priority. Equation 4.1 is used to compute the linear motion of a body A and is the Newton Euler formulation:

$$\begin{aligned} \mathbf{A}_A &= \mathbf{F}_A M_A^{-1} \\ \mathbf{V}_A &+= \mathbf{A}_A \\ \mathbf{P}_A &+= \mathbf{V}_A \\ \text{or simply: } \mathbf{P}_A &+= (\mathbf{V}_A += \mathbf{F}_A M_A^{-1}) \end{aligned} \tag{4.1}$$

Similarly, Equation 4.2 shows the Newton Euler formulation for rotational motion.

$$\begin{aligned} \overset{\curvearrowright}{\mathbf{A}}_A &= c\mathbf{I}_A^{-1} \sum_{n=1}^{N} \mathbf{f}_{A.n} \wedge (\mathbf{p}_{A.n} - \mathbf{P}_A) \\ \overset{\curvearrowright}{\mathbf{V}}_A &+= \overset{\curvearrowright}{\mathbf{A}}_A \\ \mathbf{p'}_{A.n} &= \mathcal{A}\left(\overset{\curvearrowright}{\mathbf{V}}_A += c\mathbf{I}_A^{-1} \sum_{n=1}^{N} \mathbf{f}_{A.n} \wedge (\mathbf{p}_{A.n} - \mathbf{P}_A)\right).(\mathbf{p}_{A.n} - \mathbf{P}_A) \\ &+ \mathbf{P}_A \end{aligned} \tag{4.2}$$

In both cases integration is approximated using addition. Hence the linear acceleration of the body is calculated from $\mathbf{A} = \mathbf{F}/M$ and subsequently added to the current velocity which is in turn added to the current position of the body.

In principle the mathematics for rotation is the same, but using corresponding quantities. For force we use torque, angular acceleration corresponds to acceleration, angular velocity corresponds to velocity and angle (or direction) is analogous to position.

A convention has been adopted for representing rotational quantities using a curved arrow, as in $\overset{\curvearrowright}{\mathbf{A}}_A$. Rotations take place perpendicular to this vector. To perform a rotation about it, a conversion is made to an equivalent transformation matrix using the method described in §C.1 (p. 209).

The magnitude of the rotation vector $\overset{\curvearrowright}{\mathbf{V}}_A$ is proportional to the rotation angle. These angles typically correspond to rotations (in radians) which are too large, so a constant of proportionality c is introduced to avoid too much rotation in any one frame. Typically a value of c between 0.3 and 0.9 was found to be satisfactory.

Equations 4.1 and 4.2 have been combined into Equation 4.3 resulting in the

equation used to compute the position of every particle per frame.

$$\begin{aligned} \mathbf{p'}_{A.n} &= \mathcal{A}\left(\overset{\frown}{\mathbf{V}}_A += c\mathbf{I}_A^{-1}\sum_{n=1}^{N}\mathbf{f}_{A.n} \wedge (\mathbf{p}_{A.n} - \mathbf{P}_A)\right).(\mathbf{p}_{A.n} - \mathbf{P}_A) \\ &+ (\mathbf{V}_A += \mathbf{F}_A M_A^{-1}) \\ &+ \mathbf{P}_A \end{aligned} \tag{4.3}$$

Most of the terms are common to all particles in a given body and these need not be recalculated for each particle.

In the real world objects experience a resistive force while travelling through a medium, such as air or water, which causes them to lose energy and slow down. This is called *damping* and has to be reflected in physical simulations otherwise they will not behave as we would expect. A damped body experiences a force which acts along $\mathbf{V}$, the velocity of the object.

$$\mathbf{F}_{resistive} = -k\mathbf{V} \tag{4.4}$$

where k is known as the mechanical damping factor, and is dependent on the fluid medium and geometry of the object. A similar end effect was achieved by applying damping at the end of each frame through multiplying velocities by an arbitrary value between zero and one. A high degree of damping corresponds to values close to zero, while negligible damping can be achieved with values such as 0.95 or 0.995.

Now that a particle's equation of motion has been derived, it is possible to use this as a building block for the simulator. A need arises to implement hinges between bodies, so we will discuss how this can be achieved.

Hinging between rigid bodies

Enhancing the simulator to respect the notional 'sharing' of particles between bodies and so provide 'hinging' adds a great deal of complexity to the model described. The primary reason for this is that the particle which is hinged will, by Newton's Third Law, exert forces to try to match all the forces applied to it. The *net* internal forces exerted will be zero. Where there is more than one hinge near another, the forces exerted by each will have dependencies on one another.

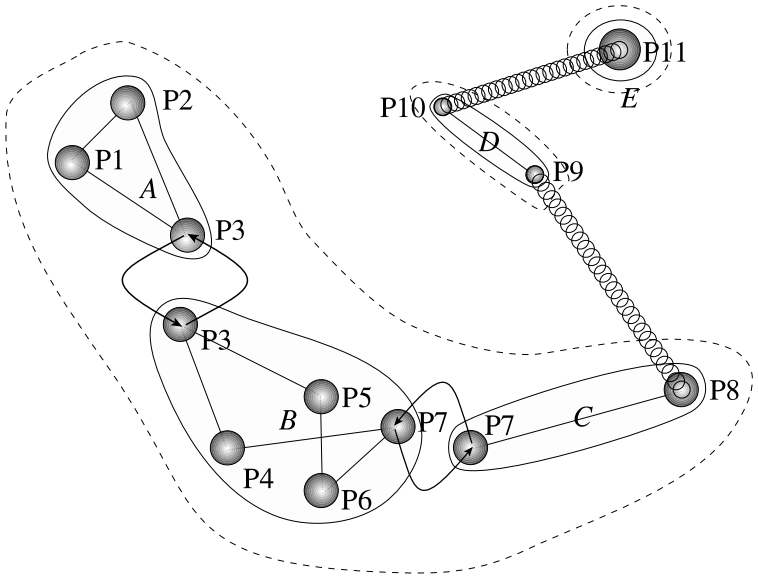


Figure 4.6: Initial state of the system before the constraints are met; notice the curved arrows which indicate that forces are required to bring the bodies together at the hinge

For points in a hinge to remain coincident they must all arrive at the same position on the next frame, as calculated by Equation 4.3. For example, in Figure 4.6 a necessary requirement to construct and maintain the hinge constraint at P3 which joins bodies A and B is that

$$\mathbf{p}'_{A.3} = \mathbf{p}'_{B.3} \text{ which is better expressed as} \tag{4.5}$$

$$\mathbf{p}'_{A.3} - \mathbf{p}'_{B.3} = 0 \tag{4.6}$$

and so after expansion becomes

$$
\begin{aligned}
& \mathcal{A}\left(\hat{\mathbf{V}}_A += c\mathbf{I}_A^{-1}\sum_{n=1}^{N}\mathbf{f}_{A.n}\wedge(\mathbf{p}_{A.n}-\mathbf{P}_A)\right).(\mathbf{p}_{A.3}-\mathbf{P}_A) \qquad (4.7)\\
+\ & (\mathbf{V}_A += \mathbf{F}_A M_A^{-1}) + \mathbf{P}_A\\
-\ & \mathcal{A}\left(\hat{\mathbf{V}}_B += c\mathbf{I}_B^{-1}\sum_{n=1}^{N}\mathbf{f}_{B.n}\wedge(\mathbf{p}_{B.n}-\mathbf{P}_B)\right).(\mathbf{p}_{B.3}-\mathbf{P}_B)\\
-\ & (\mathbf{V}_B += \mathbf{F}_B M_B^{-1}) + \mathbf{P}_B\\
=\ & 0
\end{aligned}
$$

The particles $A.3$ and $B.3$ will exert opposing, but currently unknown, forces upon one another. A similar set of equations can also be derived between the particles $B.7$ and $C.7$. Hence there are two simultaneous equations which hold true for the bodies A, B and C.

The two sets of particles will exert additional forces to pull themselves to the same position. It is these forces which must be calculated before the motion of the bodies can be calculated. In the general case this is too complicated to solve analytically, so the use of a solver such as the one which will be described in §4.3.2 is required.

A number of observations may be made about the number of simultaneous equations which will be derived for a system: each hinge with n particles will contribute $n-1$ simultaneous equations and $n-1$ unknown internal forces. Furthermore, note how no constraint forces are exerted outside of bodies A, B and C and so the two simultaneous equations may be solved in isolation. If there were additional sets of hinges elsewhere in the system, for example another three, then it would be inefficient and more challenging to solve five sets of equations with five unknowns when it is possible to solve separate problems with two and three unknowns.

To support this notation, a novel technique of maintaining data structures was adopted in this work which keeps unrelated sets of bodies in separate 'containers'

known as *articulates*. An 'articulate boundary' has been shown using dotted lines around the bodies A, B, C, body D and body E in Figure 4.6.

Using a solver

A great deal of research has been invested in designing algorithms which can find solutions $x_{1...N}$ to a set of simultaneous equations $F_i(x_1, x_2, \ldots, x_N) = 0$. Well-known FORTRAN numerical solver packages include LINPACK [DBMS79], Lapack [ABB+95] and NAG [NAG]. These libraries include solver algorithms which use general methods but have been specifically optimised for special cases. More recently Diffpack [Lan96] has been developed by the University of Oslo as a C++ library which provides a class hierarchy of solvers.

Press, Teukolsky, Vetterling and Flannery [PTVF92] make the dramatic statement in their book

> There are *no* good, general methods for solving systems of more than one non-linear equation. Furthermore, it is not hard to see why (very likely) there *never will* be any good, general methods.

This quotation is defended with an example in which the two Equations 4.8a and 4.8b need to be solved simultaneously in two dimensions.

$$f(x, y) = 0 \tag{4.8a}$$

$$g(x, y) = 0 \tag{4.8b}$$

The functions f and g have no relationship to each other, so there is nothing special about any common points in terms of either function. Press et al. show that to find all common points requires mapping out the full zero contours of both functions. They also highlight the fact that the zero contours will have an unknown number of disjointed closed curves. How can one guarantee that all

these disjoint regions have been found? They go on to say that for problems in N dimensions, points mutually common to N unrelated zero contour surfaces each of dimension $N-1$ have to be found. Based on this argument they say that additional information about the problem is always required to facilitate root finding. While the situation in general is quite bleak, equations which are continuous, 'well-behaved' and for which the vicinity of the solution is known in advance, solvers can generally find a solution.

There are a number of multidimensional root finding methods suitable for solving simultaneous equations of the type we wish to solve. Some of these methods are more complicated than others, perhaps optimised for particular cases and may be more robust (possibly at the expense of speed). Again a trade-off is involved in deciding upon which particular solver to use. Two general purpose root finding methods which are believed to be suitable are outlined below. The choice between them is arbitrary since they are relatively similar.

Newton-Raphson Method An excellent insight into Newton-Raphson and related methods is presented in [DS83, PTVF92] and paraphrased in Appendix C.4. They deal with its stability and ways to improve its convergence. This method is considered to be the simplest multidimensional root finding method. It gives a very efficient means of root finding if seeded with a suitable initial guess. The best way to visualise how it works is as an extension of the Newton-Raphson algorithm in two dimensions. Instead of the gradient simply being a scalar value, a matrix of (partial derivative) values is used to represent the gradients in each dimension in relation to each of the unknowns. This matrix is known as a *Jacobian* matrix. From this a new set of possible values for the unknowns may be calculated which bring the evaluated equations closer to zero. Naturally the algorithm has a number of advantages and disadvantages.

Advantages

- Quadratically convergent from good starting guesses if Jacobian is non-linear.
- Exact solution in one iteration for an affine F. That is, in cases where a straight line along the gradient passes through a solution, the solver will find the solution on that iteration.

Disadvantages

- Not globally convergent for many problems.
- Can spectacularly fail to converge, indicating that a root may not exist in the vicinity.
- Requires calculation of Jacobian for each iteration.
- Each iteration requires the solution of a system of linear equations that may be singular or ill-conditioned.

Shortcomings in Newton's algorithm may be largely overcome when it is used in conjunction with other techniques such as a *line searching* algorithm.

Broyden's Method is known as a quasi-Newton or secant method. Essentially, the algorithm is analogous to Newton's method but substitutes an approximation to the Jacobian matrix [DS83].

The Newton-Raphson method is used in our simulator because it was found to be the simpler of the two methods outlined above. Broyden's method may perform better than Newton's method in certain circumstances however, it loses that edge when its estimate of the Jacobian becomes too inaccurate.

Each iteration of the solver tries to obtain a better set of unknown input values by inverting the Jacobian matrix and multiplying it by $-F$, a vector that represents how far we are from the solution. This gives a vector, known as the

Newton direction. Travelling in the direction indicated by this vector ideally moves us closer to the solution. At each iteration of the method, the Newton direction should improve the chance of converging upon a solution.

Convergence of Newton's method is improved substantially by incorporating a line searching and backtracking algorithm. It is possible to overshoot the solution by taking a full Newton step, so the Newton direction vector is adaptively scaled to find an optimal distance to travel along the solution space.

4.3.3 Simulator classes

The simulator consists of a number of classes (in the OO sense) which will be described in this section. Before a brief description can be given of them, let us clarify the simulator representation of a particle. To address the need to allow a single particle to be associated with multiple bodies in a hinge, a data structure is required for each member that makes up the particle. Within the simulator, these are known as *points*. One of the points in a hinge particle is arbitrarily chosen to be the *master point*, the role of which is described later in this section.

Point_Local is a data structure containing the attributes which are 'per-point', such as its position, the net force acting upon it and its parent body.

Point_Shared is used to hold 'per-particle' data: primarily particle mass and a pointer to the master point. A particle's colour and other similar attributes may also be held.

Point is a member of a ring which conceptually represents a particle. Each point has its own Point_Local and shares a Point_Shared with all the other members which make up the particle.

Body is a container for points. Most of its attributes are recomputed each frame: total Mass, Inertia, Position of COM, total Force/Torque, body damping

etc. In addition, it contains a list of CBonds described later.

Articulate is a container for bodies. Moreover, it inherits a solve method from the Solver class because it *is* the domain in which the problem is posed.

System is a container for articulates and holds the gravity vector.

Bond is an abstract class which references two points. A virtual method must be overridden describing the force function that it exerts on the points.

CBond *Cosmetic bonds*[2] can be used to display notional bonds which exist within a (rigid) body. A representation of internal bonds in the simulator may provides useful detail for visualisation purposes and will be illustrated in Chapter 6.

Solver is a general purpose solver which uses Newton's method for solving simultaneous equations. A single method is provided which attempts to find a solution. Three methods can be overridden: `number_of_equations` which returns the number of equations to solve, `equations_evaluate` to evaluate the equations, and `equations_jacobian` which can be overridden if the equations can be differentiated analytically.

Miscellaneous mathematics classes are used heavily in the simulator, primarily for matrix and vector calculations. Both position and direction vector classes exist, as the two have different types of associated methods. For example normalising and vector products apply to directions whereas positions may be used in bounding box calculations.

A particle can possess a number of attributes: a mass, position, velocity, orientation, colour and lifetime. This data represents the state of any instance of the particle primitive. The fact that particles may be shared by several bodies

[2]These bonds are referred to as cosmetic because they do not take part in the simulation.

means that some information must be maintained separately for each body that a particle is a member of.

This could be implemented in any number of different ways, but the data structure found to be most appropriate was a ring. All the points in a particle share the data which is 'per-particle', and each has its own local data together with a pointer to the next point.

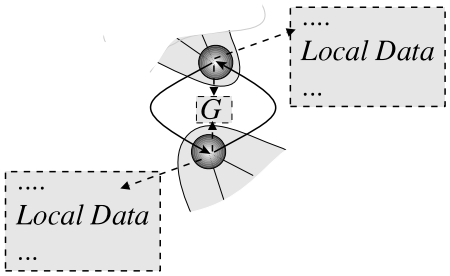


Figure 4.7: Figure showing two points in a ring making up a hinge particle, each with its own local data, and sharing global data (G)

The ring elements can be readily cycled to find all the points which belong to the particle, and the membership to bodies may be examined in a point's local data structure. The shared data retains a count of the number of points which reference it, and is released when the last point disappears.

One point in the ring is arbitrarily nominated the master point, and is given a special significance when formulating equations to be passed to the solver. The reason for this is that the constraint equations are generated by traversing the ring elements. Each point is visited, together with the point adjacent to it and an equation is built by subtracting the positions of these two points as was discussed in §4.3.2. Since only $n-1$ equations are required for an n point hinge, an equation is not constructed when the current point is the master. It is clearly necessary to ensure that one, and only one, point in the ring is the master. It does no harm to consider points which are not hinged to be in a ring by themselves. After all,

they will generate $1 - 1 = 0$ equations.

While the techniques that have been discussed allow for simulation of a system to take place, it is still necessary to examine the internal representation of the scene and more importantly how it can be manipulated before or during the simulation.

4.3.4 Complexity management and interaction

Merely simulating the motion of objects in a VE is interesting, but not nearly as useful as being able to interact with them, modify the model and watch the simulation take form with different criteria. The difficulty with allowing any type of user manipulation of the physical model used to represent an object, is that it is impossible to predict what the user may wish to do. Potentially, a user might want to move the object around, deform it, break it or attach something else to it. Thus much effort was invested in developing powerful, robust scene manipulation and restructuring routines. Furthermore the dynamic restructuring routines can be exploited to manage the complexity of simulations. This benefit will be illustrated in the case studies in the following chapter.

Scenes are represented internally as a tiered (upside-down) tree with a system object as its root, followed by articulates, bodies and finally points. We can represent Figure 4.6 (p. 103) as the tree shown in Figure 4.9 via an intermediate step shown in Figure 4.8. It is critical for correct solver behaviour that the articulate containers have the right bodies beneath them. If there are too many then the solver will not optimally converge towards a solution, too few and the scene will not behave correctly.

Manipulation of the tree is therefore considered sufficiently complex to necessitate the creation of high level methods to perform the manipulation. These routines are described here, while some of the subtle details are discussed in the

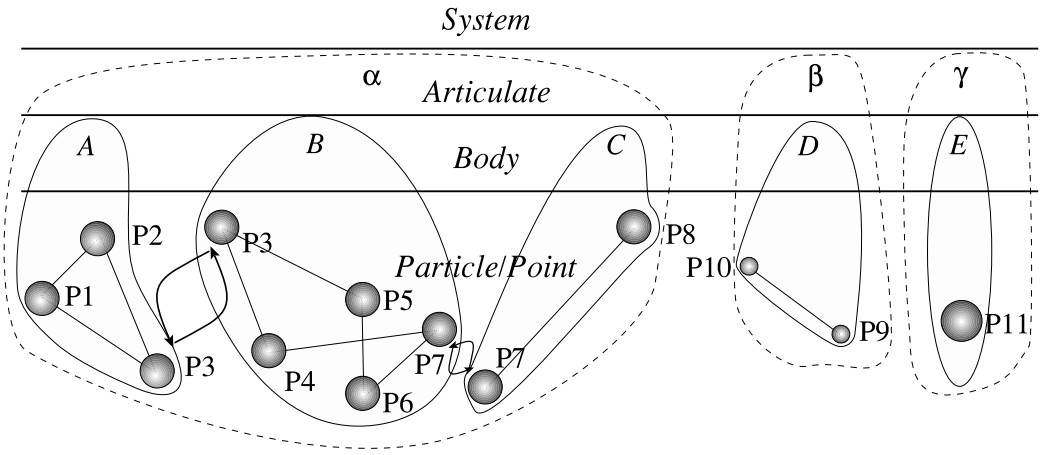


Figure 4.8: Rearrangement of Figure 4.6 into start of hierarchy

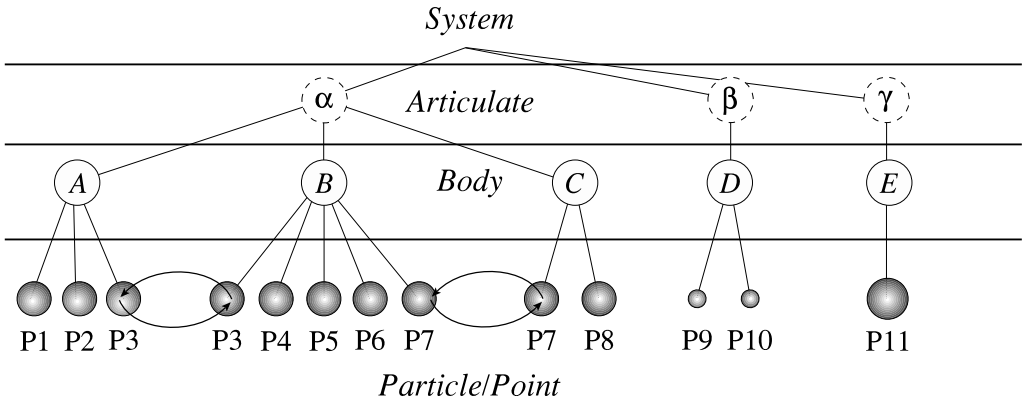


Figure 4.9: Rearrangement of Figure 4.8 into hierarchy

Method	Applies to	Action
Public		
`Index`	system, body	Since systems and bodies (and articulates) are effectively containers, they provide a method for indexing member objects
`Create`	system, body, particle	This method creates a new instance of an object. In the case of bodies and particles it is necessary to register them with a parent
`combine`	two particles or bodies	Combining particles has been given different behaviours depending on the relationship between the two particles, and is expanded upon in Table 4.2
`separate`	two particles	Separating particle has been given different behaviours depending on the relationship between the two particles, as shown in Table 4.3
`Destroy`	system, body, particle	This method will remove an object and its children
Private		
`validate`	System	This method is used internally to traverse the object tree to check for inconsistencies. It is not intended to be called in the production version of the simulator, but is useful for checking for programming errors as methods are developed.

Table 4.1: Methods for manipulating scenes

next section.

Scene manipulation is carried out through the use of *dynamic restructuring* methods, shown in Table 4.1, which allow the scene graph to be modified at run time. From the complete list of methods that may be applied to objects in a scene, the two most important routines can be used to dynamically combine and separate points. These routines take two points as their arguments, but their behaviour differs depending on their relationship. There are four possible classifications of relationship between a pair of points.

Scenario 1 Both points passed refer to the same point.

Scenario 2 They belong to the same body.

Scenario 3 The points are members of the same (hinged) particle.

Scenario 4 The pair of points are from two different bodies and are unrelated.

Tables 4.2 and 4.3 show the behaviour of these routines for each scenario. Notice that according to scenarios $2, 3$ and 4 in Table 4.2 one of the points involved in the `combine` is always destroyed. This is a potential source of problems because the user may have C++ references to particles which will no longer exist. This problem can be solved in C++ but we catch and handle it through the scripting language instead.

Examples of successive `combine`s and `separate`s are shown in Figures 4.11 and 4.12. Each subfigure follows from the previous example. This behaviour is very powerful because it enables these general routines to always be called in exactly the same way, but achieving very different effects. `combine` in this sense has the effect of making the relationship between the two points closer, `separate` has the opposite effect, as shown in Figure 4.10.

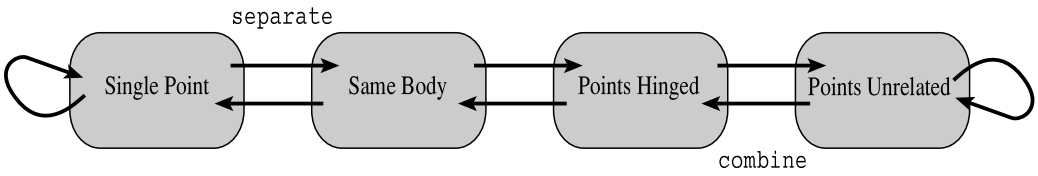


Figure 4.10: Effect of successive combines and separates on points

An application of these routines is illustrated here by considering an example scene which simulates a shoal of fish swimming in the sea. Suddenly, some of the fish may decide to nibble on some seaweed. A subset of those feeding fish may actually break off a segment and swim off with it trailing from their mouths. Such a simulation has been implemented using Iota and is reported in the following chapter.

This could be achieved by simulating the shoal of fish as rigid bodies which attract each other from the front and repel each other at the tail. The user may

Scenario	Behaviour	Figure
Points are in fact the same	No action taken.	N/A
The two points are found to belong to the same body	One of them (the latter) will be removed. No further manipulation of bodies or articulates is required.	P_{B6}/P_{B5}: 4.11(a)
Both points belong to the same particle	One of them becomes redundant and is removed, and the two parent bodies are subsequently combined (no articulate manipulation is ever necessary). The end effect is to make a hinge rigid. Ordinarily the two points combined will have the same position, so in practice it is not possible to observe one of the points being removed.	P_{B7}/P_{C7}: 4.11(b)
The two points are from different bodies	This results in a hinge where the two points will now take on the attributes belonging to the former point.	P_{D9}/P_{B8}: 4.11(c)

Table 4.2: Scenarios for the `combine` method

Scenario	Behaviour	Figure
Points are in fact the same	A new point is created in the same body at the same position.	No change [4.9]
The two points are found to belong to the same body	One of the points will be removed and recreated in its own (new) body and hinged.	P_{D9}/P_{D10}: 4.12(a)
Both points belong to the same particle	One of the points will be detached from the ring. It is highly likely that a new articulate must be created to accommodate it	P_{B3}/P_{A3}: 4.12(b)
The two points are from different bodies	No action required	N/A

Table 4.3: Scenarios for the `separate` method

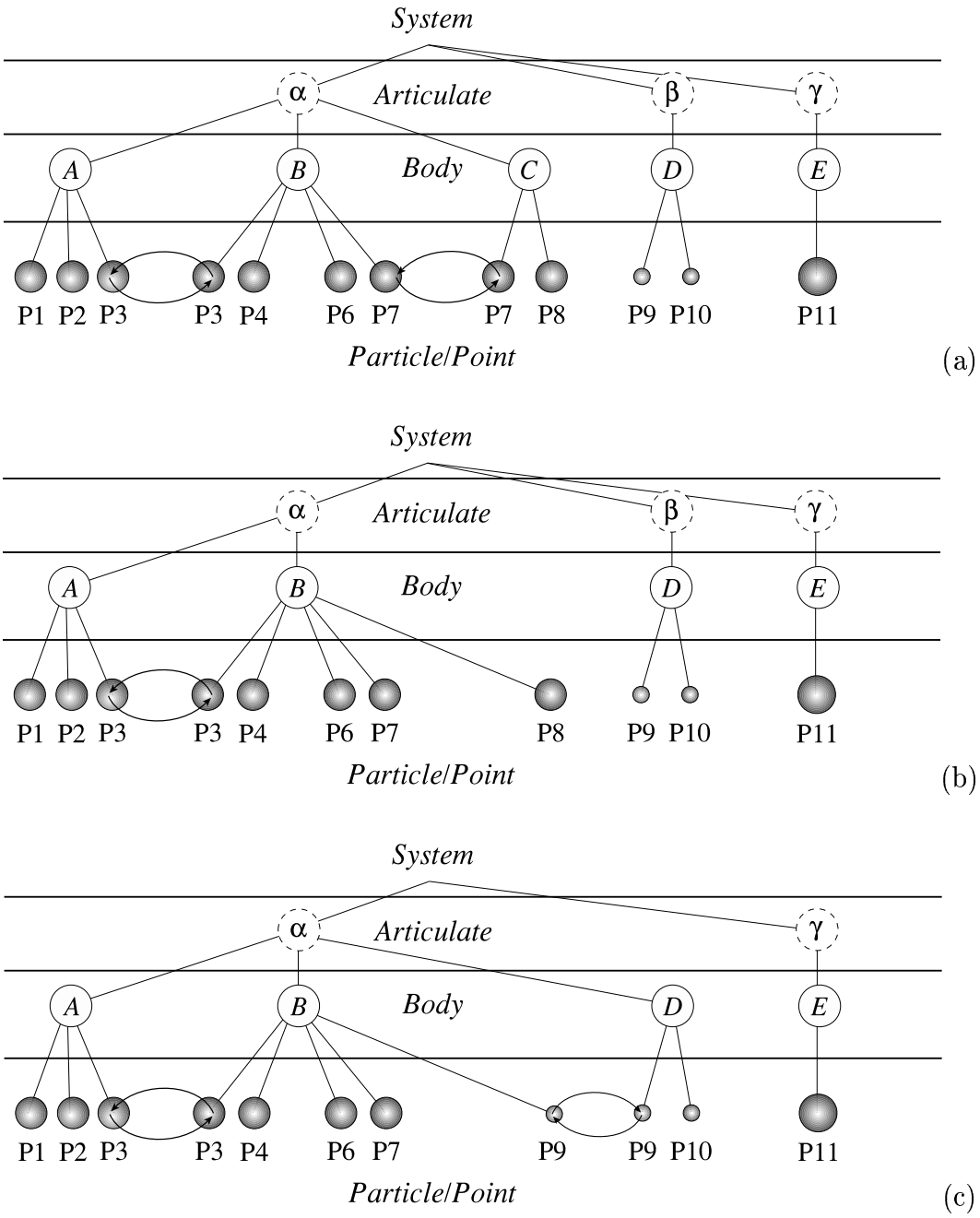


Figure 4.11: Example of successive combines applied to a scene, Figure 4.9 shows the original scene graph

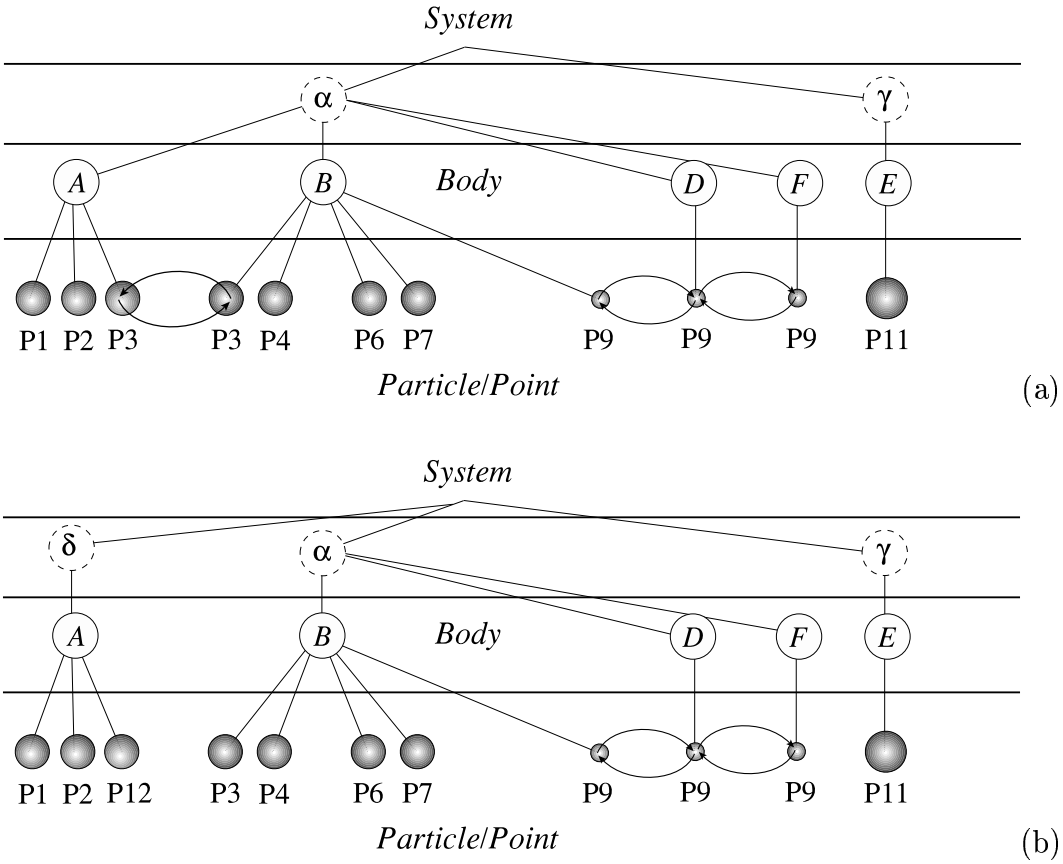


Figure 4.12: Example of successive separates applied to a scene and follows on from the final scene graph shown in Figure 4.11

control a driver fish which leads the shoal. Articulating chains may be used to simulate the seaweed. Force functions can be specified to attract a set of fish to the food and a combine call could be applied to attach them. A temporal event could be activated to separate certain hinges in the chains and finally, the fish possessing such segments of seaweed may be programmed to swim off on a set path at the next frame.

One major feature of these routines is that they hide the articulate layer from the user. This is because articulates exist only for the solver's benefit, and their explicit manipulation would be a hindrance to the user.

4.3.5 Advanced implementation details

Whilst the combine and separate routines provide a natural way of manipulating scenes, their implementation is far from trivial. To illustrate, this section will focus on one of the separate scenarios described in the previous section. We will study two hinged points which are members of the same particle being separated (scenario 3), and belong to the structure shown in Figure 4.13.

Depending on the topology of the shape on which the separate takes place, it is possible that this operation will result in the structure remaining a complete entity, as in the example given. This happens because topologically the two points existed in a cyclic configuration so after a cut, a single structure still exists.

Under different circumstances two entities may in fact result and so two separate articulates would have to be created. The challenge is to decide whether one or two articulates will result from a `separate` call. Essentially, this requires algorithms capable of determining the topology of the entity being manipulated.

Two algorithms are implemented to achieve this: A call is made to Algorithm 1 which marks points that are in the same articulate as the point is passed. Algorithm 2 can subsequently be applied to find bodies which might appropriately need to be moved if the second point is no longer in the current articulate.

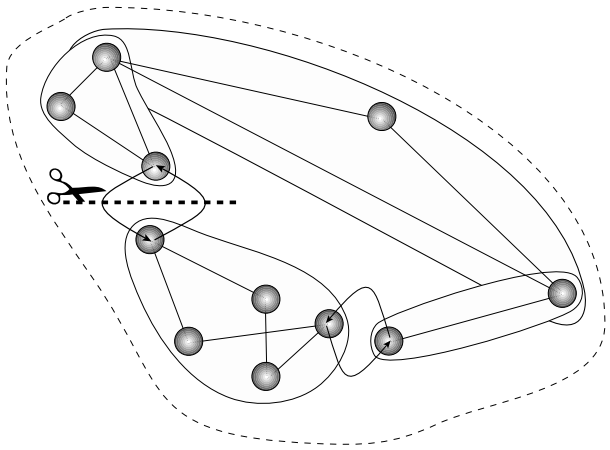


Figure 4.13: Will two articulates result from separating the two points?

Algorithm 1 is called on the parent body of one of the points being separated. It works by walking through the data structure ticking off an instance variable per point, in the region of interest. Basically, this means that the algorithm flood fills until it reaches a loose end or a point which has already been ticked off.

Algorithm 1 Algorithm to perform a recursive flood fill on all the points in an articulate (`Body::validate_flood`)

Require: `validate_flag`

```
1: for all points in body do
2:    Flag point with validate_flag
3:    for all points in particle ring do
4:       if point not flagged with validate_flag then
5:          Flag point with validate_flag {This is an important step to avoid
            flooding coming back this way}
6:          Make recursive call to parent body's
            validate_flood (validate_flag)
7:       end if
8:    end for
9: end for
```

Algorithm 2 performs the difficult task of redirecting the appropriate pointers to refer to the correct parent articulate. It uses a similar flood fill algorithm to identify which points have to be moved to the new articulate.

Algorithm 2 Algorithm to move separated bodies to a new parent articulate (`Body::repoint_flood`)

Require: `validate_flag new_articulate`

```
1: Unsubscribe this body from its parent articulate
2: Subscribe it to new_articulate
3: for all points belonging to this body do
4:    Mark current point visited
5:    for all points in point's ring do
6:       if point is unvisited and is flagged as being destined for new articulate
         then
7:          Mark this point as visited
8:          Make recursive call to
            this_point->repoint_flood (validate_flag, new_articulate)
9:       end if
10:   end for
11: end for
```

Another issue which should be noted refers to the implementation of bonds between particles or bodies. Although the simulator implementation must be able to access the force acting on particles, to retain a high degree of customisability it is beneficial to avoid having to implement each bond type within the simulator itself. This is achieved using a generic bond class which contains a handle to a function responsible for calculating the force exerted by the bond.

4.3.6 Solver convergence and verification

Since general purpose solvers still continue to suffer from convergence problems [PTVF92] it is necessary to describe the actions taken when the solver is unable to return a solution. Furthermore, it is important to verify the results returned by Iota's solver in order to gain some confidence in the simulations.

There are two well-known situations in which general purpose solvers can fail to return a satisfactory solution to the constraint problem posed. These are known as singularities and ill conditioning, and are discussed in turn.

A determinant of zero is indicative of a *singular* matrix which cannot be inverted. A singular Jacobian can arise typically when links in an articulated body align themselves or are fully extended as shown in Figure 4.14. In this particular example, notice how the two COM's and the hinge are coaligned. Rotating either

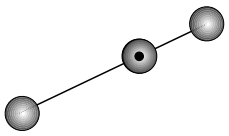


Figure 4.14: A fully extended articulated body

of the links will produce the same motion at the end of the articulate, so one degree of freedom has been lost. The determinant may be used to identify convergence problems caused by a singular Jacobian. Solutions to this problem typically involve either never allowing the articulate to become fully extended, or using

a different solver in the region of the singularity. Both of these solutions were investigated but they did not improve convergence significantly enough to prove satisfactory.

The second problem of ill conditioned equations arises under circumstances where the possible solution leads to a near singularity. The distance to the solution may in fact be very small, but when the configuration of the articulate is close to the singularity the determinant of the Jacobian is near zero. This leads to very large and erratic Newton direction vectors which are not useful. This problem is often solved by using a numerical method called a *Singular Value Decomposition* which breaks up the Jacobian into three matrices. One of these matrices will give insight into which terms of the Jacobian cause problems. These offending values should be zeroed so that their contribution is nullified rather than producing large inappropriate values when their reciprocal is taken. While this tactic did improve convergence it simply did not prove to be a robust enough solution.

The most successful ad-hoc methods to counteract convergence problems often involve perturbing terms within the Jacobian if singularities or near singularities are encountered [PTVF92]. Whilst these techniques greatly improved the success rate for the solver, even a small number of failures can cause disturbing artifacts. One example that illustrates this is a damped pendulum which on each swing reaches a height less than that on the previous swing. On failing to solve, an inappropriate force is provided by the solver which suddenly causes the pendulum to gain energy and swing in an erratic manner.

The main contributing factor to failure to converge was found to be caused by the rotational component applied to bodies. This is consistent with the discussion above on singularities. Simply reducing the rotational contribution in instances where the solver failed to converge, was found to dramatically improve the success rate. This is guaranteed to find a solution, since in the worst case it will eventually

reduce to the linear case which is always solved.

Experiments were conducted to obtain an idea of the point at which the complexity of a constraint equation would cause the solver to fail to converge. Typically chains of eight links or more started to exhibit occasional problems. To put this into context, the solver is inverting and multiplying matrices with approximately 24x24 cells (three unknowns for each link). As the solver approaches a solution to an equation involving such large matrices, any small numerical inaccuracies will begin to accumulate over the hundreds of multiplications and additions and introduce errors into the near zero Newton direction.

Now that solver convergence issues have been addressed, it is appropriate to compare and verify the solutions returned by Iota's solver with the University of Oslo's Diffpack [Lan96] solver[3]. To carry out comparisons and verification a Diffpack test harness was implemented to solve our formulation of the equations of motion for the case of a six link chain of length 50 units.

Our solver's ability to meet a point to nail constraint was tested in an experiment by scripting the top of the chain to meet constraints at regular intervals along a Lissajous figure. The nail is moved to co-ordinates specified on the Lissajous and the solver is challenged with meeting the new point to nail constraint. The results achieved using both Iota and Diffpack's solvers were plotted on the graph shown in Figure 4.15. In all cases Iota's solver matches Diffpack position for position and this verifies that Iota's solver works.

A further set of experiments were conducted in which the six link chain was pinned at one end and allowed to fall under gravity from a horizontal position (cf. chain case study in Chapter 5). Damping was minimised such that the chain would swing for approximately two and a half thousand time intervals. The position of the end point of the chain was studied to see how much variation typically arises between Diffpack and Iota simulations.

[3]A version of Diffpack is available free of charge for non commercial use

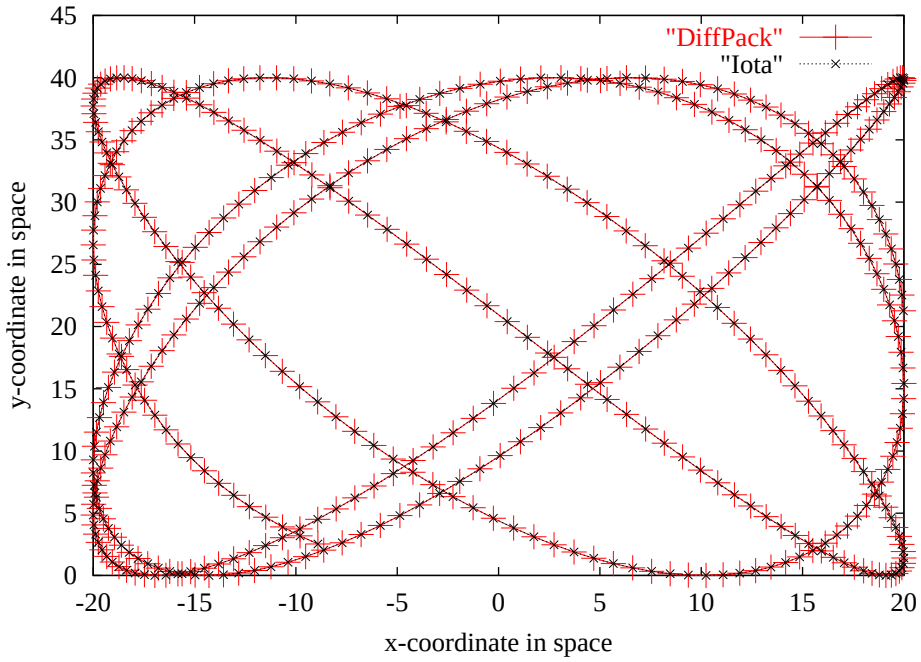


Figure 4.15: A graph showing a position for position match between Iota and Diffpack when meeting a point to nail constraint

It was found that in general the solutions calculated using both solvers for the given scenario were normally to within four significant figures. Approximately twenty tests were conducted and the average displacement between each chains end position was plotted against the frame number (or time interval) and this is shown in Figure 4.16. Notice that the largest displacement, 0.00035, is unlikely to be distinguishable on-screen. The fall-off in the graph is due to energy being lost as a result of damping during the simulation.

This is to be expected, because the only real difference between the solvers is the nature of the fall-back algorithms they use when they fail to solve. Floating point inaccuracies between the two solutions will however accumulate over a period of time, causing drift. It is well known that this drift between the two sets of solutions caused by rounding errors can have dramatic implications, and an example of this is shown in Figure 4.17. In this extremely rare case the end of the chain

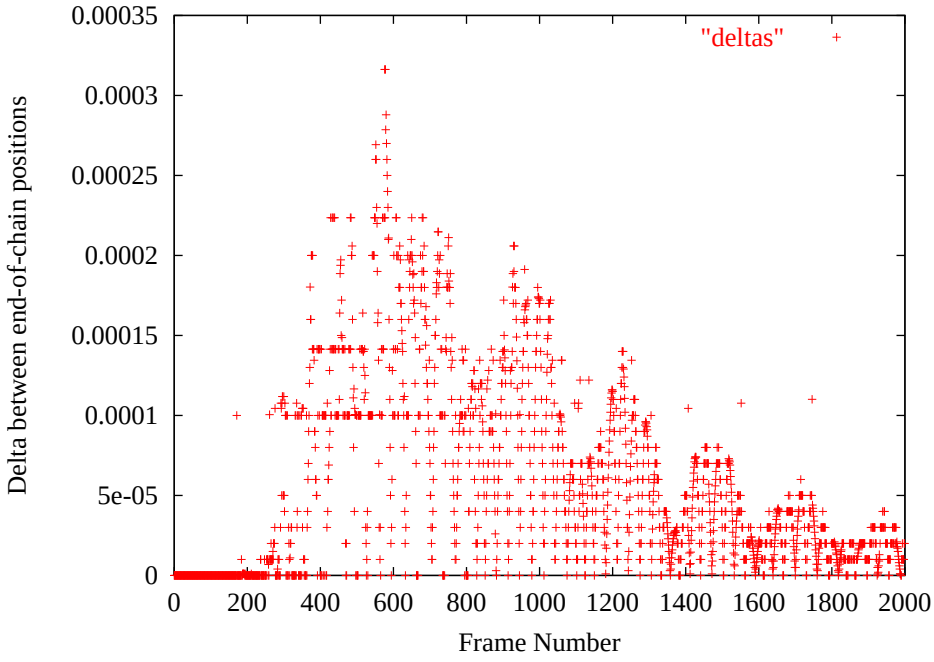


Figure 4.16: This graph shows the displacement in end positions of a chain using Iota and Diffpack's solvers

was seen to tip over in the Iota simulation, but not the Diffpack one. From this event onwards (after around 400 frames) the simulations bore little resemblance to each other; this phenomenon is predicted by theories of chaotic behaviour, in which minor numerical differences may lead to divergent behaviours.

In fact, underlying the Newtonian behaviour of the simulations are a number of other metrics which if plotted are chaotic, such as the number of iterations taken to find a root for a given equations. Even the slightest deviation of the requirements have a significant and unpredictable outcome. Even the simplest of root finding problems equations can exhibit chaotic behaviour when solved using the Newton Raphson method. In the subject of mathematics, a number of studies of Newton Raphson in relation to chaos theory can be found [Gle88]. A common root finding problem often used to illustrate this is the equation $x^3 - 1 = 0$ in the complex plane. If the number of iterations required to find a root for different

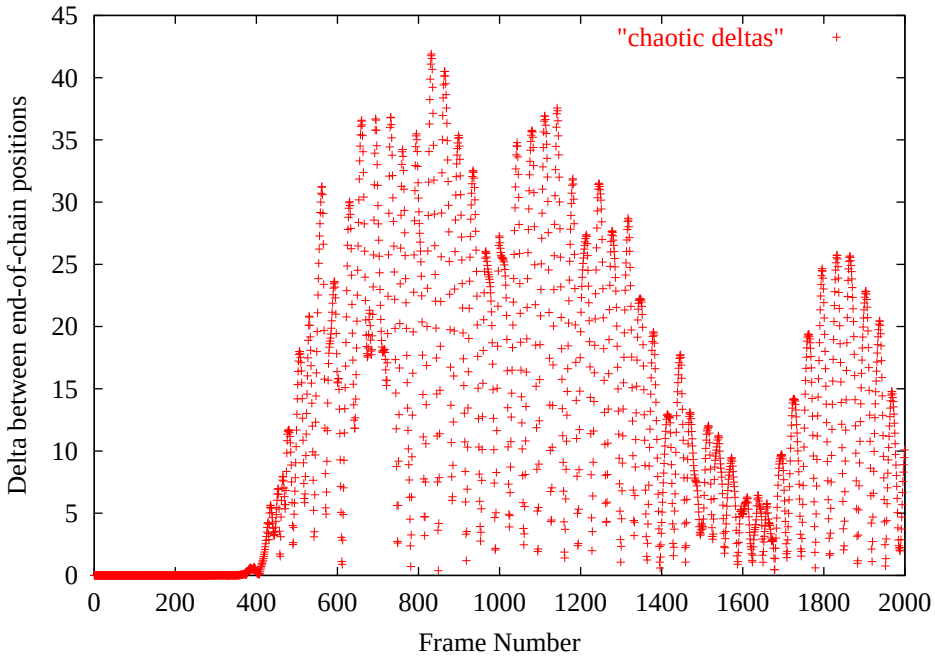


Figure 4.17: A graph showing a case in which Iota and Diffpack's simulations diverge significantly

complex numbers is plotted as a colour an interesting fractal pattern emerges; an example of this is shown in Figure 4.18.

In this case there are three solutions to the problem and these are located in three regions shaded dark blue. The number of iterations required to solve the problem is mapped onto colours ranging from dark blue to red. Dark blue regions represent solutions which require one iteration and red ones represent large numbers of iterations. Notice that at the boundary between the three solutions instabilities occur. Travelling a short distance in these vicinities will result in convergence of the method to any of the three possible solutions in radically different numbers of iterations. Since the simplest of equations exhibits such complex behaviour it is not unexpected for our formulation of the equation of motion (Equation 4.3) to also exhibit chaotic instabilities in its solution space. Given that a number of researchers use a Newton Raphson method [Fau99] (or

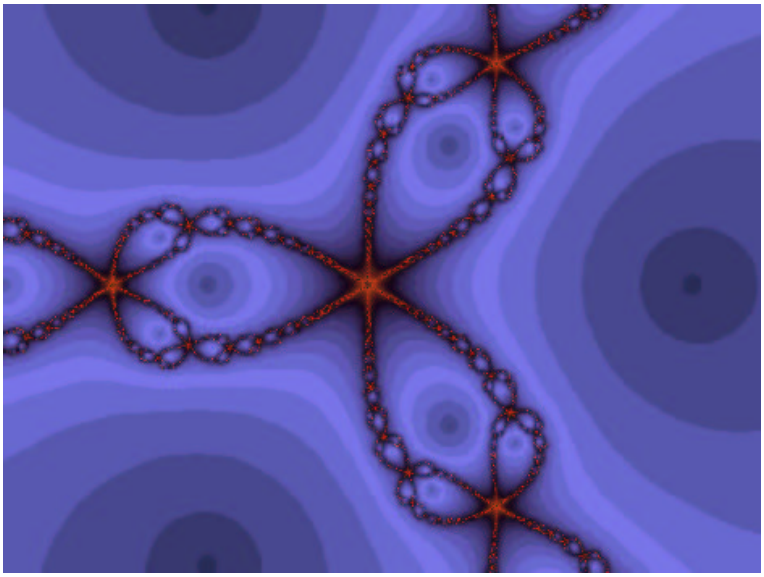


Figure 4.18: Chaotic behaviour exhibited by Newton Raphson for a simple root finding problem

a slight variation) for solving systems of non-linear simultaneous equations for motion computation it is a reasonable method to use. However, two solvers are unlikely to produce exactly the same simulations; there will always be some divergence between solutions predominantly due to floating point inaccuracies which can lead to chaotic behaviour.

It is true that in cases where Iota's solver does not converge we use an ad-hoc method after attempting a singular value decomposition. In this situation we are introducing some error in the simulation and the results from then onwards will diverge when compared with results from a solver which may resort to a different method. However, to put this into context, this scenario may only arise rarely within a given simulation (perhaps approximately once every 10,000 time steps) and there is no guarantee that a different solver will be able to return a solution. The experiments and discussions above show that the solver is generally reliable, but in rare circumstances where there are many constraints to be met within the same articulate it can fail to converge.

4.3.7 Summary of the simulator engine

We have described the underlying model used together with the core mathematics implemented. An insight into the implementation of the simulator with particular emphasis on dynamic restructuring routines, and the solver used to calculate unknown forces has been given. Choices were made to compromise on fidelity of simulations in favour of performance, these specifically involved using a Newton Euler formulation of the equations of motion together with a projection method for constraint satisfaction, and encouraging the use of dynamic restructuring in order to manage the complexity of models to simulate. The idea of using articulates to simplify the problem of solving the motion of articulated rigid bodies is introduced and described. In particular note that articulates can be used to identify independent computations in a parallel implementation. Furthermore the modular design of the simulator allows it to be incorporated with some ease into existing systems which require the ability to perform dynamic simulations. The solutions returned by the solver compared favourably with solutions computed using Diffpack. Finally Algorithm 3 draws together the various parts of the simulator algorithm.

4.4 VR component

Before discussing the reasons behind using MAVERIK as the VR component it is important to answer the question: what is MAVERIK? This can be summed up by the following quotation [HKG+98].

> "MAVERIK is a toolkit for building Virtual Reality applications. There are numerous other "VR tools" around, ranging from very low level libraries of functions for drawing three-dimensional graphics

Algorithm 3 Algorithm for the simulator

```
1: {Optional scene modification stage}
2: for all particles do
3:    Initialise net force on particle to default force {The default force will probably be a downward gravitational force but there are other interesting possibilities}
4: end for
5: for all bonds in system do
6:    Calculate force exerted by bond, and apply to points at each end
7: end for
8: for all articulates in system do
9:    for all bodies in current articulate do
10:      Compute body Inertia (Equation 2.28), Inertia^-1, Mass (Equation 2.22), Mass^-1, Centre of Mass (Equation 2.14), Net Force (Equation 2.23) and Torque (summed Equation 2.7)
11:      Count Unknown Forces and form an array of points in hinges
12:   end for
13:   Populate array of unknown forces
14:   Call Solver
15:   for all bodies in current articulate do
16:      for all points in current body do
17:         Calculate point position from body's linear and angular velocity {Body linear Acceleration, Angular Acceleration, linear Velocity and Angular Velocity were calculated by the Solver, as indeed have the hinge positions which can be skipped}
18:      end for
19:      Apply damping to body Velocity
20:   end for
21: end for
```

> and interacting with peripherals, to fully-blown "systems" that describe virtual environments in much higher level terms. MAVERIK lies somewhere in between these two extremes, and provides high performance, customisable low level functionality within a higher level framework"

MAVERIK is essentially a C library which can be used to construct single user VR applications, and is implemented in a broadly OO fashion. This library is broken down into two main components, the kernel and a set of support modules. The kernel provides low level foundation routines (such as performing matrix transformations) to modules and dispatches service requests to them. The support modules provide routines for efficient input/output and display management.

Before we can explain the motivation for using MAVERIK it is necessary to describe the essential components of a VR system. The debate regarding what is and is not VR still continues on [Vin98], but there is general agreement that a VR system should feature, to varying degrees, the following:

- Navigation — a means of travelling from position a to position b in the three dimensional virtual space.

- Interaction — enabling the user to affect the model of the virtual world which they inhabit, for example selecting an object and copying it.

- Immersion and presence — recall that this was briefly mentioned in Chapter 1. Immersion is not easily quantifiable, all that can be said is that it contributes to the sense of presence. Immersion and presence are inextricably linked and currently various devices contribute to their enhancement. The sense of presence can be increased by maximising user participation in the environment and there is much scope for advances in software which facilitates this.

The motivation for using MAVERIK is largely twofold: firstly, it provides us with the functionality which enables our environments to fit within the above definition of a VR system. Secondly, all the routines required to navigate, interact and drive VR hardware comprise a large amount of code and much effort has been expended by the MAVERIK developers to supply an efficient resource. Moreover, MAVERIK is developed locally, available as source code and is free under the GNU General Public License (GPL). We do not claim however that MAVERIK is the only VR system which can be used as the VR component in the Iota framework, merely that it is convenient for our purposes. Now it is possible to describe the architecture of the component together with the implications this has in terms of supporting MAVERIK functionality through Perl.

4.4.1 The MAVERIK architecture

MAVERIK is implemented in a modular fashion such that the kernel and support modules are loosely coupled, except in the case of the callbacks and object modules which are more closely related. This can be seen in Figure 4.19, which shows how the support modules relate to the kernel and each other. The benefit of this approach is that developers may add new support modules without changing any of the existing code.

There is a great deal of co-operation between the kernel and support modules, but generally not between modules themselves.

The kernel makes few assumptions about the internal representation of the virtual environment which may vary considerably between different applications. Instead it works with abstract objects and classes of objects allowing it to maintain independence.

Support modules are required to register themselves with the kernel via an initialise routine, and subsequently register specific objects with the kernel for it

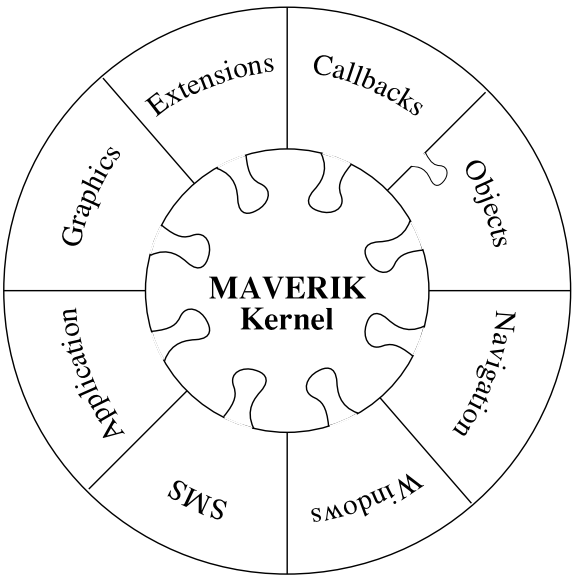


Figure 4.19: The MAVERIK architecture

to manage. In turn, the kernel will request services from external modules such as to render the object or calculate its bounding box. Extensive use is made of the callback mechanism so an overview is provided in Appendix B.

The data structure shown in Figure 4.20 is used to implement abstract objects at the heart of the kernel. An object is a combination of data specific to that object, and methods which are derived from the class to which it belongs. The object data can take any form as far as the kernel is concerned, so object methods must be able to convert the data into a type which is meaningful. The methods for a class are contained in a two dimensional array of pointers to functions indexed by 'window id' and 'callback id'. The values of both these handles are allocated by the kernel as modules are initialised. There is no distinction between system callbacks and user defined callbacks.

Object and Callbacks modules provide default objects and their appropriate callbacks for a large number of common primitives, polylines, textured primitives, text objects, and composite objects. A small change was made to the MAVERIK

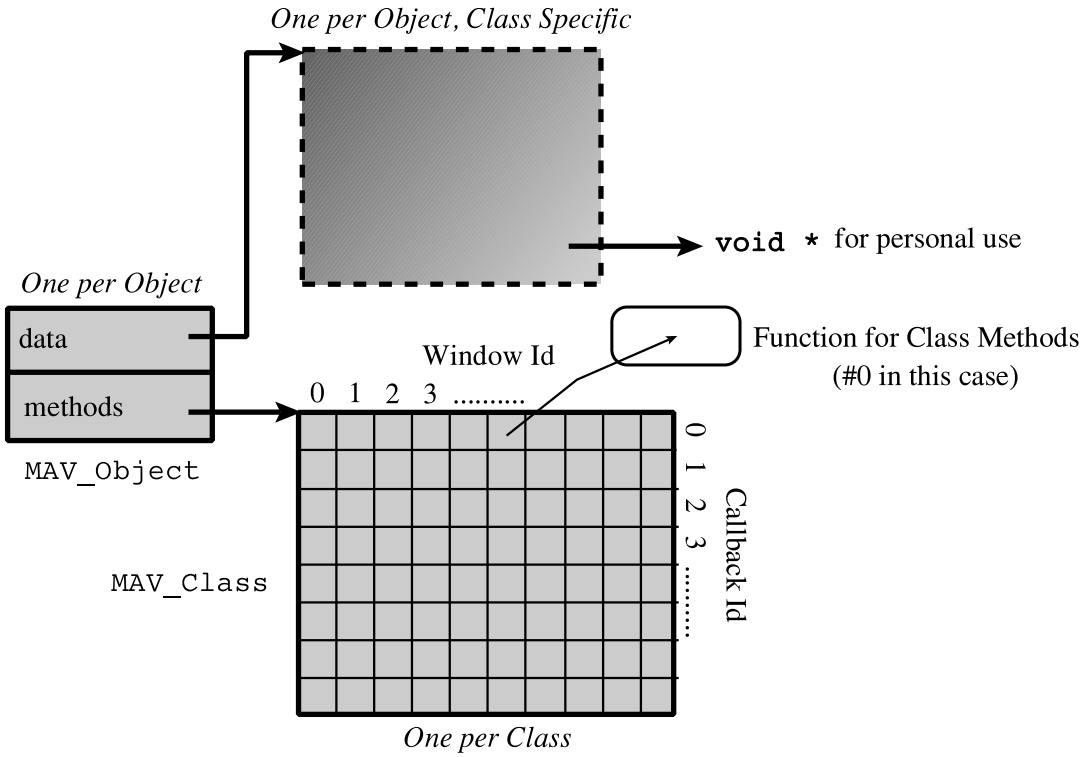


Figure 4.20: The object data structure in MAVERIK

source code to support callbacks from MAVERIK into Perl (supplementary information on callbacks into C and Perl can be found in Appendix B). This allows a high degree of customisation to be made in the user's scripts.

Finally note that this architecture is designed such that data describing the environment is assumed to exist outside the MAVERIK kernel. Ideally the applications developer is encouraged to extend MAVERIK by tightly coupling new software via the use of the void pointer provided in the object data structure. This is an inconvenient consequence of using C to implement an object oriented system. However for the purpose of integrating with the simulator we made a choice not to follow this through, as we prefer to have our simulator data exist

solely within Perl. One of the main reasons for this approach is that Perl objects are higher level than C structures, notably because they are associated with types. While C and C++ are strongly typed, this is only a compile time concept, so no type checking can occur at runtime.

Although it is easy to convert Perl objects to equivalent C ones simply by discarding some information, conversions the other way require assumptions to be made, likened to casting a void pointer to one of a defined data type. Currently Perl is only called from C when callbacks to Perl are invoked. A problem therefore arises if Perl is under the control of MAVERIK because any object passed from MAVERIK require manipulation into the correct type. Furthermore, generic objects represented by `void *` would need type information which is simply not available from C at runtime. This reinforces the idea that it is simpler to control low level code from a high level language than vice versa and complements the idea of script driven VR.

4.4.2 Summary of the VR component

MAVERIK was chosen as the VR component because it provides the requisite VR functionality as is available locally as source code. Further to supporting MAVERIK functionality through Perl some modifications were made to MAVERIK to be able to provide callbacks into Perl. This mechanism enables a high degree of customisations to be made from within the user's scripts.

4.5 Custom modules

Custom modules can be implemented as Perl modules and included in the user scripts. This functionality is in many ways similar to C include files but more

powerful because of the high level routines available in Perl. Providing a placeholder for user customisations in the design of the framework is advantageous because it enables customisations to be made in a structured fashion. If it is possible to rapidly prototype customisations in a simple and easy language then there is little need for a developer to resort to editing low level code.

In the Iota prototype implementation two custom modules called *builder* and *shapelib* are provided. The builder module constructs, using the internal rich common data format, a selection of different types of models with a default graphical representation; for example chains, rings and cuboids. By default, particles are rendered as spheres, rigid bodies as cuboids or cylinders and flexible connections are rendered as ellipsoids. Developers can create models in the rich common data format. Alternatively the shapelib module could be used to import and convert models from other sources. Currently, shapelib contains modules for importing molecular and polyhedra data.

To provide a high degree of flexibility and customisability, the builder routines are implemented using callbacks which have suitable defaults. A good example of this is how visually different a chain can be made by overriding the representation of a link: it can be made to look like a rope or a chain with interlocked links, as will be shown later in §5.1.2 (p. 143).

Model construction in general is significantly simplified through the use of the builder and shapelib modules and default graphical representations. Further custom modules can easily be added at this level by a developer to support the types of models that may be used in a particular application.

Another facility provided in the custom modules is a routine to compute an initial transformation matrix for an Iota object. This is necessary because the simulator stores bodies made up of points which exist within world space whereas in MAVERIK shapes exist in a local co-ordinate system. Consequently, MAVERIK

shapes are centred at the origin of their own local co-ordinate system, are of unit size and have a predetermined orientation when created. A transformation matrix is required to size, position and map the shape into world space. Since we wish to render bodies as MAVERIK shapes, some effort has to be invested into calculating initial matrices for them and keeping them synchronised with the simulator description. The initial transformation matrix is calculated using Algorithm 8 in Appendix A to seed the newly created shape.

This routine is supported as a method of bonds and allows them to be queried for a transformation matrix having the benefit that any desired MAVERIK shape can be associated with a simulator bond. Moreover, the method allows shapes to be resized by any chosen scalar values in the x and y dimensions. This achieves a richness of visual effects as the graphical representation of a bond can be much more complicated then the underlying model implies.

4.5.1 Summary of custom modules

Custom modules provide a mechanism for added user functionality in a structured fashion. In other words there exists an informal protocol dictating where user customisations should generally be made. The Iota prototype currently contains two major custom modules designed to simplify construction and rendering of models. Furthermore, an additional customisation for computing an initial transformation matrix to simplify initial placement of Iota objects is also supported. Due to the high level functionality inherent in Perl, custom models are simple to implement by developers familiar with the language, but this holds true for any equivalent scripting language.

4.6 User scripts

The Iota API is intended to be natural, object oriented and easy to use. In the Iota philosophy a developer will implement customised models suitable for an application domain and then build the application in a high level scripting language. This offers the benefits of rapid prototyping, access to a complex low level simulator and VR engine though a defined and easy to use API and protection from low level memory management problems. Recall that Schmalstieg and Gervautz [SG95] use Python to customise avatar behaviour implemented in C++ because it enables fast and simple construction of applications. This is very similar to the Iota approach of providing low level functionality implemented in C or C++ through a high level API and enabling further customisations in Perl

A developer would clearly need to devote some time to learn to use the API. The amount of time it would take to learn to develop applications using the prototype implementation of Iota would depend largely on the developers background. Attaining a suitable level of proficiency could take between two days to a week approximately provided the developer is a proficient programmer.

Since a thesis is not an appropriate place to detail an API, a feel for the syntax is given through examples of simple scripts in Appendix 5. There are three scripts which show simple scenarios and do not make use of the builder routines.

4.6.1 Summary of user scripts

An OO paradigm was regarded to be suitable for describing concepts in virtual worlds because it is intuitive. High level scripting languages can provide these concepts together with an easy to use API because they are weakly typed, interpreted and do not expose the user to low level memory management. Perl can also provide equivalent functionality to C/C++, but without the need to recompile.

4.7 Integrating components

Both the simulator component and MAVERIK were integrated into the Iota framework by implementing Perl XS stubs and supporting a one-to-one correspondence between Perl routines and C or C++ function or method calls. The simulator was purpose built whereas MAVERIK was taken as an off the shelf component and integrated into the framework. This took approximately two weeks to provide full functionality through Perl. Integrating MAVERIK was complicated by the fact that an early release was used which was relatively inconsistent. The stubs were hand crafted in order to ensure that access to public data and functionality was made available through the API. Any data or methods that could be considered to be internal or private were not supported through the API.

To give an impression of the ease in which alternative components can be integrated into the framework and the limitations imposed on them, we briefly describe replacing each. The following comments are made on the basis that the author integrated both the Simulator engine and MAVERIK into the framework.

To incorporate an alternative version of the simulator engine would be simple provided that the replacement simulator uses particle dynamics. This restriction is necessary because the dynamic restructuring routines rely upon the model being particle based rather than body based. If a new solver is required then the solver method contained in the Solver class could be overridden with a different implementation. This means that an alternative solver could also be used in cases where Iota's solver fails to converge.

If a different VR component is desired then integrating a library such as DIVE should be relatively simple. A tool called SWIG (Simplified Wrapper and Interface Generator) exists for automatic generation of Perl XS stubs simplifying the task of integrating components considerably [Bea96], provided the component lends itself to automatic integration. In this case a VR engine such as DIVE could

be properly incorporated into the framework in a matter of days.

4.8 Summary

The requirements of the prototype implementation of Iota were: a simulator which uses a hybrid particle-rigid body approach to simulating physical behaviour, support for complexity management, core VR functionality, provision of a high level scripting language and an implementation which can be classified as a component framework. The architecture of the prototype framework was described before each component was detailed individually. Finally since each of the low level components are independent of each other, they can be replaced comparatively easily especially if SWIG is used to automate the integration process.

Chapter 5

Case studies and evaluation

In this chapter we present some example simulations which have been implemented in Iota, carried out using the simulator, and rendered in MAVERIK. Many of these case studies were chosen from the subject domain of animation and reimplemented in the Iota framework in order to provide suitable comparison with the types of simulations typically carried out. Where this is the case we draw attention to the original work.

We commence with a section describing rigid body simulation, followed by a section on particle based simulation. Then both techniques are drawn together in a section describing hybrid simulations. Finally, user interaction and a simple VR application are discussed by revisiting some of the examples presented in earlier sections. It is hoped that the variety of simulations illustrated will give an indication of the breadth of different behaviours which can be simulated and manipulated using the Iota framework. All the simulations included in this thesis are available on the accompanying CD as MPEG and Quicktime movies.

To give an impression of the relative complexity of each case study and the ease with which a simulation could potentially be implemented an approximate breakdown of each script is given. This is divided into three categories: the first pertaining to setting up data structures for the models, the second constitutes

the code required for graphical presentation and finally the third category covers the logic, simulation and display loop. A general discussion of the frame rates achieved in the simulations is given in §5.6.

5.1 Rigid body simulation

The following examples are used to illustrate various aspects of rigid body simulation using Iota. Firstly, a simple example of a molecule hung from a particle using a point-to-nail constraint is used to show successful motion computation together with the use of temporal scripted events. Then a simple chain is simulated illustrating the power of graphical representation. Even though a simple underlying model is used, the graphical representation contributes to the visual appeal of the scene. Subsequently, a *Jacob's ladder* simulation is used to highlight the power of dynamic restructuring of the scene and finally a *Newton's cradle* is used to show that much of the detail of a physical system can be abstracted away while still maintaining enough physical behaviour for the simulation to be plausible regardless of the actions carried out on it.

5.1.1 Simulating the motion of a rigid molecule

A simple rigid cyclic molecule is pinned using a point-to-nail constraint and allowed to swing about the pinned particle much like a three dimensional pendulum, as shown in Figure 5.1 (cf. mol1.mpv and mol1.qt on CD). Navigation towards the model is shown from subfigure J onwards. This type of ball and stick model could be useful in a molecular visualisation package. The VR context provides a means for the user to interactively explore the model, and makes it possible to interactively design a molecule by generating events to make and break connections.

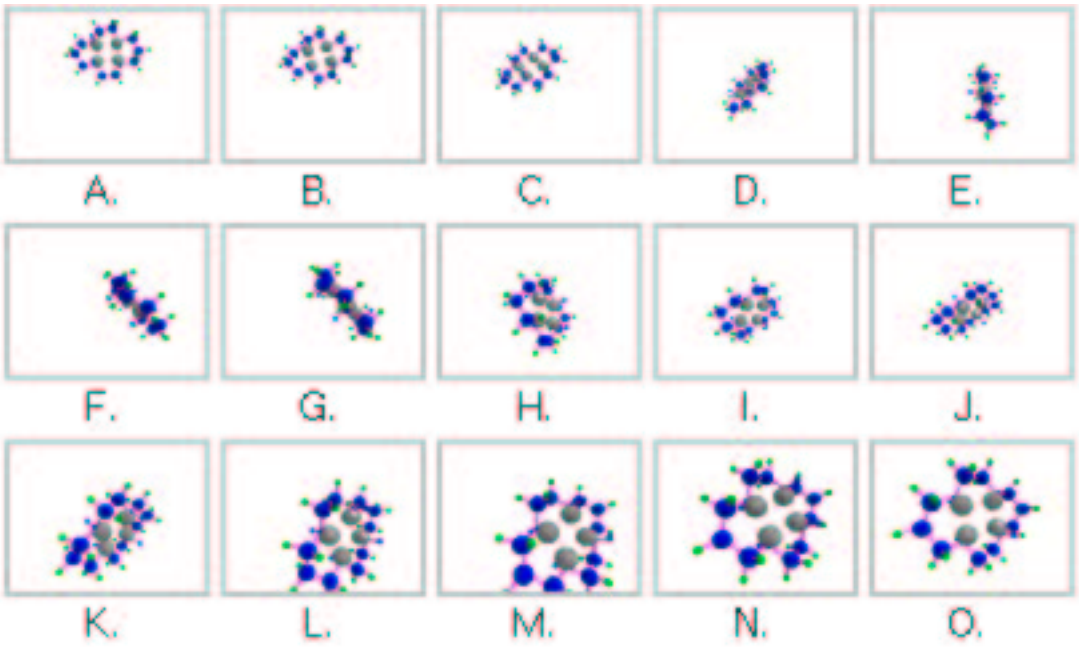


Figure 5.1: Molecule pinned with a point-to-nail constraint

High level scripting languages are excellent at manipulating data in their own right and can be used to retrieve information from other sources thus allowing us to use existing data. To illustrate this we have written a simple import facility which constructs molecules and polyhedra from textual descriptions readily available online.

Since the polyhedra and molecule data are in different formats they were imported and converted into an internal common data structure, from which Iota can build objects. This means that it is simpler to import new shape databases in the future as the build routines do not have to be changed.

The Iota build routine (`Iota::Build`) delegates the work to an extensible set of callback functions supplied by the user. Iota by default provides some simple callbacks in which particles are represented by spheres, bonds by cylinders and users need only supply callbacks that they wish to override. This provides a framework for building objects, but in a very flexible manner as the user has the

freedom to implement any desired visual representation.

Input formats for both polyhedra and molecules contain the relationship between points which represent either vertices or chemical bonds. This relationship can be reflected in the visual representation of the shape through the use of cosmetic bonds. Cosmetic bonds can be broken as shown in Figure 5.2 (cf. mol2.mpv and mol2.qt on CD). In this simulation two connections were broken, the first at the top of subfigure A, and the second at the other end of the molecule in E. The detached portion falls away under gravity while the pinned portion continues to swing, but differently due to the change in its moment of inertia and centre of mass.

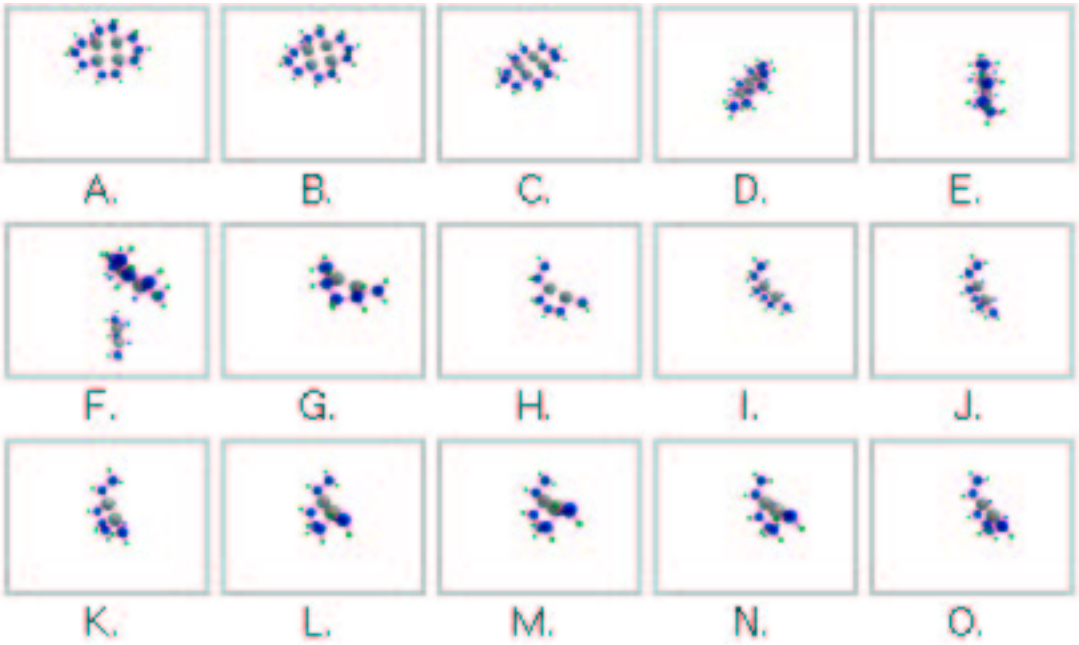


Figure 5.2: Molecule is pinned; two bonds are broken and the molecule breaks into two

In practice there is some overlap between the look and feel of cosmetic bonds and what can be achieved through simply using MAVERIK groupings of shapes (composite objects). However, maintaining a simulator representation of cosmetic bonds simplifies computation in the event that the scene graph is modified. One

of the most useful features of cosmetic bonds is that they can be queried for a transformation matrix to be applied to MAVERIK objects whose local co-ordinate system has them centred at the origin and of unit size.

Code breakdown: Scene Setup: 10 lines. Presentation: 20 lines. Simulation/Behaviour/Display: 10 lines.

5.1.2 Simulating a chain

To construct a high level representation of a chain we use a routine defined in Iota (`Iota::chain`) to build the same common data structure referred to in the previous section. Subsequently the Iota builder routine is called to build the simulator and MAVERIK models, again as described earlier.

In this particular example the chain is represented in one of two possible ways by passing a named callback function to `Iota::Build` which will create a MAVERIK representation of a link. If a rope-like effect is desired then this is simply a shaded cylinder and if a chain-like appearance is desired then each link is rendered as a torus. To give the appearance of the alternating orientation of links as found in a real chain, the tori are rotated using functionality implemented in Iota's transformation routine (p. 202). Both these representations of a chain are shown in Figure 5.3.

During the simulation, the chain is shown suspended from a nail using a point-to-nail constraint and is allowed to swing under gravity (c.f. Barzel and Barr [BB88]). Scripted events have been used to invoke calls to `combine` and `separate` to script a portion of the chain falling off and reassembling itself. The sequence of images in Figure 5.4 taken at intervals of five frames is used to illustrate this simulation. Upon separation in subfigure L, a force function is activated between the two particles to prevent the detached portion from falling away completely. After some time has elapsed, a second force function is activated

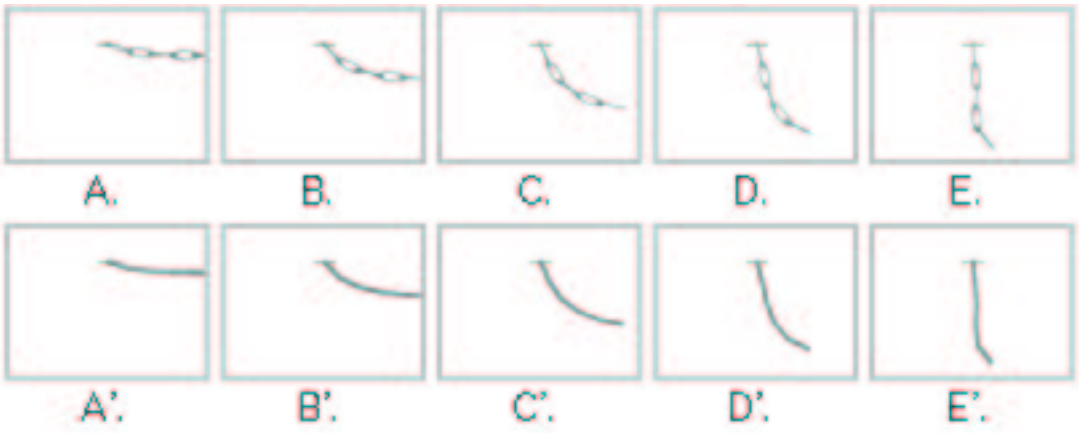


Figure 5.3: Two representations of a chain

to bring the separated portion into close proximity with the suspended chain and a call to `combine` is made to reassemble it as shown in subfigure R.

Code breakdown: Scene Setup: 20 lines. Presentation: 20 lines. Simulation/Behaviour/Display: 10 lines. Two alternate representations are included in the presentation category.

5.1.3 Simulating a Jacob's ladder

In this section we introduce a children's toy known as a "Jacob's ladder". It is traditionally constructed using blocks of wood and ribbons as shown in Figure 5.5. Typical Jacob's ladders are between five and nine links long.

Due to way in which the ribbons connect to the blocks, each block can really only tip in one direction which alternates along the length of the toy. If there are enough blocks in the ladder, multiple ripples can be sent down its length. Consider the situation in which two ripples are propagating along the toy; one can speculate that the situation may arise where the second ripple commenced too soon after the first and so could catch up with it cancelling them both out.

To be able to simulate the motion of a virtual Jacob's ladder [GM99], it

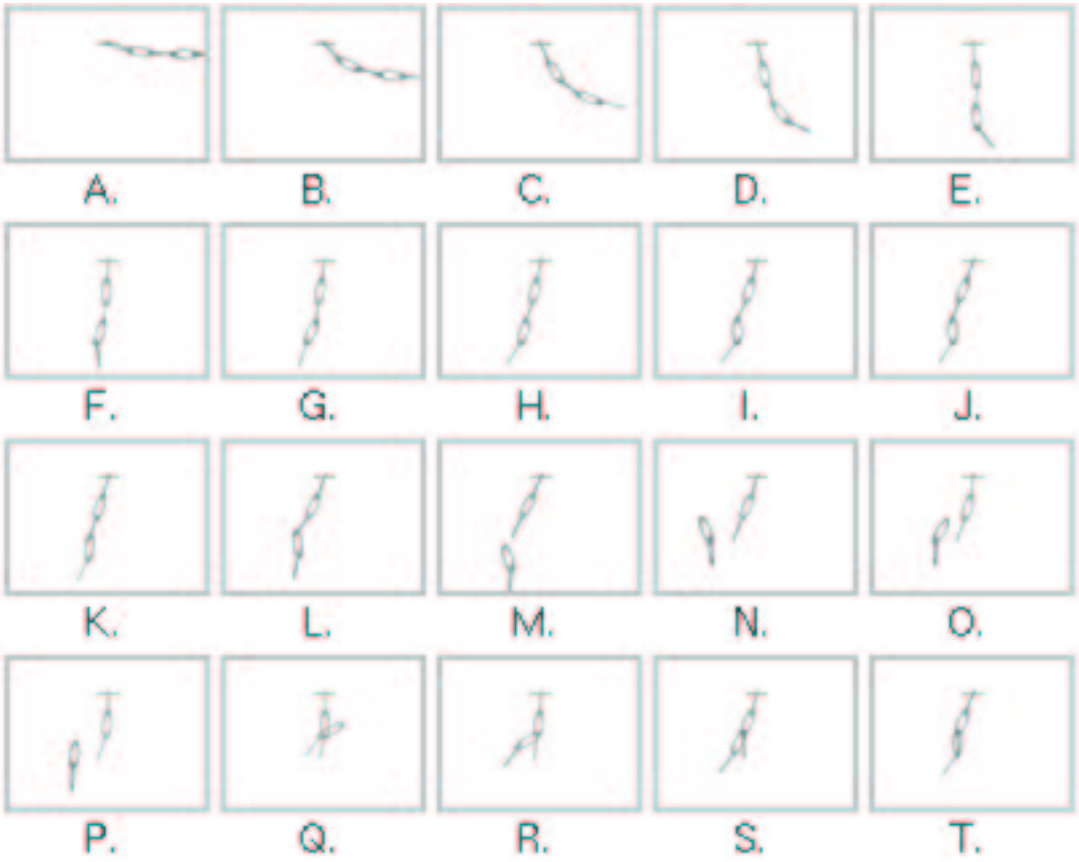


Figure 5.4: Scripted breaking and reassembly of an articulating chain

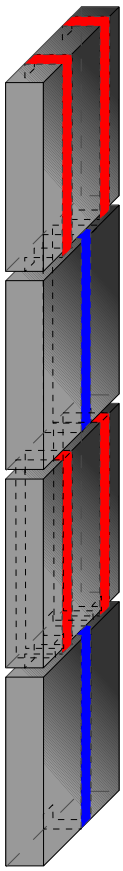


Figure 5.5: A Jacob's ladder

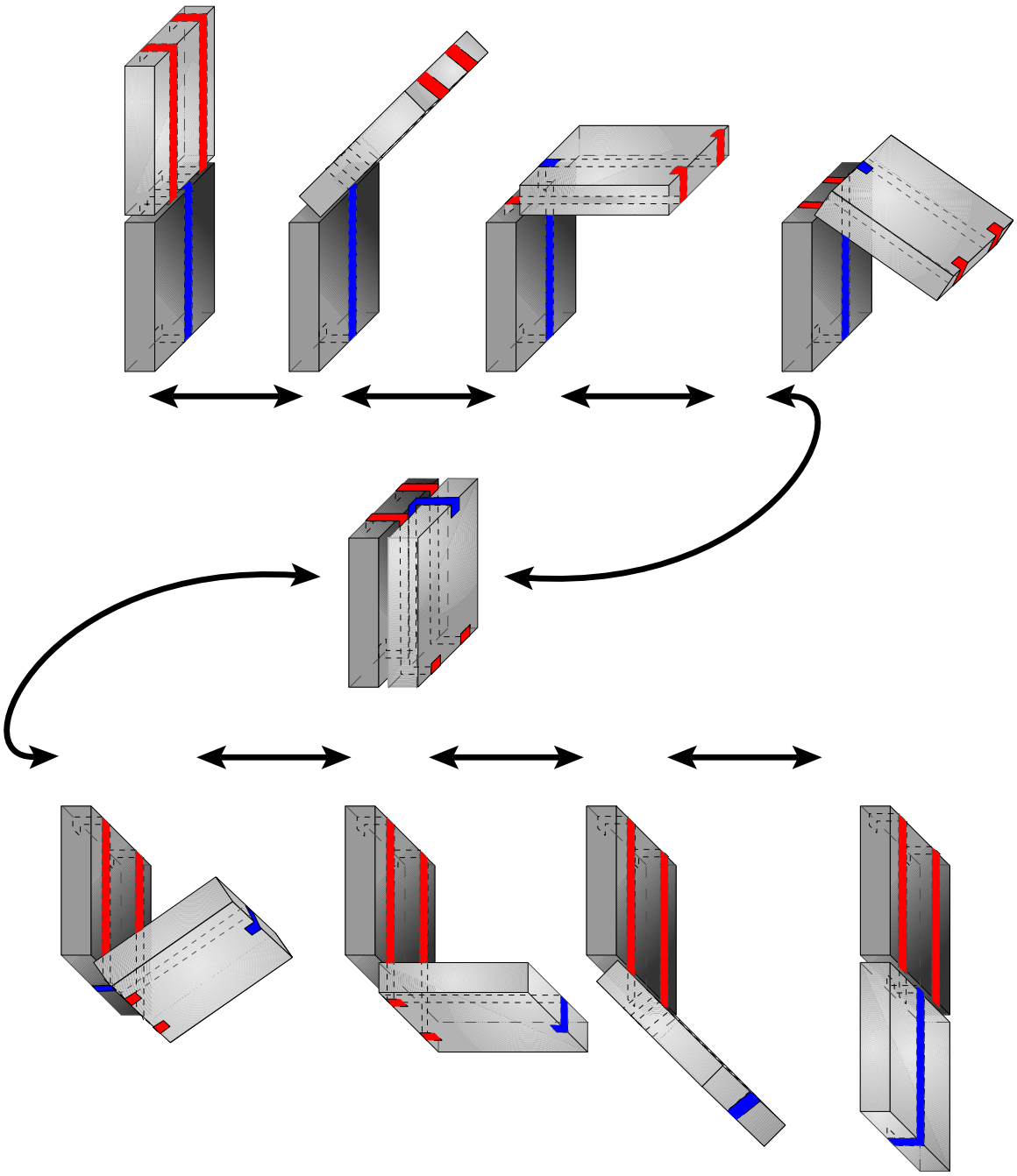


Figure 5.6: The behaviour of a single link in a Jacob's ladder

is necessary to understand the behaviour of one link (two blocks) as shown in Figure 5.6 We assume that the second block is held still so that the behaviour of the first block can be seen in relation to it. The first block, shaded light grey, starts at the top with two red ribbons on it. When it is tipped over, it swaps to be at the bottom turning 180° at each end of the second block. Notice how the ribbons swap so that the block now at the top still has two ribbons. If there were another block in the ladder (three blocks and two links) then the first block would be permitted to topple through 180° while hinged at the top end of the second block but only approximately 90° at the bottom due to the physical restriction imposed by the third block.

Now consider what happens when we hold the first block and allow it to tip. Say it rotates clockwise through 180° bringing the top of it in contact with the bottom of the next block; if we continue to hold it then the second block would tip through 180° anti-clockwise and so on allowing a ripple to propagate down the length of the ladder. The important thing to realise is that each block's orientation is flipped and this can be seen by the alternation of the ribbons on its face. The motion of the blocks appears simple at a first glance but as we can see it is complex and hard to visualise. To be able to simulate this type of motion we need to specify a suitable model which will exhibit the appropriate behaviour.

Building the model

To utilise the physical model for computing motion in the simulator, each block is modelled using simulator primitives. A number of possible configurations of these primitives could be used to construct the model. Recall from Chapter 3 that a hinge consisting of n particles contributes $n - 1$ equations in each dimension to be solved simultaneously. In the interests of performance it would be desirable to minimise the amount of computation required. A carefully chosen model can

significantly improve the interactive response of the simulation.

Imagine viewing a Jacob's ladder side-on as was shown in Figure 5.5. In this diagram the toy was oriented such that the narrow side face of each block appeared to be at the front with the remainder receding into the paper. Our model abstracts away much of the detail while trying to secure as much of the behaviour as possible. This is achieved by retaining the essence of a block by representing it as a two dimensional body with four particles situated at the corners of the narrow face and one in the centre. The advantage of this approach being that each hinge consists of two particles thus providing a contribution of one equation in each dimension, in other words two equations per hinge. We justify this by the observation that the links in the ladder cannot twist and so each hinge actually only has two degrees of freedom.

Unlike the model, blocks in a real Jacob's ladder do not rotate about a hinge, instead they rotate approximately about a point where the ribbons on each block cross. However at speed it appears simply as if the hinging occurs at the point where the blocks meet, corresponding to hinges in our model, further justifying the approach taken. A numbering convention, shown in Figure 5.7 was adopted to represent the bodies and particles which make up the model.

Each of the N blocks is constructed from a body of five particles and is assigned a number from 0 to $N-1$. The four corner particles are themselves numbered such that they reflect the system's symmetry and this in turn allows concise code to be written using the relationships between adjacent blocks. A number of observations may be made with respect to this labelling of particles:

- Particles which hinge will always be numbered the same on both blocks.
- Odd numbered blocks (1, 3, 5, ...) will hinge on corners 0 and 1 with previous blocks.
- Even numbered blocks (2, 4, 6, ...) will hinge with the previous block on

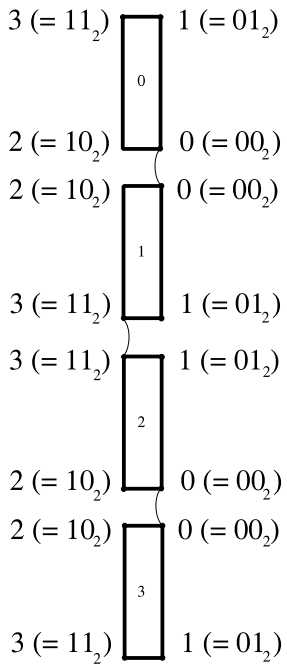


Figure 5.7: Numbering convention adopted in the Jacob's ladder

corners 2 and 3.

- Exclusive-oring with 01_2 will enable us to traverse a block along its length from any corner.
- Similarly, exclusive-oring with 10_2 enables the width of a block to be traversed from any corner.
- It has been found useful to view each block as possessing two orientations, *lengthwise* and *widthwise*. In the initial state lengthwise orientations are alternately up and down, whereas widthwise orientations point right to left.

The implications of these observations with regards to simulating the model will become clearer in the following section. Before the process of simulating this model can be described, it is appropriate to discuss the data structure used to store relevant information for the simulation.

Figure 5.8 shows the data structure for the ladder representing its state as well as retaining Perl references to C structures in MAVERIK and the simulator.

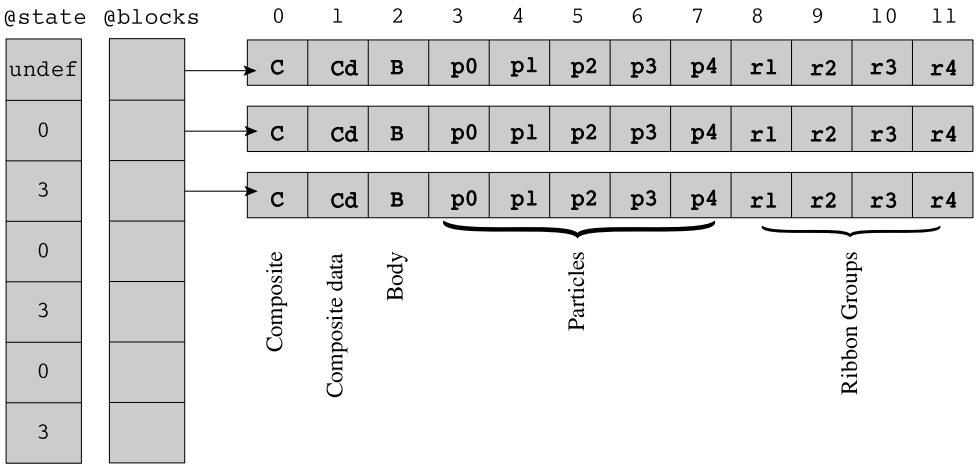


Figure 5.8: Data structure for representing Jacob's ladder state

Initially the first pair of blocks hinge on corner 0 and the second on 3 alternating similarly along the length of the ladder (c.f. Figure 5.7), and these are stored in the `state` array. Each element in the `state` array holds a conceptual relationship between pairs of blocks stored in the `blocks` array. Since the first block in the Jacob's ladder does not have a relationship with a previous block there is no need to store any information about it. Each block is composed of a number of particles and this is reflected in the data structure. Pointers in the `blocks` array reference further arrays containing the following data per block:

Composite and composite data: These contain the MAVERIK representations of a block. One is used for rendering and the other for transforming the composite object.

Body and particles: Simulator representations for each block are stored in these elements.

Ribbons: It is necessary to store some information relating to ribbons as their configurations change during the simulation.

Since Perl is weakly typed it is possible to package together the MAVERIK, simulator and Perl data into an array. Logically we are storing all the attributes of the blocks, in the Jacob's ladder object, in simple arrays. Contrast this with other languages such as C where arrays can be used to do little more than store elements with the same type.

Simulating the model

To simulate the motion of the entire ladder only involves specifying the behaviour of a single hinge. Once the constraints on the motion of the model are understood, the physical model in the simulator can be used to compute the overall motion of the system. As we will see, many subtleties in the motion of the real toy appear in the virtual one. First let us consider the motion of one hinge again, but this time in a more detail.

Figure 5.9 shows a pair of hinged blocks. Their range of allowable motion is confined to quadrants 1 and 3. Notice the position of the black dot representing the hinge as it is significant and implies that the shaded quadrants 2 and 4 are out of bounds. The lighter grey shaded regions are inactive but become actively forbidden regions after a flip-flop. Moreover observe the lengthwise orientation vectors on the blocks as these too are also important.

Consider the first configuration of the pair of blocks, assuming that the second block remains stationary. If an attempt is made to push the first block into the shaded region both blocks will *lock* preventing any further intrusion. The first block is allowed to move into quadrant 1 if it is subject to an appropriate unbalanced force. Following the sequence of allowable motion (reading Figure 5.9 across and down) of the block it can be seen that after a 180° rotation the hinge is broken and reassembled at the bottom end of the second block and then the first block is allowed to continue through another 180° turn. This swapping of

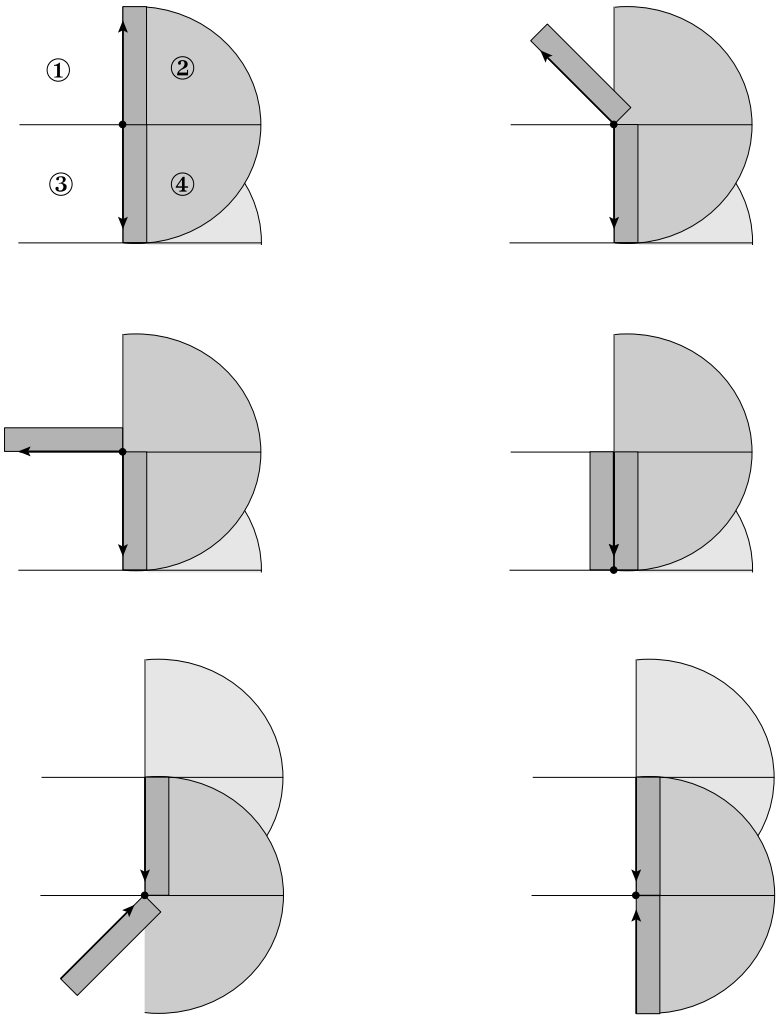


Figure 5.9: Allowable motion of a hinge in a Jacob's ladder

Algorithm 4 Main Jacob's ladder algorithm

```
1: for all block in ladder (excluding first one) do
2:   if hinge is locked then
3:     if torques acting on current and previous blocks cause them to unlock
       then
4:       Unlock hinge {Algorithm 7}
5:     end if
6:   else
7:     Calculate lengthwise orientation of current block
8:     Calculate lengthwise orientation of previous block
9:     Calculate cross-product of both orientations
10:    if cross-product < 0 then
11:      Sum both orientations
12:      if magnitude of sum is large then
13:        Perform flip-flop {Algorithm 5}
14:      else
15:        Lock blocks {Algorithm 6}
16:      end if
17:    else
18:      {Normal tipping – No action}
19:    end if
20:  end if
21: end for
```

the blocks is termed a *flip-flop*.

Now observe the relationship of the lengthwise orientation vectors, initially the first block's vector points upwards and the second block's points downwards. After a 180° turn the vectors both point down and after a flip-flop they point towards the hinge. These orientations can be examined to determine a number of things. Firstly if the magnitude of the cross product of the two vectors is less than zero then an attempt has been made to enter quadrant 2 or 4. This means that some sort of action is required, the type of which can be determined by adding both vectors together and examining their magnitude. A small number resulting from the sum of the vectors implies that the pair of blocks have to be locked (c.f. Algorithm 6) whereas a large number implies that the blocks have to be flip-flopped (c.f. Algorithm 5). If the magnitude of the cross product of both

vectors is greater than zero then no action is required. The behaviour of one pair of hinging blocks has now been formalised so we can describe the algorithm for simulating the Jacob's ladder (shown in Algorithm 4 above).

The flip-flop algorithm works by performing a `separate` on the two points which make up the hinge. To derive the numbers of the new particles to hinge upon, the separated particle number is exclusive-or'ed with 1 (01_2) and recombined. The flip-flop also causes the configuration of ribbons to change. While this is only cosmetic, it is an important visual cue in the simulation. The inner and outer ribbons have to be swapped on each of the two blocks. A number of possible ways of implementing this exist, the best and simplest is to create all the ribbons which can be rendered but to hide those which are not visible.

Algorithm 5 Algorithm to flip-flop blocks in the Jacob's ladder

1: Separate currently hinged corners
2: Calculate particle numbers for new hinge {by exclusive-oring current particle numbers with 01_2}
3: Combine particles for new hinge
4: Hide and draw ribbons as appropriate

To lock hinges a further `combine` on the hinged corner is performed. To understand why, recall from Chapter 3 (p. 116) that combining two points from the same particle results in the two parent bodies being combined, and so making the hinge rigid. Recall also that during this type of combine, one of the points is removed because it is considered to be redundant. Anticipating that this point will be required when the hinge is unlocked, it is subsequently recreated so that there are still ten particles in the new body (five from each original body). Since one of the two original bodies is destroyed, the body entry in the `blocks` array is set to be undefined.

Finally the hinge is marked as locked, so that it can be treated appropriately, and the following method has been adopted for this. Since the elements in the `state` array only ever hold the two particle values which a block hinges on, the

two unused values may be used to signify a locked hinge.

Algorithm 6 Algorithm to lock a hinge in Jacob's ladder

1: Perform a further combine on the hinged corner {This has the effect of causing both blocks to become consolidated into the same body}
2: Clone hinged point {to retain ten points in new body, as before}
3: Mark hinge as locked

Unlocking a hinge is performed by detaching the central particle from one of the bodies using `separate` to create a new body which is then entered into the `blocks` array. Each of the block's four remaining particles must also be separated and combined with the new body. In very rare circumstances the body which has become unlocked may actually be locked to a another block and some further manipulation is required to maintain the correct parent articulate. Finally the hinge is recreated, by combining the corners in question, and marked as unlocked.

Algorithm 7 Code to unlock a hinge in the Jacob's ladder

1: {Reconstruct two separate bodies as it was prior to the locking action}
2: Detach central particle from notional second body
3: Insert new body into data structure
4: **for all** particles to be repointed to second body **do**
5: Detach particle from parent body
6: Repoint to new body
7: **end for**
8: If next body is also locked, repoint it to new parent
9: Rebuild hinge using combine
10: Mark hinge as unlocked

The first set of results presented in Figure 5.10 (cf. jacob.mpv and jacob.qt on CD) shows screen shots of the Jacob's ladder taken at intervals of eight frames. Although it is difficult to see the subtleties of the motion from such images, we would like to draw the reader's attention to subfigures A through G. Notice how the third block is pushed down and then pulled back up over the sequence, in particular a significant change in position has occurred between E and F. From subfigures G to R we have gradually navigated down to follow the propagation of

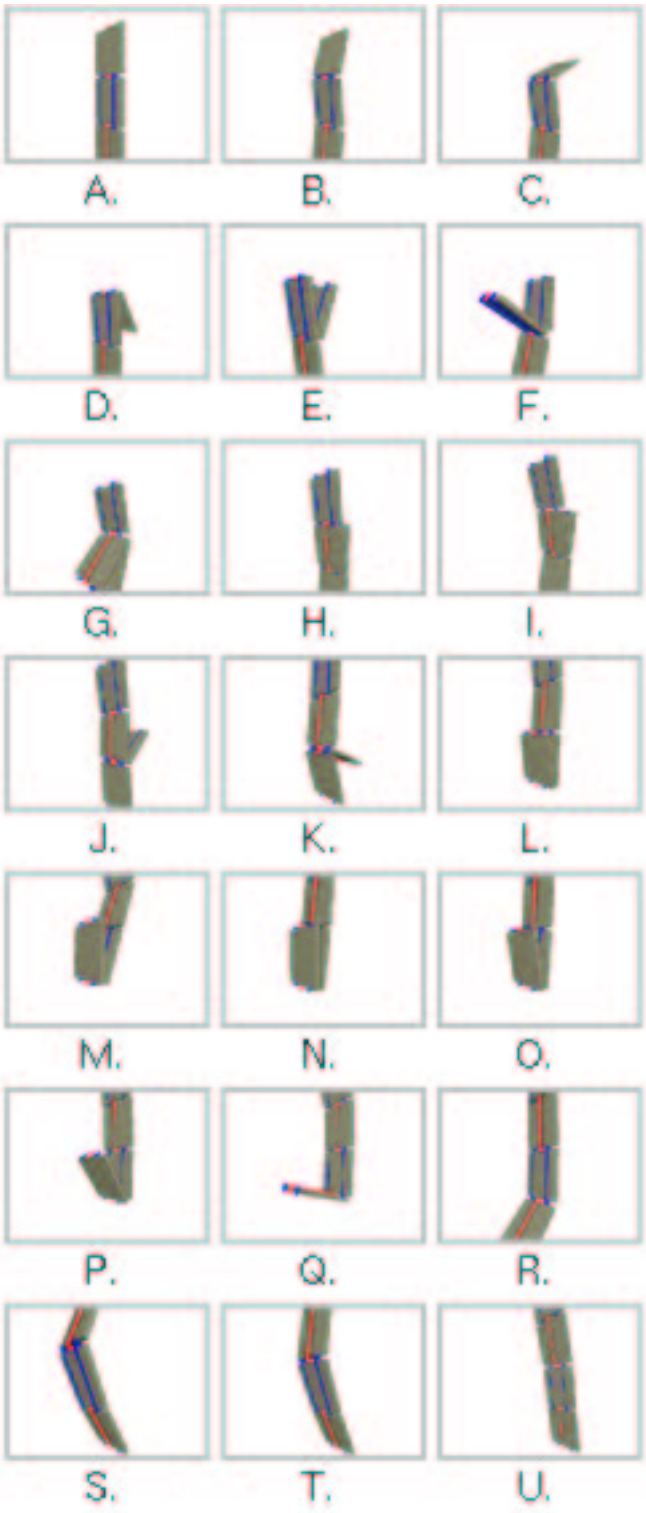


Figure 5.10: Jacob's ladder simulation

the ripple. Subfigure N is of interest as the third block is momentarily unable to tip because the previous two blocks have locked and their united motion disrupts it. This behaviour can occasionally be observed in the real toy. Also, notice how the second and third blocks appear to have become locked in S and T and are then possibly unlocked by U. The types of subtleties introduced can only result from the use of a physically based model.

Summary of Jacob's ladder simulation

This particular example makes heavy use of our dynamic restructuring routines and each flip-flop or lock fundamentally changes the underlying model and was a non-trivial case study to implement. It was originally unclear whether or not a convincing simulation could be achieved through the careful use of dynamic restructuring. We have only shown this restructuring occurring as scripted events for the time being, but there is no reason why the user cannot affect such changes. Indeed any interactive breaking or reassembling is possible via these routines. Any degree of dynamic restructuring can be effected by various permutations of combines and separates.

Although many constraint solvers have been implemented to generate animation sequences [BB88, Bar92a], few have been tailored specifically for virtual reality applications [Fau99, Woo99, KSZB95, KSB] and even fewer enable scene data to be dynamically restructured [WGW90].

We have found that our Jacob's ladder model can be simulated at interactive frame rates. No collision or contact constraints are implemented so blocks can be seen to occasionally pass through one another for a single frame.

Although the connectivity of blocks in a real Jacob's ladder differs from that in our model and as such the model is not a faithful representation, our virtual ladder contains all the subtleties of motion exhibited by the real toy. Finally, the

virtual Jacob's ladder is as addictive as the real thing.

Code breakdown: Scene Setup: 120 lines. Presentation: 100 lines. Simulation/Behaviour/Display: 160 lines.

5.1.4 Simulating a Newton's cradle

A Newton's cradle is an interesting executive toy often found on office desks and its appeal lies in the hypnotic unintuitive motion it exhibits. The toy is composed of a set of metal balls suspended on wires from a cradle like frame, as shown in Figure 5.11. Lifting a ball at one end of the cradle, and releasing it results in a collision between the moving and stationary balls. Interestingly, upon collision, the moving ball almost comes to an abrupt halt and an impulse (force exerted over a negligible time period) is transmitted through the set of balls causing the ball at the other end of the cradle to continue the motion. This behaviour is attributable to the fact that momentum before and after a collision is conserved, and since in this case collisions are considered to be elastic, energy too must be conserved.

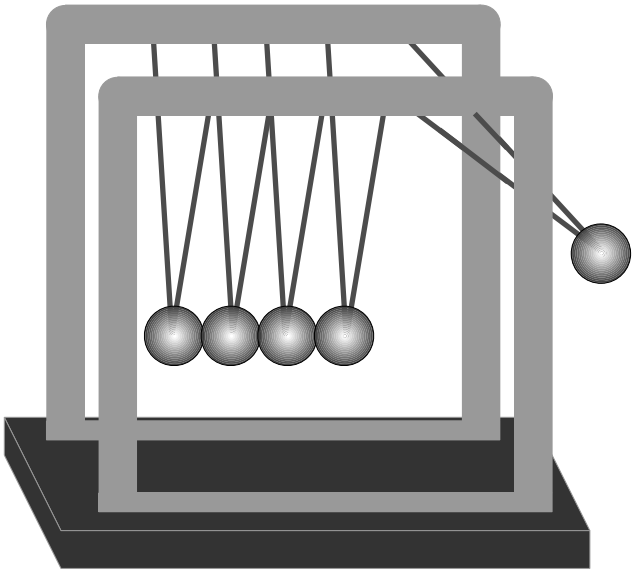


Figure 5.11: A Newton's cradle

Grouping together two balls, lifting them and releasing in the manner described above, results in the impulse being transmitted through to two balls at the other end of the cradle. This behaviour can be shown for any number of balls and some examples are illustrated in Figures 5.12 and 5.13 (read in the order indicated by the arrows).

Notice how the cradle behaves when a single ball is raised at one end of the cradle and two balls at the other as shown in Figure 5.13. When both sets are released at the same time the groups appear to swap upon collision. This and further observations regarding the motion of a Newton's cradle are discussed in the following section, together with the motivation for the choice of model used to represent it.

Building the model

There are numerous examples of the Newton's cradle available on the World Wide Web but very few are suited to incorporating into a virtual environment. To explain this statement, consider the traditional methods of modelling a Newton's cradle.

Purely physically based methods exist which simulate the impulses travelling through the balls suspended from the cradle (c.f. MathEngine [Woo99]. In this case no specific insight into the behaviour is required since the physics of the real world is modelled. The problem with this approach for VR lies in the processor time required to simulate impulses with purely elastic collisions effectively. Moreover, the real world is not as perfect as the results obtained by using simple Newtonian physics would indicate. To simulate the subtleties of motion requires proper consideration of drag caused by air resistance and energy loss during collision. Most physical models employ an ad-hoc technique for simulating this rather than a proper physical treatment as this would rapidly become too complex to

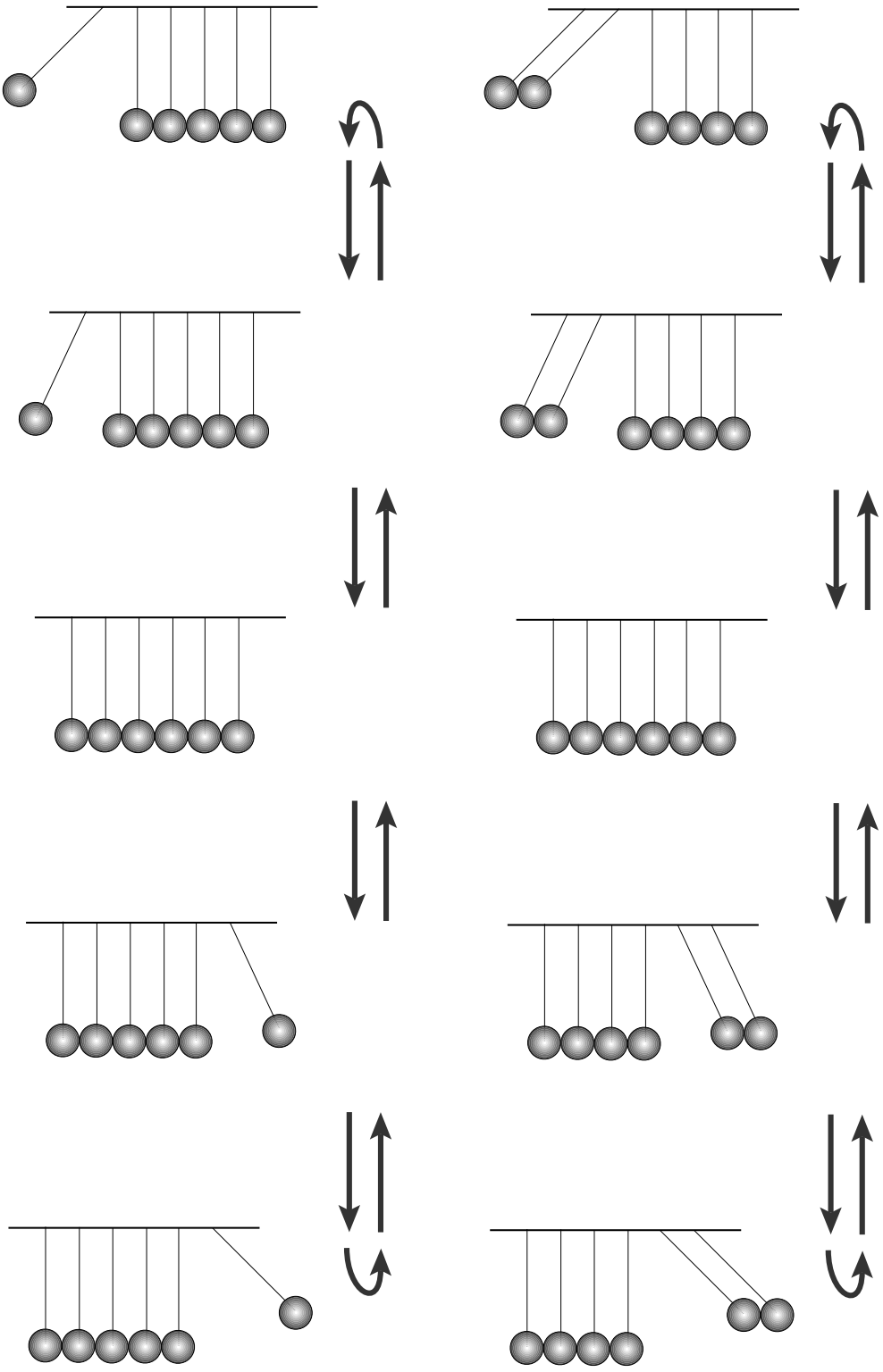


Figure 5.12: Motion of a Newton's cradle

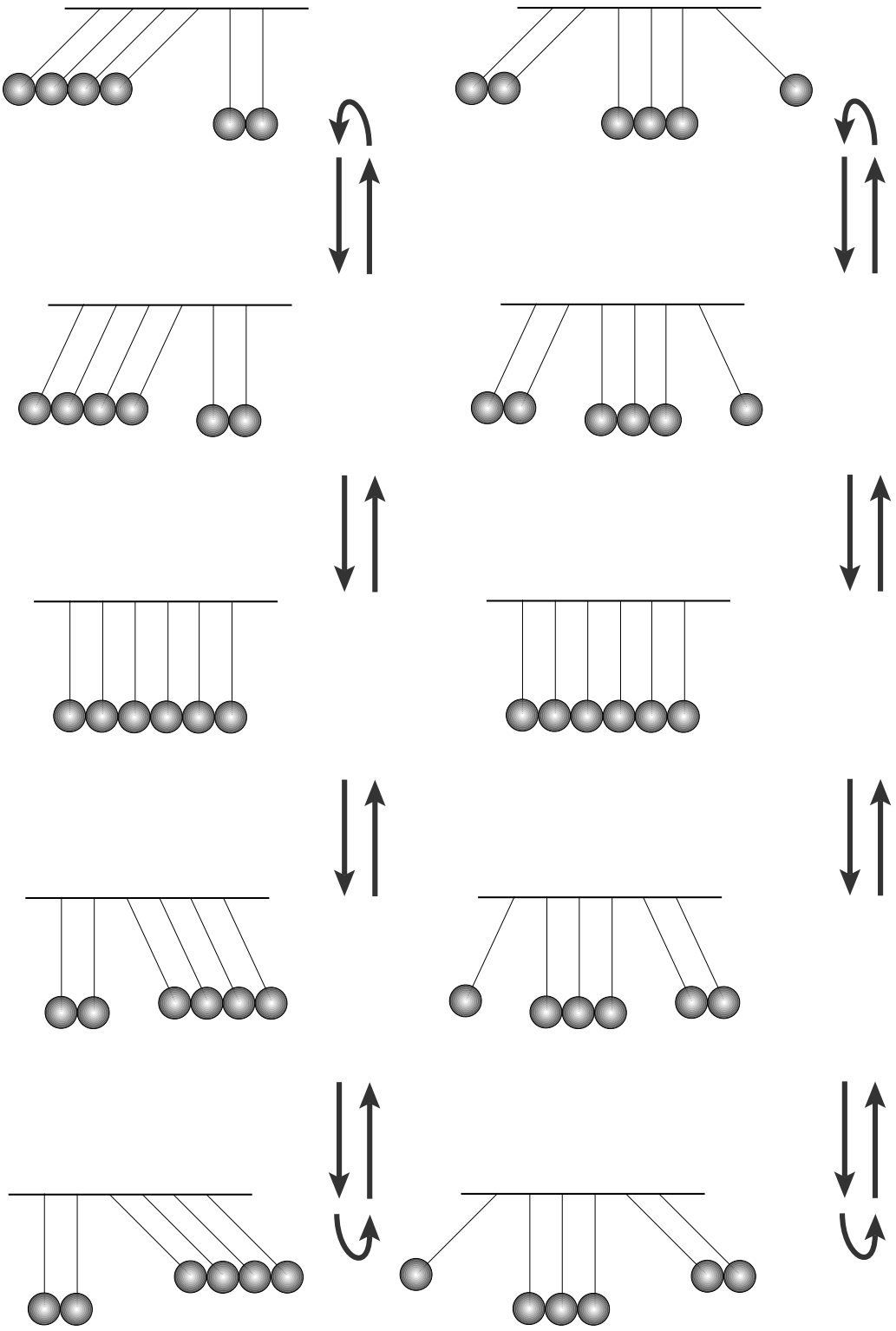


Figure 5.13: Further motion of a Newton's cradle

simulate. Even if the simulation is fast enough to be interactive (which is almost certainly the case in the example of a Newton's cradle) a rich VE provides a broader context which typically cannot afford to have a large proportion of the processor time devoted to one particular simulation.

Another approach often used involves ad-hoc animation of a Newton's cradle. This is the other extreme, the behaviour of a Newton's cradle is known and this knowledge is exploited. The user may press a key to instruct the program to lift a specific number of balls upon which the animation is played. Many of these examples can not simulate the swapping behaviour exhibited by a Newton's cradle and often the model has no real concept of gravity. The motion is merely computed using elementary trigonometry. In a VE the user should be free to *play* with the Newton's cradle, so for every combination of balls lifted, the model must promptly simulate in a correct and visually pleasing manner. If the type of ad-hoc approach described above is used to simply play back an animation then every possible combination of the motion would have to be provided.

So, from this discussion it can be concluded that a physical model is desirable but it is also ideal to reduce the problem to the simplest possible form to minimise the amount of computation required. Two observations regarding the behaviour of a Newton's cradle are exploited in the design of our model; the motion of each ball in a group is exactly that of a pendulum and such groups swap positions in the cradle upon collision. We use a combination of an ad-hoc model with physically based simulation of a pendulum. This choice of model is justified by the fact that in isolation each ball suspended from the cradle is in fact a pendulum, furthermore a group of any number of pendulums move as one. So the simulator is used to evolve the motion of a single pendulum until required to intervene in the case of a collision

Since all balls in a group move in unison only one pendulum is required to

simulate each group. Consider the case in which collisions are ignored; initially when the balls in the cradle are all at rest, only one pendulum will suffice to simulate the motion. If all the balls are pushed maintaining the group intact then again simulating only one pendulum is enough. Splitting the group into two changes the situation, now two pendulums are required to simulate the motion. So it is safe to say that a linear relationship exists between the number of groups and the number of pendulums required to simulate them. Figure 5.14 illustrates two possible groupings for a six ball Newton's cradle together with the underlying pendulums simulated. The first underlying pendulum used to simulate the motion of a group consisting of two balls of unit diameter is centred about the group at an offset from the centre of the cradle of -2. A second group consisting of four balls is represented by the underlying pendulum offset at $+1$ unit to the right of the origin.

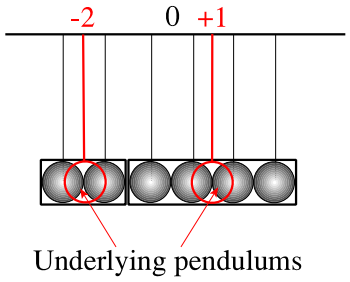


Figure 5.14: Newton's cradle groupings

A data structure, shown in Figure 5.15 is used to represent each group together with the attributes specific to it. The centre of a group corresponds to the position of the underlying pendulum being simulated. Furthermore, the number of notional balls in a group must be stored to be able to render them.

Since there is only one pendulum in the underlying model of a group, only one simulator body has to be stored together with its constituent six particles: one at the top hinge, one at the centre of the ball and four around its circumference.

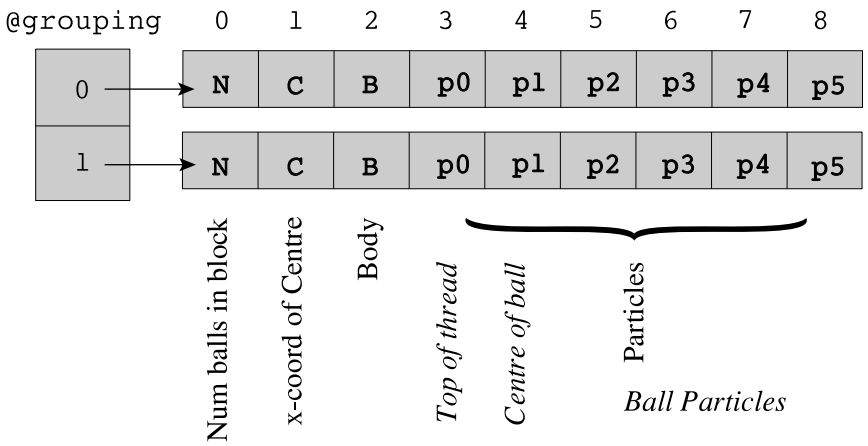


Figure 5.15: Newton's cradle data structure

An appropriate moment of inertia is maintained by placing four particles around the circumference of a ball. All the groups are held in the `grouping` array from left to right by convention, to simplify collision detection.

Simulating the model

Simulating the motion of the reduced number of pendulums in the Newton's cradle is automatically handled by the simulator until a collision takes place, at which point some action is required. Before any action can be taken, we have to actually determine whether a collision has taken place. A custom collision test based on geometry is used; alternatively it is possible to use MAVERIK's collision detection routines if required.

In the event of a collision an offset, as shown in Equations 5.1, is applied to both groups involved and then they are swapped in the `grouping` array.

$$p_a = p_a + n_b * \text{width\_of\_ball} \tag{5.1a}$$

$$p_b = p_b - n_a * \text{width\_of\_ball} \tag{5.1b}$$

The quantities p_a & p_b represent the positions of the underlying pendulums in

the cradle and n_a & n_b are the number of balls in each group involved in the collision. Another collision may occur immediately with the swapped group and its neighbour so it is necessary to retest before rendering. The effect of this manipulation is that groups are literally swapped and allowed to continue their motion.

A simulation of the Newton's cradle is illustrated by Figure 5.16. Two balls are scripted to be raised and then allowed to fall. Although it is difficult to do justice to the simulation from a sequence of images, it is hoped that an impression of realistic behaviour is given. By subfigure B, the initial group has been split into two. All subsequent images retain these two groupings as no other balls are raised.

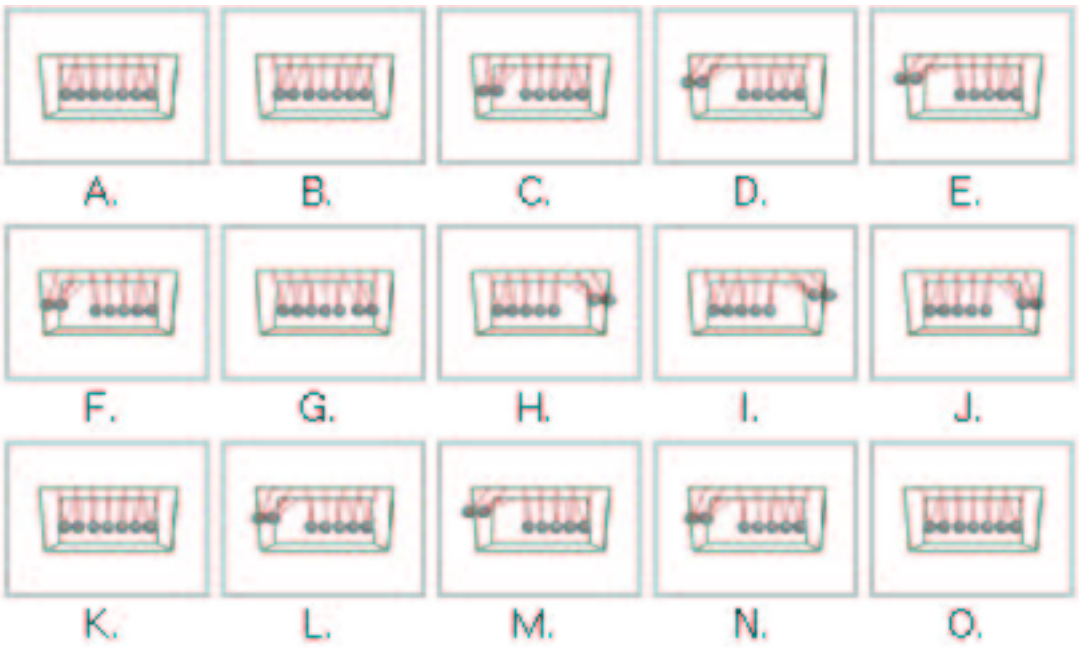


Figure 5.16: Simulation of a Newton's cradle

Code breakdown: Scene Setup: 30 lines. Presentation: 60 lines. Simulation/Behaviour/Display: 140 lines.

5.2 Particle based simulation

Particle systems are used in this section to simulate the behaviour of virtual *boids*, a soft molecule and a 'breathing' Bucky ball. Although these simulations are quite simple examples of interacting particle systems they have been chosen to illustrate the idea that any force function can be scripted in Iota and attached as a callback function to create bonds (statically or dynamically) between particles. This mechanism allows the user to exercise very precise control by setting up several different types of internal and external force functions to manipulate the system.

5.2.1 Simulating boids

Reynolds' classical boids simulations [Rey87, Rey, Nab] are well known in computer animation and such algorithms have been widely used to simulate the flocking behaviour of animals in films. This simple but effective result was achieved by using interacting particle systems and is reproduced here using Iota.

To simulate the motion of our version of virtual boids [Bro, ROD+98, RSG], we use an interacting particle system. All particles interact with each other through a long range attractive and short range repulsive force function. In effect, each boid maintains a small separation from every other boid but they still move in unison. The simulation is driven by a scripted driver particle which moves along an arbitrary path and the flock will follow.

Currents are added to the simulation to enhance the complexity of the motion of the flock. This is achieved by partitioning the world into grid cells and using a *fractional Brownian motion* (fBm) function to generate a three dimensional fractal force vector field. An implementation of Perlin's noise and Musgrave's fBm functions [MPPW94] were added to the vector class in the simulator. The fBm function provides a fractal three dimensional field with currents and uses

noise as a basis function. This results in a simulation of the gradual transition of flow direction through a fluid medium.

MAVERIK provides a means for reading a description of a composite in AC3D format (a 3D file format). Such descriptions are readily available online for many common objects, but because data was locally available for an AC3D fish the boids were rendered as fish. The sequence of images shown in Figure 5.17 illustrates a shoal of fish schooling towards the driver particle. Subtleties in the motion of the fish are more apparent in the actual simulation where the fish appear to glide while jostled by turbulence. This can be clearly seen in the movie on the accompanying CD (cf. fishies.mpv and fishies.qt).

Figure 5.17: A shoal of fish

Code breakdown: Scene Setup: 30 lines. Presentation: 40 lines. Simulation/Behaviour/Display: 15 lines.

5.2.2 Simulating a soft molecule

Recall from §5.1.1 that descriptions of molecules and polyhedra could be read into Iota and used to render rigid molecules. Now consider, what happens when each connection is represented as a force function rather than a rigid connection. The resulting system, an example of which can be seen in Figure 5.18 (cf. mol3.mpv and mol3.qt on CD), appears to be soft and deformable.

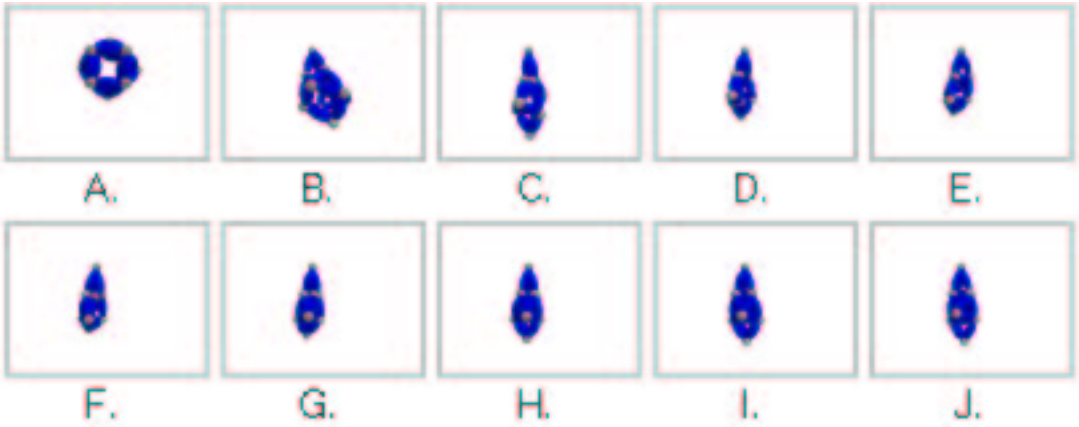


Figure 5.18: A soft molecule

In this particular example a simple force function was used to represent connections. One of the particles in the model was fixed using a point-to-nail constraint and the molecule was allowed to swing about it like a three dimensional pendulum, much like the rigid molecule example earlier. However, since the molecule is built as a soft object, its motion characteristics differ significantly from the rigid example. Notice that the molecule settles into an elongated shape due to the effect of gravity. Finally, for aesthetic appeal, connections are rendered as ellipsoids.

Code breakdown: Scene Setup: 10 lines. Presentation: 30 lines. Simulation/Behaviour/Display: 10 lines.

5.2.3 A breathing Bucky ball

In this section the same method (described above) is used to import and construct a soft molecule, but with two major differences; firstly gravity is not applied to the system, and secondly the implicit system particle is placed in the centre of the molecule. Setting up a repulsive force function between this central particle and all the other particles in the molecule prevents the structure from collapsing.

Buckminster fullerene is the common name for Carbon-60. For many years chemists believed that it would be possible to synthesise carbon in this peculiar form, and only relatively recently has this objective been achieved. Buckminster fullerenes are commonly known as 'Bucky balls' and the connectivity of carbon atoms is akin to the patchwork on a football. Mathematically, this shape is referred to as a truncated icosahedron, shown in Figure 5.19 (cf. bucky.mpv and bucky.qt on CD), and has generated much scientific interest due to the prospect of synthesising super-conducting Bucky balls and tubes by inserting particular metals inside the mesh.

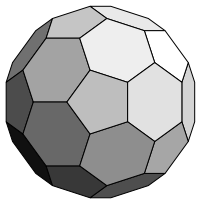


Figure 5.19: A truncated icosahedron

Oscillations about the rest length of a chemical bond occur naturally thus causing Bucky balls and other chemical molecules to appear to breathe. The degree to which this occurs cannot be easily observed as atomic distances are so small. In the context of a molecular modelling application, it would be useful to visualise the breathing behaviour of such compounds, since this could give some insight into whether or not metals could be introduced through the gaps as the

molecule breathes. So simulating this behaviour is an interesting exercise and the montage of images in Figure 5.20 shows our virtual Bucky ball pulsating. After subfigure H, the viewpoint is placed (by navigation) within the Bucky ball.

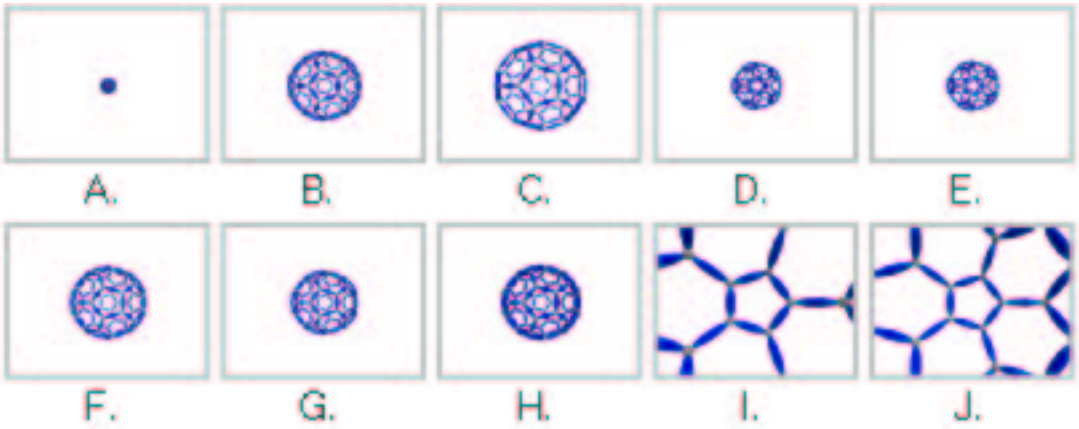


Figure 5.20: Breathing Bucky ball simulation

Although, the force functions used for this particular example are not especially realistic, we have shown that this type of soft mesh can be simulated at interactive frame rates using Iota.

Code breakdown: Scene Setup: 10 lines. Presentation: 30 lines. Simulation/Behaviour/Display: 20 lines.

5.3 Hybrid simulation

In the real world rigid objects and soft objects coexist, and in many cases within single objects, for example an umbrella contain both rigid and flexible components. As we have already seen many animation and the few VR systems which currently exist perform either rigid or flexible body simulation, few will enable a natural marriage of the two. In this section, an undersea simulation example is used to show how effective this combination can be.

5.3.1 Undersea simulation

For this particular simulation, we make use of the boids (c.f. §5.2.1) and chain examples. Seaweed is simulated using both flexible and articulating chains; these are constructed with articulating nodes of four bodies all hinged at the base (c.f. François Faure [Fau99]). Each rigid body then has a particle system based strand attached. Another variety of seaweed is simulated using a single articulated chain only. The benefit of this combination is to reduce complexity in terms of solving large numbers of unknown forces for each strand. Gravity acts upwards rather than downwards to simulate buoyancy and the simulation is choreographed by activating and deactivating force functions at certain time intervals.

Notice that a fish is detached in Figure 5.21 (cf. undersea.mpv and undersea.qt on CD) by deactivating the force function between it and the rest of the shoal. A new force function is then activated between the seaweed and the lone fish, which then dives towards the seaweed where it lingers for a time thus simulating feeding. Another temporal event then separates a segment of seaweed and combines it with the fish particle. The force function between the seaweed and the fish is deactivated and the one between the fish and shoal is reactivated. Notice how the fish with the segment of seaweed now has difficulty keeping up with the rest of the shoal due to its change in mass. Finally another temporal event occurs, separating the fish from its segment of seaweed allowing it to properly rejoin the shoal while the seaweed drifts away.

This simulation has been used to illustrate the way in which articulating and particle based models can be intermingled, together with the idea that a variety of different force functions can be constructed, activated and deactivated as desired. Also, the use of temporal events to choreograph the simulation was illustrated.

Thus-far we have only shown simulations controlled by scripted events, but naturally we wish to interact with them directly and affect the outcome of the

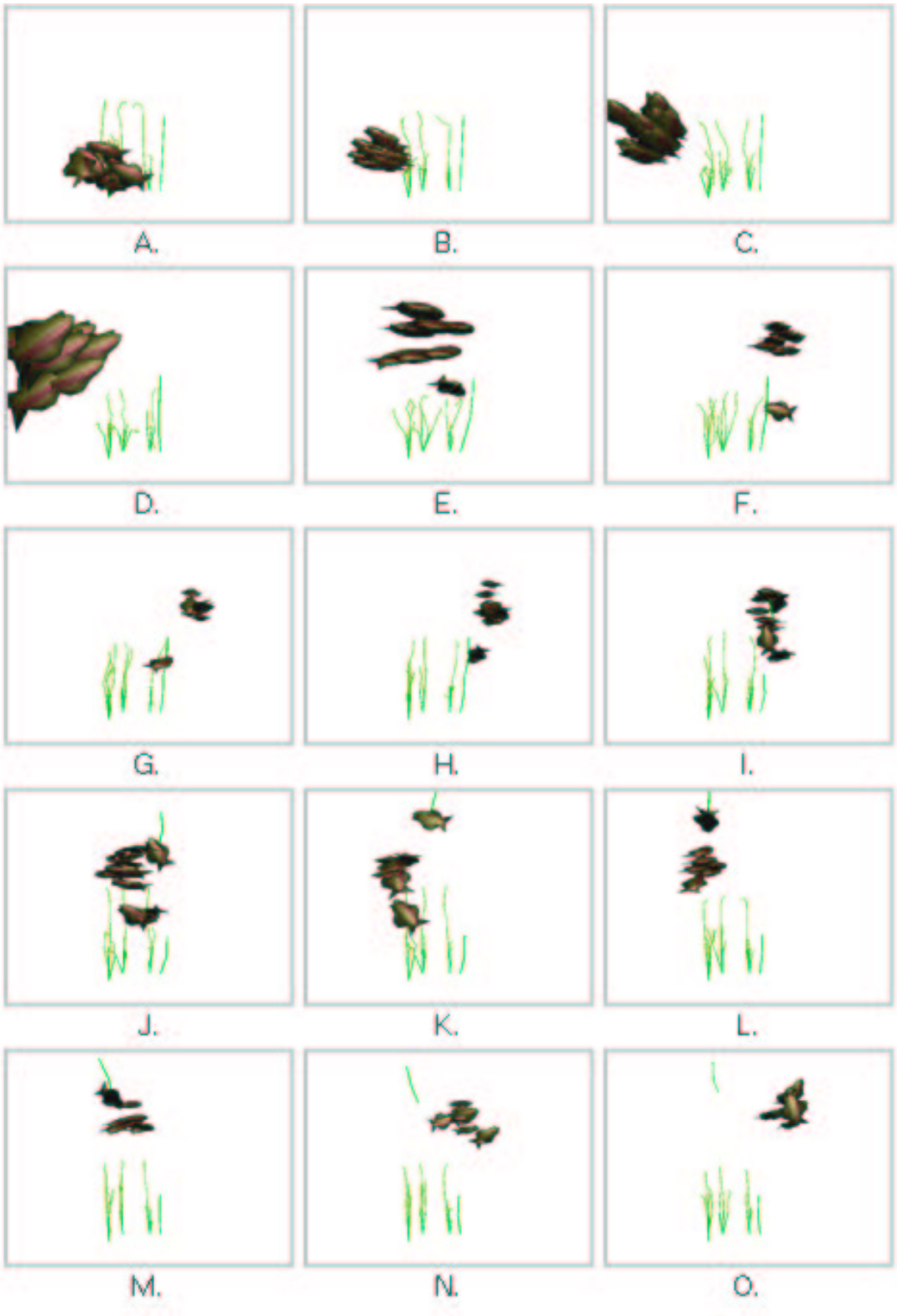


Figure 5.21: A shoal of fish in an undersea simulation

simulation. For this reason we will now consider interactive scene manipulation.

Code breakdown: Scene Setup: 50 lines. Presentation: 60 lines. Simulation/Behaviour/Display: 150 lines.

5.4 Interactive scene manipulation

VR is about providing natural interfaces for human computer interaction in a highly visual context. If the simulations are to be interactive they must not only be computed at interactive frame rates but the user should be able to manipulate them. In this section we describe the enhancements made to the chain and Newton's cradle examples to facilitate interaction with the scene using a standard mouse. It is inappropriate to revisit all the examples covered so these were chosen because they illustrate different aspects of user interaction with the scene. All the examples described above could easily have been modified to include user participation.

5.4.1 Chain revisited

This particular example is used to illustrate the power of the `combine` and `separate` routines to interactively dynamically restructure a scene. The chain example was chosen because it is quite simple and so the kinds of user invoked behaviour which have to be programmed are relatively trivial.

A mouse callback was registered to be invoked when the middle mouse button is pressed or released. If a link in the chain is selected then the particular one chosen is identified through a simple intersection test and a force function is constructed between it and the mouse particle. Once a link has been selected it is coloured red by convention as shown in Figure 5.22 (cf. chain.mpv and chain.qt on CD), to indicate to the user that a relationship exists between it and the

mouse. The link can be dragged around by the user via a force function, while the simulator maintains the constraints intact. This means that the link will not actually follow the mouse exactly, rather it will do its best to move in the direction of the force.

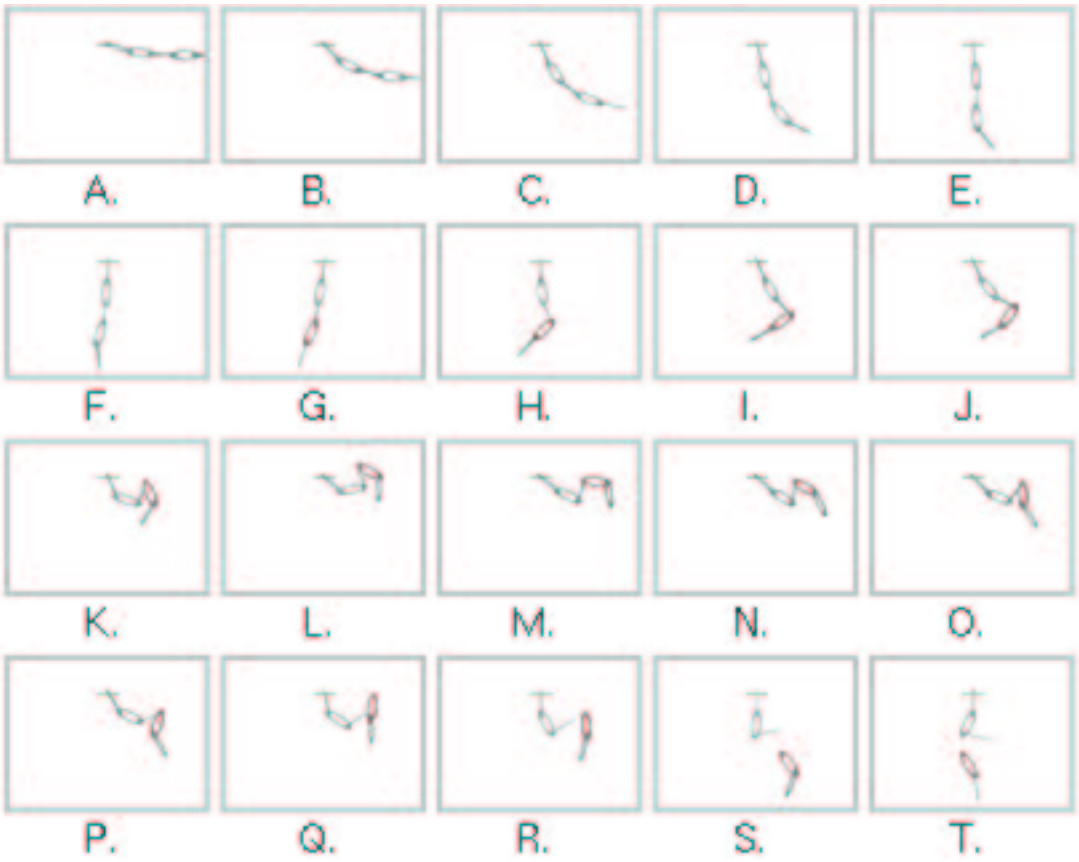


Figure 5.22: User interaction with the chain model

Two further actions on keyboard events are registered. Pressing the `b` key breaks the chain by invoking `separate` on the selected link and the one above it. On the other hand, provided the body is detached, pressing the `c` key invokes combine on the selected body and its nearest neighbour. Subfigure G in Figure 5.22 shows the fourth link from the top coloured red as it is dragged up and right. It is then detached from the chain in subfigure Q.

Code breakdown: Scene Setup: 20 lines. Presentation: 20 lines. Simulation/Behaviour/Display: 130 lines.

5.4.2 Newton's cradle revisited

In the Newton's cradle example, ideally the user ought to be able to select a ball and raise or lower it while the motion of the other balls is correctly simulated around it. There are subtleties which must be taken into account; for example selecting the centre ball in a cradle (at rest) consisting of seven balls and dragging it left should result in a group of four balls being raised, leaving three balls stationary. Similarly dragging the ball to the right should result in a different set of four balls being raised. So from this it can be seen that not only the selected ball but the direction in which it is pushed will affect the number of balls in each group.

A suitable test has to be implemented to decide how to split groups into subgroups as a result of moving balls. In the scripted case, the initial number of balls in each group were explicitly coded. If a user is responsible for selecting combinations of balls to lift, there is no way of predicting how many balls should be in each group, when the user may choose to intervene or the direction in which a group will be pushed.

The nature of the test to split groups involves considering either of the two following cases:

- Held ball is pushed left and is not the rightmost ball in the group, in which case the split occurs to the right of the held ball.
- Held ball is pushed right and is not the leftmost ball in the group, in which case the split occurs on the left of the held ball.

Another issue which must also be addressed is how to handle a collision with a group that is being dragged. Should the groups be swapped as has already been

described, or should the two groups be consolidated into one? This issue really addresses the natural expected behaviour of a Newton's cradle. Both choices described above appear to be plausible but which is better or appears more natural? The decision is not a clear cut one and has much to do with anticipating what the user really intends to achieve from selecting a ball. We chose to opt for consolidating the group for two reasons; one being that previously there was no means for consolidating groups. This meant that in the worst case scenario each group could consist of one ball adding a significant performance overhead. Secondly, if a user is holding a ball about to face a collision they intend to do something with it, the only real actions which can be performed are to push a ball left or right so consolidating the groups seems the natural thing to do.

Appropriate behaviour was attached by registering a callback to activate when the middle mouse button is pressed or released. If pressed, the selected ball (if any) is determined by an intersection test which takes the viewer's line of sight together with the position and orientation of the cradle into account. Once a ball is selected, the user is able to drag it around as desired. This behaviour is achieved through the use of a force function constructed between the selected ball and a particle which tracks the mouse. Notice that in Figure 5.23 (cf. newton.mpv and newton.qt on CD), by convention the selected ball is coloured red to convey this information to the user.

In subfigure B the two rightmost balls are raised and released in subfigure G. Subsequently the three rightmost balls in subfigure K are also raised and then released in M. Note how both sets of balls are involved in a collision with the stationary group containing two balls in the centre. The number of balls in each moving group swap upon collisions. From subfigure T onwards the cradle is spun by the user.

Adding user interaction almost doubled the size of the script due to the wide

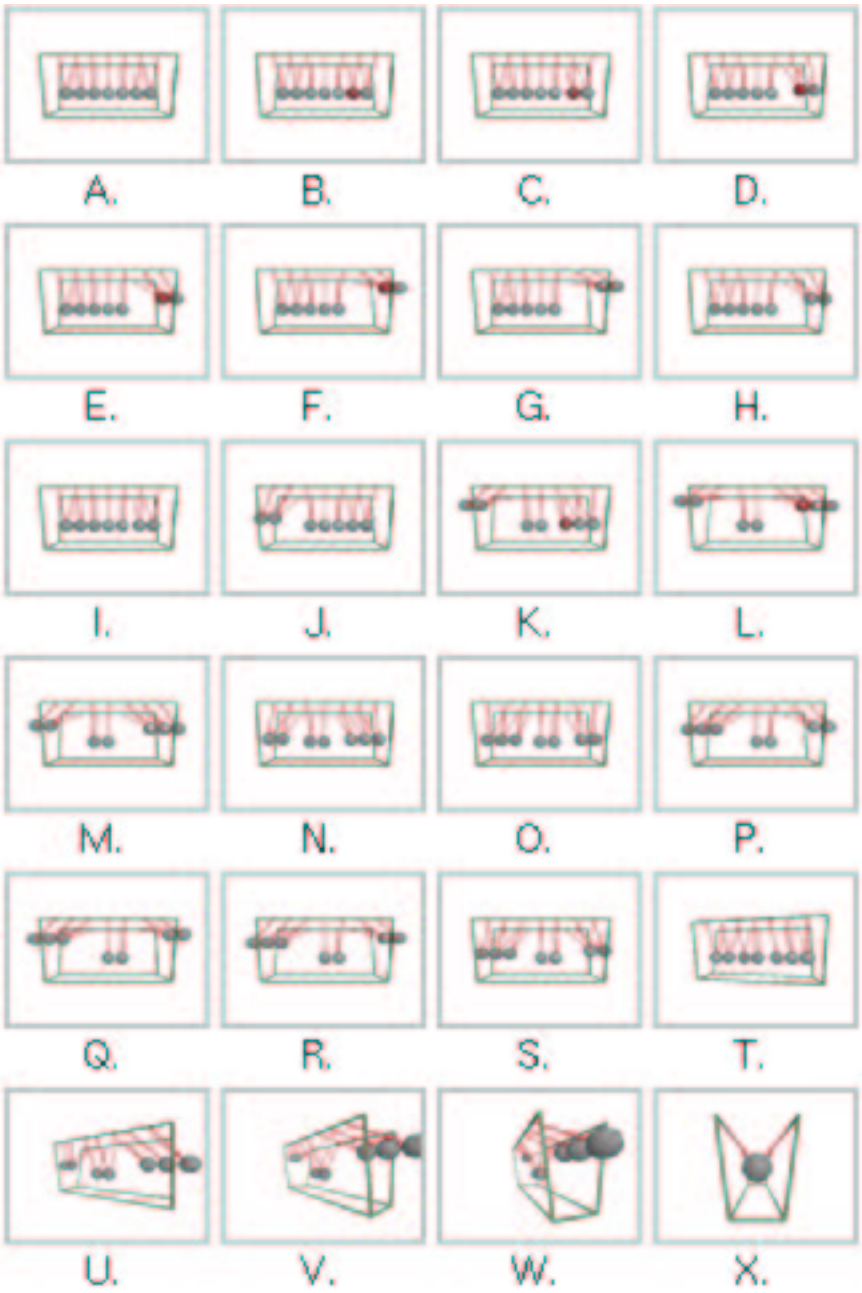


Figure 5.23: Interactive simulation of a Newton's cradle

range of possible scenarios, but given that the Newton's cradle script is relatively short, in real terms this is very little. The extra effort was deemed to be worthwhile as user participation in the simulation became natural. Most people who have tried the Newton's cradle example found the interaction to be responsive, natural and enjoyable. Moreover, from a programming point of view, the extra code involved consisted of case handling logic rather than complex algorithms.

Code breakdown: Scene Setup: 30 lines. Presentation: 60 lines. Simulation/Behaviour/Display: 300 lines.

5.5 A simple VR application

A simple VR walk-through application consisting of a locally available room model (Advanced Interfaces Group laboratory) was implemented within the Iota framework. This was used to provide context for the Jacob's ladder and Newton's cradle models. Both models performed well within the context of a large model that could be explored. The user was able to interact with both toys in the laboratory. The Jacob's ladder can be seen to appear to be on a computer monitor in the environment, as shown in Figure 5.24 (cf. aig.mpv and aig.qt on CD).

The Newton's cradle is placed upon a desk in the laboratory, as can be seen in Figure 5.25.

Both simulations coexist satisfactorily in the environment. Furthermore they provide a user with an interactive task in an otherwise static model.

Code breakdown: Scene Setup: 150 lines. Presentation: 160 lines. Simulation/Behaviour/Display: 490 lines.

Figure 5.24: A Jacob's ladder in the AIG lab

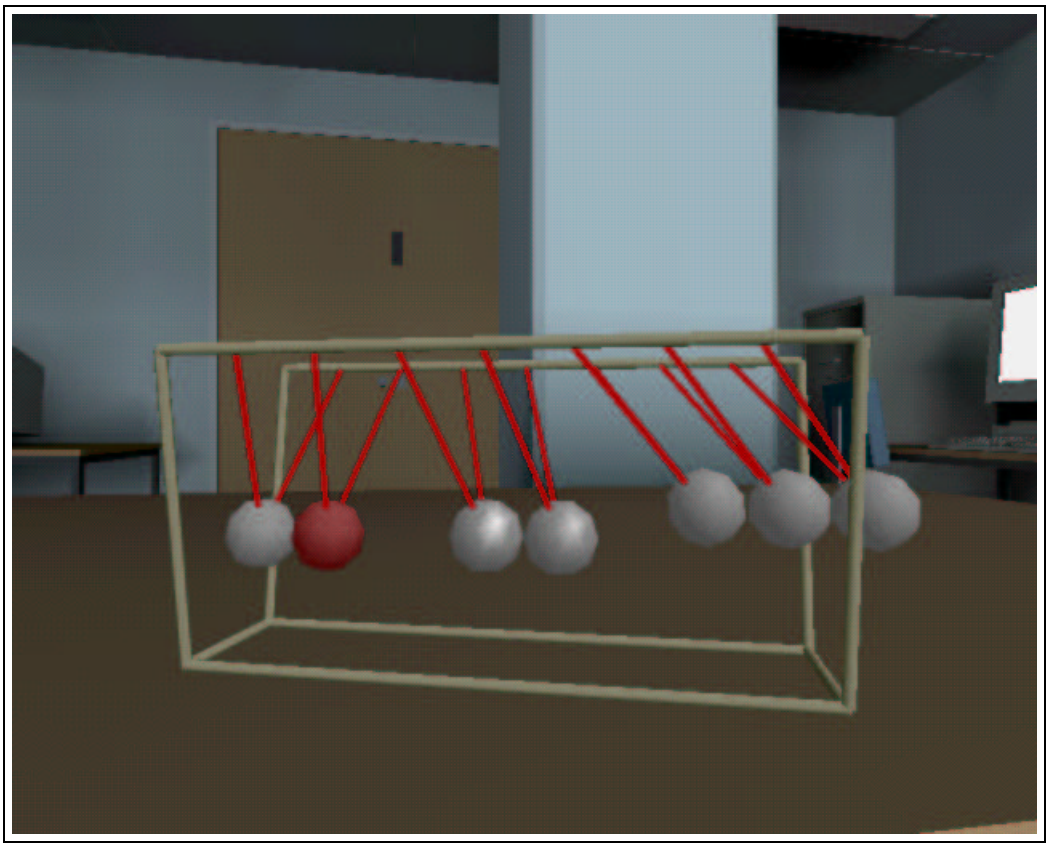


Figure 5.25: A Newton's cradle in the AIG lab

5.6 Performance evaluation

A quantitative impression of the performance of case studies can be achieved by using a profiler to gather statistics, and so obtain a measure of the amount of processor time spent in specific functions. The profiler delivers very accurate but low level statistics about the thousands of subroutines which are called, so to provide more meaningful results they have been post-processed to attribute them to individual components of Iota. One of the benefits of this experiment is that it is now possible to obtain an impression of the performance penalty paid for using a high level scripting language. Furthermore the performance improvements possible through the use of specific optimisations can also be seen. Performance data presented in this section was obtained using Silicon Graphics' 'SpeedShop' software on an Indy with a MIPS R4400 processor and R4000 floating point unit. It is important to note that the following charts only show the contribution of listed components to less than twenty five percent of the total processor power. The remaining seventy five percent (approximately) which is not shown was devoted to other processes, in particular rendering and synchronisation delays.

Since each simulation can vary dramatically in terms of simulator involvement, profiling was carried out for the Newton's cradle and Jacob's ladder examples. These simulations demonstrate widely differing aspects of the Iota approach and are in fact at two opposite extremes; the Newton's cradle being largely ad-hoc and the Jacob's ladder being predominantly physically based. As such, profiling data should exhibit significantly different characteristics.

The Newton's cradle minimises the use of the simulator by reducing the problem to single underlying pendulums for each group in the cradle. The majority of the logic is coded in Perl and so can be seen to play a dominant part in processor usage, as shown in Figure 5.26. SpeedShop was unable to attribute twenty four percent to any specific component, but the general picture is still clear. It is

apparent that for this model Perl and the simulator represent a large component but of a small pie. To reinforce the context within which these statistics are presented, the 45% CPU utilisation is out of a pie chart which captures 25% of the CPU time, so represents 1/8 CPU time.

Another possible measure of performance can be obtained by examining the frame rate at which the case studies were rendered. It was found that the case studies were rendered at between 15 to 75 frames per second (FPS) on a 366MHz Pentium II Laptop computer and marginally slower on a Silicon Graphics Indy. The simulations at the slower end of the frame rate spectrum were either graphically intensive, as is the case in the AIG laboratory application; or simulator intensive, in the Jacob's ladder case study for example. Simpler case studies such as the Newton's cradle were rendered at 25 FPS. The simplest case studies such as the articulating chain and the rigid molecule were rendered at the top end of the range (75 FPS).

A possible reason for the round number of FPS typically achieved could be attributed to the fact that graphics libraries tend to synchronise each frame update with the scanline on the computer screen. Typically worldwide screen updates occur approximately 75 times per second. It is therefore not coincidental that the Newton's cradle case study is rendered at 25 FPS frame rate as this means that calculations and the screen update took more than a fiftieth of a second, but less than a twenty-fifth. The frame rate cannot be improved much until a frame can be computed faster than one fiftieth of a second, which is generally difficult, and unlikely to be solved by removing the scripting language. These types of benefits may come from reducing the graphical detail, for example simplifying the lighting model or rendering spheres using fewer facets.

In contrast Figure 5.27 on shows the result of profiling the Jacob's ladder simulation with a model consisting of six blocks. In this simulation ten unknown

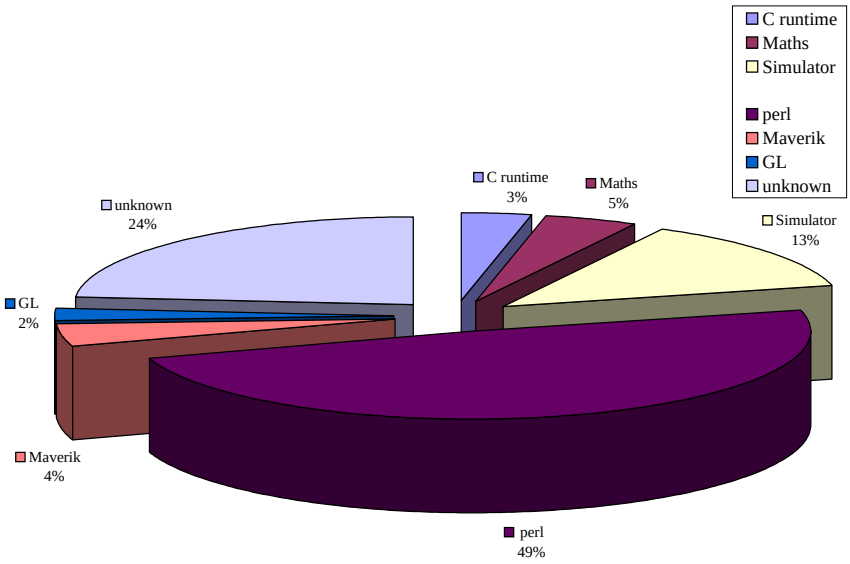


Figure 5.26: Newton's cradle profile results

forces have to be solved simultaneously for each time step. Thus it is not surprising that in this case, the simulator dominates. Perl is seen to also contribute significantly to the computation cost and this is to be expected as much of the logic to determine whether to flip-flop, lock or unlock hinges is carried out at that level.

The performance benefit of using articulates to structure the problem domain is illustrated in Figure 5.28. In this case two Jacob's ladders, one with three and the other with four blocks, were simulated. It can be seen that solving ten unknowns as one set of six and another of four is not as computationally intensive and that the simulator is now less dominant. Interestingly though, a matching significant gain in frame rate is not achieved. This reinforces the earlier point that to improve the frame rate by a notable factor is far from trivial.

Although Perl has been shown to contribute to at least a quarter of the pie

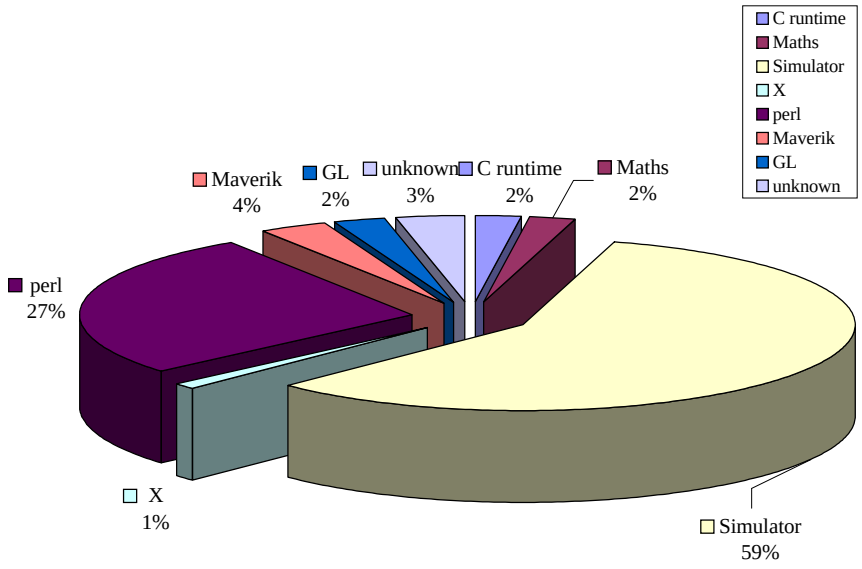


Figure 5.27: Jacob's ladder profile results

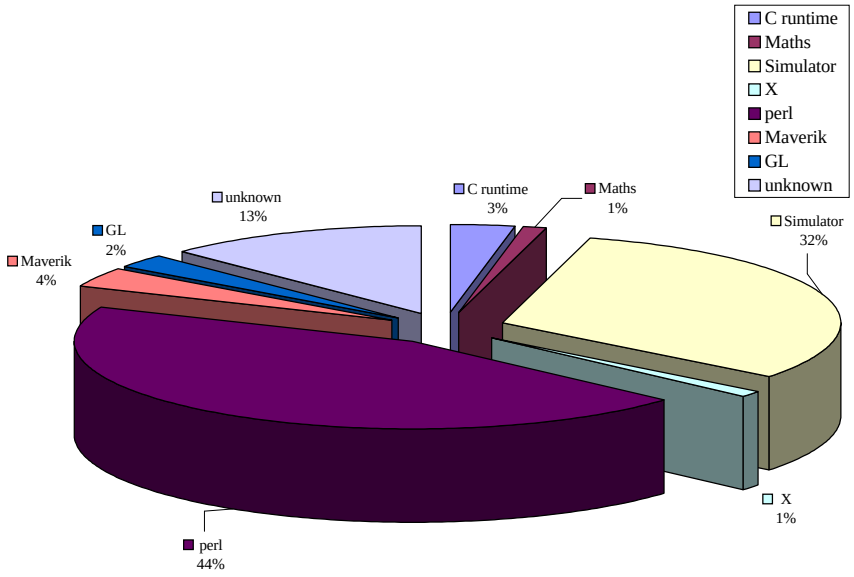


Figure 5.28: Profile results for a pair of Jacob's ladders

(in the best case), this cost is considered to be acceptable within the context that the main processor intensive task is still rendering. Small scale simulations are carried out in only twenty five percent of the processor's total usage and around three quarters of that is attributable to both Perl and the simulator.

5.7 Evaluation of division of coding effort

To convey an impression of the complexity of each case study a division of the coding effort into three categories (Scene setup, Presentation and Simulation/Behaviour/Display) was given. This exercise shows two things in particular; firstly the ease in which simulator functionality is invoked indicates that the developer does not require an in depth knowledge of physics. Secondly the amount of code necessary to implement user interaction in the case studies was significant. This is due to the fact that currently the Iota prototype supports little high level functionality for selecting objects, tracking the mouse and similar support for user interaction and this problem can be addressed by providing a custom module.

Implementing user interaction with the Newton's cradle and chain case studies shows a significant increase in the Simulation/Behaviour/Display category. For the Newton's cradle this was found to roughly double the coding effort in this category. The chain example required about 120 lines more code to enable user interaction and assembly/disassembly of the model. Broadly speaking this means that implementing user interaction in the current framework can be considered to add approximately 100 to 200 lines of additional code.

In the simple case studies a greater contribution to coding effort was found to be made by the Presentation category which involved setting up MAVERIK colours and customising the default graphical representations. Whereas in the more complicated case studies the dominant coding effort was found to be logic

describing the behaviour of the models.

The Simulation/Behaviour/Display category was grouped together because they are interrelated but most of the coding effort in this category can be attributed to the logic describing the behaviour of the models. Only one line of code could be directly attributed to a call to the simulator and typically eight lines was contributed by the display part of this category.

So given the amount of complex simulation involved in each of the case studies and in particular the complexity of the Jacob's ladder case study, it is fair to say that in many cases a surprisingly small amount of high level code has to be implemented by a developer.

5.8 Evaluation of user perception of motion

Slater states in his response to the Witmer and Singer questionnaire [Sla99] that he is loathe to use questionnaires but that currently there is little alternative for measuring presence. We cite this argument as a basis for using a subjective questionnaire to assess whether or not people consider the motion of objects simulated in the case studies to be plausible. Measuring plausible motion is in many ways similar to the problem of measuring perception because both may depend on the knowledge and experiences of the observer. Although this approach can be argued as being subjective it does give an insight into the value of supporting physically based simulation in an interactive VR context. An impression of users' emotional response to each case study can also be gained.

Fifteen test subjects were given an opportunity to scrutinise and in the relevant cases interact with each case study. For each case study the question: "What was your impression of the motion?" was asked and the resulting remarks recorded. Tables showing the remarks made by each of the volunteers are shown in Appendix F.

Test subjects commented to varying degrees about the motion they had seen but the general consensus was that for each case study the motion was indeed believable. The only case study which caused some conflicting opinions was the Bucky ball simulation §5.2.3. Three of the volunteers said that the simulation gave them the impression of the viewpoint being moved in and out rather than the structure pulsating. Another three likened the simulation to a heart beating, one person was reminded of the universe expanding and shrinking. The remaining eight volunteers considered the motion to be 'correct' or at least plausible.

The interactive examples generated a noticeably greater degree of enthusiasm amongst subjects which can be seen from the comments made. On average the tests took between half an hour and forty five minutes each depending on the enthusiasm of the particular volunteer.

5.9 Summary

In this chapter we have shown that a wide range of different kinds of simulation are possible within the Iota framework. The power of dynamic restructuring is used very effectively in the Jacob's ladder example and the benefit of the ad-hoc and physical modelling combination is illustrated in the Newton's cradle. Moreover, user interaction is relatively easy to incorporate and significantly enhances the simulation. It has been suggested that participation improves the user's perception of a simulation. Both the Jacob's ladder and Newton's cradle simulations are demonstrated in the context of the AIG laboratory model. Finally an evaluation was carried out in three parts. First a comparison of the performance of the lightweight simulation of a Newton's cradle and the simulator intensive Jacob's ladder. Second user trials were carried out to assess whether or not the motion of objects in each case study was plausible, and third a discussion of the level of difficulty of implementing each case study was presented with an analysis of the

type of programming required to implement applications in Iota. The findings of this evaluation are that there is a performance penalty to be paid through the use of Perl as a high level development language, however implementing applications can be easier. The majority of test subjects, from a sample of fifteen, considered the motion exhibited by models in each case study to be plausible.

Chapter 6

Further work and conclusions

In order to summarise the scope of the Iota framework we begin this chapter with a discussion concentrating on the classes of applications that could be implemented within the framework. Following this, possible suggestions for future work are made, and finally this thesis is concluded with a discussion of the relative merits of the approach and a summary of the elements within the framework.

6.1 The Iota framework: Application areas

The main application areas of the Iota framework are categorised as physically based modelling in simulation, visual, educational applications and entertainment applications. Each of these is described with examples to give a clearer impression of the scope of the framework.

In the category of physically based simulation it is fair to say that many particle based models can be simulated within Iota so the typical kinds of particle based simulations used for modelling crowds, flocking behaviour, fire, gases and other similar effects can be simulated. These models have been used in serious applications, for example Colt VR [COL] use particle based approaches for simulating evacuation procedures in buildings. They use this approach to analyse

the ability of a crowd to escape from a building during a disaster such as fire or flooding.

A number of serious applications can benefit from some level of physical simulation for visual purposes. In particular a greater impression of realism or presence can be achieved by providing rich behaviour through the use of simple physical simulations. For example applications for designing large models of buildings, off shore rigs, cities, theme parks and other similar installations could benefit from physical behaviour for modelling a variety of real world objects.

Educational applications for interactively teaching elementary physics could be built within the framework. This particular suggestion was an unsolicited comment made by a participant F.15 during the user perception tests described in the previous chapter.

Physically based simulations of the type which can be carried out within the Iota framework could be used in VR entertainment applications. MathEngine [Woo99] implement similar simulation techniques for use in gaming and entertainment applications.

Finally, Iota's simulator is not appropriate for use in precision assembly and maintenance tasks or applications requiring robust satisfaction of a large number of constraints within the same articulate. The simulator component currently used was designed for use in desktop VR applications using physically based modelling and so does not contain a variety of different techniques for solving overconstrained, underconstrained or ill conditioned systems. It is this that limits the scope of its applicability rather than any failing in the design of the framework itself.

6.2 Further work

The development of the Iota framework has provided an answer to the question: what are the core elements required for physically based simulation in VR? However, many questions have been raised as a result of the work described in this thesis. Further work can be conducted in one of two general categories; development and analysis which are described in the following section. In terms of development a number of issues are worthy of consideration, in particular model generation and simulator functionality.

Another area worthy of further research is physically based simulation in a distributed VR context. The work described in this thesis has only considered single user VR but many interesting issues would have to be handled in the distributed case. For this reason, further work is considered in two subsections, one dealing with general enhancements and the other with distributed VR.

6.2.1 General enhancements

Model generation is currently difficult so interactive generation of models to simulate is one possible approach for relieving some of the scripting burden from the programmer. Although a more user friendly approach might be a combination of procedural and interactive model generation, as purely interactive model generation could potentially be tedious. Much thought would have to be given to the mechanisms by which a user may interactively build a model and the type of building blocks to make available. The disadvantage of making model development easier is that much of the current flexibility in Iota would be lost and the user would be forced to work with the tools given. Thus it can be argued that the freedom for innovative design is worth the inconvenience of handcrafting models; necessity is the mother of invention.

The simulator functionality is currently rather basic, but the simulator is extensible and alternative solvers, orientations of bodies and a variety of different types of constraints could be incorporated. Although the simulator is fairly simplistic, it has been sufficient for the examples in this thesis. It can be argued that the simulator contains enough functionality for physically based modelling in VR, as long as the objective of simulating only plausible behaviour remains true.

Finally a number of different types of analysis could be carried out. Firstly it would be interesting to release the framework to a number of developers to see how easy or difficult it may be in practice to develop an application within Iota. This was considered to be an inappropriate exercise within the scope of a PhD largely because the Iota implementation is a prototype. Secondly it would also be interesting to analyse how much performance gain is achieved through complexity management of models and adaptively adjusting the level of detail of simulations. Finally it would also be a good exercise to analyse how quickly a new component could be integrated into the framework by a novice developer.

6.2.2 Physically based modelling in distributed VR

One of the more recent challenges in the field of VR is represented by the area of distributed VR. There are still many issues which need to be resolved in terms of where the master representation of the VE's model of reality exists together with how and when information is distributed to other systems. Furthermore, questions exist in terms of how to handle delays across networks and synchronisation of users' views. Many of these types of problems have been addressed in the DEVA framework implemented by Pettifer [Pet99]. He provides a metaphor which maintains consistent world behaviour. The details of the specific rules used to govern the physics of the world can be coded within his framework. The work

has been biased towards maintaining consistency and handling issues related to distributed VR and Pettifer has not as yet considered the specifics of physically based modelling.

To be able to simulate complex behaviour of objects in a distributed context would require resolution of still unanswered research questions. Potentially, the concepts in Iota could be applied within the DEVA framework but as the two systems were developed independently, this would involve significant implementation effort. Incorporating physically based modelling adds a large degree of complexity to the problem of distributed VR.

Distributed VR generally uses a client/server model, where the server stores the VE, and clients provide I/O to it. There are three alternative methods which could be used to implement this type of architecture. The first and simplest is to maintain synchronisation across server and clients and the simulation runs at the speed of the slowest client. This approach has the benefit that little drift occurs in solutions as they are held either in one simulation (server), or synchronised identical simulations in clients. The main disadvantage is that the slowest machine connected controls the rate of simulation.

Another extreme is to maintain both client and server asynchronously, therefore potentially allowing clients to be out of step with each other. This is a satisfactory approach for representing avatars coexisting in a VE for example, because the server will maintain an up-to-date world state. However it is difficult to fit physically based modelling into this scenario because clients may run at different frame rates which do not coincide well with simulation timesteps. State drift which may result in each client's simulation diverging can become a significant problem as each simulates the scene at different points in time. Furthermore, complex user interaction with the scene would present significant problems as the implication of any one person's modifications would be difficult to interpret in

the context of other clients' state.

A possible solution to this problem would be to compromise and to some extent decouple the clients from the simulation in the server. The simulation would still be carried out in discrete timesteps, but the client would need to correctly handle input and output for missed timesteps. A simulator would then only accept input from clients when they can supply it, and send screen updates to clients which are ready. Thus interaction with the scene would not be a problem in such an architecture. An impression of asynchronous behaviour is given to clients, but the simulation is maintained synchronously. Ultimately the simulator would have more control over clients than in our single user implementation. A workable approach to implementing this type of system would be to make the client download a script representing the scene from the server and run it as the client's local perception but exchanging data with a single simulator in the server. This fits well with the Iota approach of using scripting to provide high level control and communication.

6.3 Conclusions

We conclude by advocating the framework proposed in Chapter 4 for physically based modelling in VR. This framework is designed for use at the middleware level by application developers as such it is designed with maximum customisability and flexibility in mind. The Iota framework is not perfect and the relative merits of the approach are described in the following sections.

6.3.1 Relative merits of Iota

Iota is the realisation of an approach towards providing sophisticated behaviour in VR, and represents a possible strategy that could be applied to wide ranging

applications. For example, in large models of structures such as oil rigs [CHK98] it could be appropriate to simulate the behaviour of chains and pulleys, virtual actors transporting complex objects from one location to another or even helicopters landing and taking off from a helipad. These types of sophisticated behaviours add complexity to the VE and it is difficult to know the demands imposed on a general simulator in advance.

The main disadvantage of this approach is the added performance penalty paid for making calls to C/C++ through a high level scripting language but this is considered to be a worthwhile trade-off as the benefits gained outweigh the small performance cost. Furthermore, as computer hardware capabilities are rapidly increasing, it is argued that this will not be a major issue for long. Another source of problems is Iota's solver; it was intended to be general purpose and functional but not the main focus of research effort and as such is far from optimal. A number of issues would need to be resolved before Iota's solver would be appropriate for use by a commercial organisation, in particular an increase in the diversity of constraints and joint types possible would be desirable. Moreover, the convergence of the solver could be further improved by incorporating alternative solution techniques, although to be fair it is relatively robust (compared with Diffpack) for small scale simulations. Finally, although concepts are provided for interaction with the simulations little supporting high level functionality has been implemented within Iota. Certainly significant improvement in terms of the API for implementing simulations would be gained by providing suitable methods for handling user interaction.

Although there are a number of disadvantages with the specific prototype implementation used to illustrate the Iota framework, it is argued that the advantages more than compensate for them. Firstly the most novel and powerful advantage is gained from the ability to dynamically restructure scenes not least

because it provides a mechanism for a simulation to cope with user interaction. Secondly, the combination of particle and articulated rigid body systems provides a diverse range of behaviours. Consequently, the implementation is sufficiently general purpose to be able to simulate a wide range of physical systems which can be modelled using such methods. Finally, Iota is both extensible and flexible meaning that the programmer is not forced into following a strict prescribed convention. If programmers want to supply additional functionality then they are free to do so. Also if an approximate model is considered more suitable for a particular simulation then this is not discouraged.

6.3.2 Dynamic restructuring

One other important issue not often catered for in VR or even animation systems is dynamic restructuring of scenes. Iota's simulator provides mechanisms to restructure scenes, primarily through `combine` and `separate` routines. These cause the underlying scene graph to reorganise itself, maintaining entities which do not interact in independent containers (articulates).

An alternative approach is presented by Witkin et al. [WGW90]. Their approach divides the world of objects into separate entities possessing attributes relating to their state and the constraints imposed. Each entity must be able to report these, together with the constraint force. From this information they dynamically construct the constraint equation and the equations of motion governing the system, these are then solved using traditional methods.

The advantage of Iota's dynamic restructuring is that the solver is not involved in reorganising the scene graph. Quite simply, the scene is restructured and the problem reposed to the solver. This approach means that the new problem does not lead from the previous problem. This is indeed a suitable choice as a `combine` or `separate` does dramatically alter the simulation. Dynamically restructuring

the scene can be exploited to a significant degree in VR, even as far as the simulator `system` level. A simulation could centre around and propagate with the user. In other words, as the user moves through the environment, so could the simulation. Objects could be dynamically added and removed (or activated and deactivated) from the current `system`, `articulate` or `body` thus significantly managing the complexity of simulations. Furthermore, simulations could become more sophisticated as the user approaches them and could be approximated while the user is too far away to appreciate any subtleties. The concepts required to perform this type of scene management already exist in Iota although some further development would be required to provide such general purpose functionality.

6.4 A Summary of the Iota approach

We argue that to successfully simulate physical behaviour of objects in VR requires a flexible high level development environment which:

- Does not tie the developer to a particular way of achieving the result
- Provides a robust physical simulator to use if and when desired
- Permits ad-hoc modelling within the context of physics
- Provides a mechanism for dynamically altering the scene graph in order to handle user interaction
- Has control over a VR kernel

These factors should be incorporated into the framework shown in Figure 4.1. The power of this particular approach lies in the fact that a high level language is used to drive the VR application. In this framework, the simulator and VR system are seen as providing low level services and can be viewed as the kernel of the system.

An exchange of information between the simulator and VR kernel occurs through the scripting language, having the benefit that both could be developed independently and significant customisations could be made at the scripting level. A direct exchange of data occurs between the scripting environment and the simulator or VR kernel when a service is required by the scripting environment. The callback mechanism enables the simulator or VR kernel to request a service from the scripting language, such as to compute a bond force or act upon a mouse event.

This framework also allows for a combination of ad-hoc and physical modelling, as the scripting language provides a powerful and flexible medium for describing ad-hoc models to simulate, only calling upon the simulator to advance the frame. This contributes to maintaining consistency in the simulation. In particular by making a simulation more manageable there is less likelihood of the solver failing to converge. Naturally, if a more physically based model is desired then further services can be requested from the simulator.

Within Iota the simulator's ability to dynamically restructure scenes enables interesting behaviour and valid scene graphs to be maintained even after the user has changed a model. This has been shown to be a very powerful mechanism for controlling a simulation as illustrated in the Jacob's ladder example. Furthermore, since dynamic restructuring reposes the problem of simulating the model, a mechanism exists for the user to intervene and dynamically break or attach components.

It is acknowledged that the behaviour of models can be difficult to describe but this is a problem particular to the nature of physically based modelling, and as such is not a failing in our approach. Once a model had been conceived, implementing the logic to govern a simulation is comparatively easy. Due to the power afforded by the flexibility of using high level scripting languages, their use

is highly recommended.

Finally, the Iota approach is demonstrably flexible and powerful and this statement has been backed up by the results and discussions presented. We have successfully shown that physically based simulation adds reasonable interest to a VE and as such is worth the effort to provide. Furthermore, being able to dynamically manipulate the model and affect significant changes at runtime has been shown to be crucial to any general purpose physical simulator for VR.

Appendix A

Algorithm for computing an Iota transformation matrix

This section contains the algorithm used to compute transformation matrices based on any two arbitrary co-ordinates to apply to unit sized objects centred at the origin. To find a transformation matrix appropriate for transforming an object from MAVERIK's local co-ordinate system to a location in world space characterised by two points. It actually works by performing a series of transformations (centre at origin, align with axes and then a resize) from world space to the shape's local co-ordinate system. This matrix is subsequently inverted to give the desired mapping. It may appear unintuitive to perform these transformations in reverse order but it is in practice much easier to use existing MAVERIK functionality to align a shape with axes, rather than transform an aligned shape to an arbitrary orientation. A pictorial representation of this algorithm is shown in Figure A.1.

Algorithm 8 Algorithm to compute a transformation matrix for a MAVERIK shape

Require: endpoints `pos1` & `pos2`
Ensure: `pos1 != pos2`

```
1: matrix_return_value = unit matrix
2: distance = (pos2 - pos1)
3: distance *= 0.5
4: Direction move_by = pos1 + distance
5: pos1 -= move_by
6: pos2 -= move_by
7: matrix_return_value *= translation_matrix (-move_by)
8: roll_angle = atan2 (pos2.gety (), pos2.getx ())
9: rot_matrix = mav_matrixDef (180/PI*roll_angle, 0, 0, 0, 0, 0)
10: pos1 *= rot_matrix
11: matrix_return_value *= rot_matrix
12: pitch_angle = atan2 (pos1.getx (), pos1.getz ())
13: rot_matrix = mav_matrixDef (0, 0, 180/PI*pitch_angle, 0, 0, 0)
14: pos1 *= rot_matrix
15: matrix_return_value *= rot_matrix
16: stretch = unit_matrix
17: stretch.mat[2][2] = .5/pos1.z
18: matrix_return_value *= rot_matrix
19: matrix_return_value = mav_matrixInverse (matrix_return_value)
```

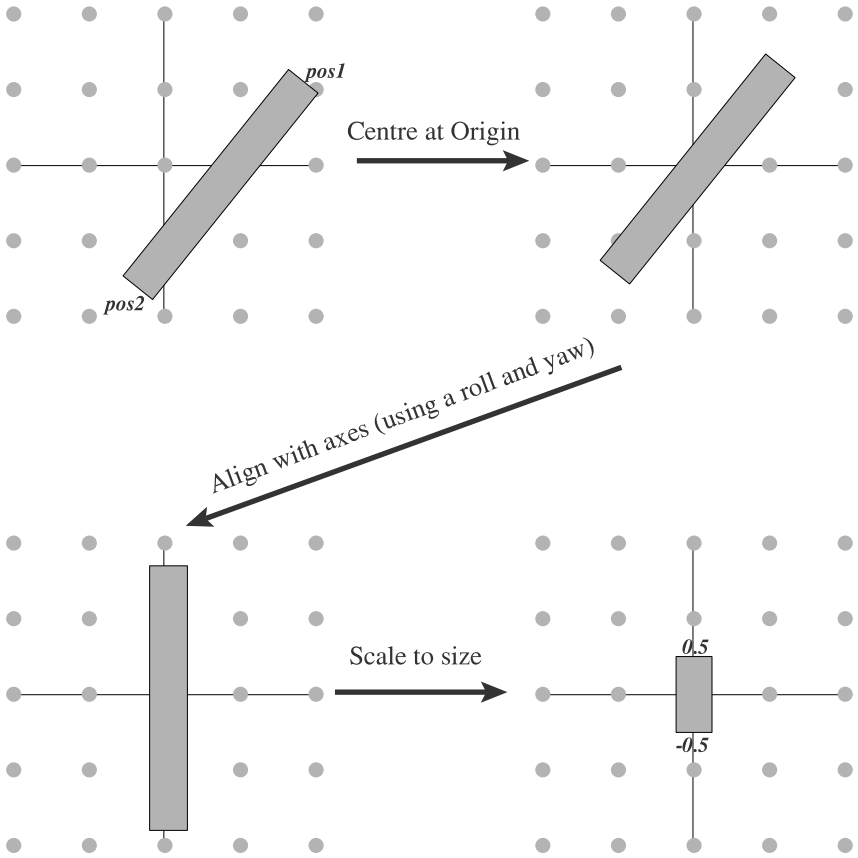


Figure A.1: Approach taken to compute an Iota transformation matrix

Appendix B

Callbacks background

This appendix provides some information on callbacks, their implementation and the additional functionality provided for creating Perl callbacks. Callbacks allow C programmers to gain some of the benefits of object-oriented (OO) programming without resorting to an OO language such as C++. The mechanism allows functions to be associated with specific instances of structures[1], allowing a clean interface between the core of the application (or system code) and the rest of the code.

A callback at its simplest is shown in Figure B.1; here the application registers a function to be called under some predetermined event, perhaps as an error handler. Setting and invoking the callback works by passing a function pointer through the Application Program Interface (API) to a function which retains it. Later another routine calls the retained address, resulting in a call to the function. It is usual for APIs to provide low level services which are called, but what is special about callbacks is that now the low level code uses callbacks to delegate work to *higher* levels.

The next example, shown in Figure B.2, creates a function specific to an

[1]This goes some way towards implementing OO principles. Strictly speaking OO requires support for *classes*, *objects*, *inheritance* and *polymorphism*. Callbacks provide the first two of these and to some extent the latter two, but not in a 'type safe' manner.

instance of a structure, akin to implementing a 'Class' in OO. A few variations exist on this theme, but only one is shown. Here a data structure is `malloc`ed which represents a class. This could take place in either the user-code or the low level code. The call to register the function passes the data structure and the function address, then the function address is written into the data structure. There is now a relationship between the data structure and the function. This process may be repeated any number of times with different instances of the same data structure and different functions. At a later point an action may be performed on a data structure which invokes a the callback. Performing the action on different instances of the data type will result in (potentially) different functions being called. It is often beneficial to pass the data type to the function enabling it to interpret the context it was called in and giving it access to other data available within.

The final example deals with extending this mechanism to allow existing C callbacks to have Perl callbacks mapped on top of them without modification to existing code. Figure B.3 shows how this is done: the C function registered with the API cannot be a Perl function so instead we write a function explicitly for handling and then forwarding Perl callbacks. It works by keeping a copy of Perl's function and data structure addresses keyed by C's data structure address and performing a further callback up to Perl with them. Obviously if the C callback does not pass enough context information to determine which Perl object and function to use, then this technique cannot be used.

It is now possible to have a mixed environment where some callbacks are satisfied by default C functions, others are satisfied by scripted Perl functions, and yet others are delegated by Perl back to predefined C functions where performance might be critical.

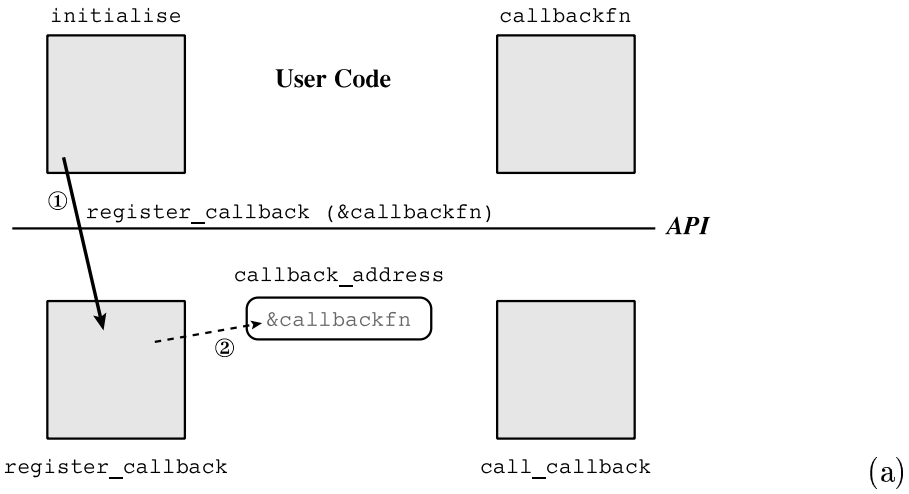
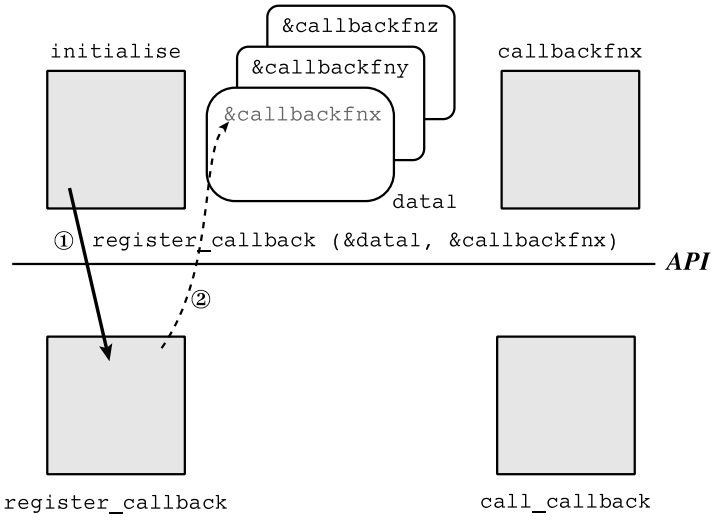
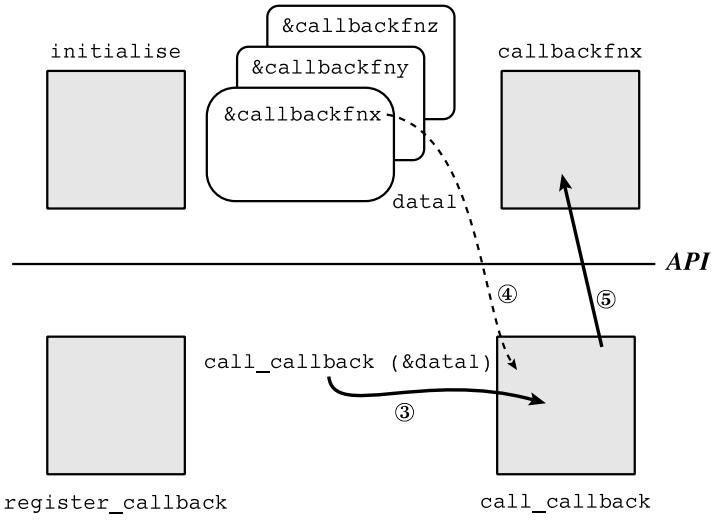


Figure B.1: A simple callback example. An application registers a function to be called under some predetermined event. Step 1: A user call is made to register a callback. Step 2: `register_callback` places an address in a variable. Step 3: At a later point an event causes the callback to be triggered by calling `call_callback`. Steps 4 & 5: The address is retrieved from the variable and called.



(a)



(b)

Figure B.2: Callback used to model OO methods. Step 1: A call is made to register a callback against a structure (object). Step 2: `register_callback` places the callback address in the structure. Step 3: The event is triggered and a call is made to `call_callback`. Steps 4 & 5: This routine retrieves the function address from the 'object's' data structure and makes the necessary call.

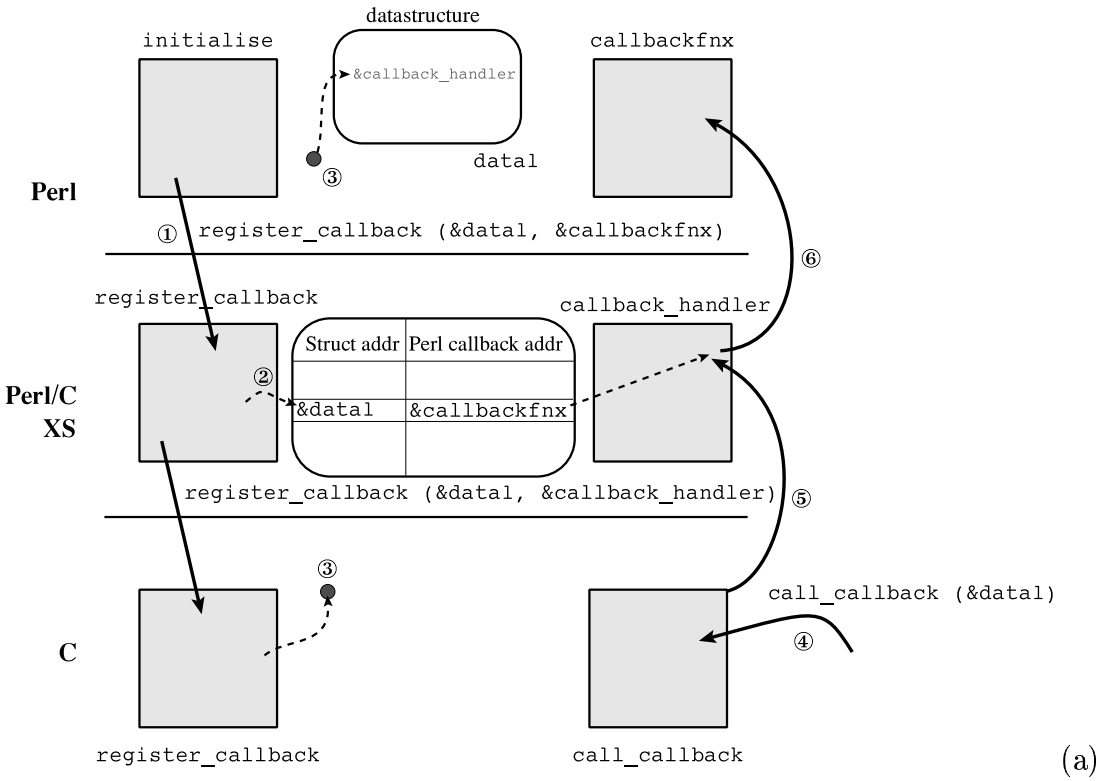


Figure B.3: Enabling callbacks into Perl. This time the callback is registered through a routine which performs some extra housekeeping before forwarding the callback information to the real `register_callback` routine. Step 1: Register the callback with a `register_callback` routine which actually resides in an XS file. Step 2: The object address and its associated callback are recorded in a hash table, and the C function to register the callback is invoked with the address of a redirector function `callback_handler`. Step 3: The C register callback routine records the callback address (always `callback_handler` if called from Perl) in the data structure. Step 4: A callback is invoked on the object. Step 5: Invokes the callback recorded in object data structure (`callback_handler` if callback was registered by Perl). Step 6: `callback_handler` locates and runs the real Perl function.

Appendix C

Derivations

Derivations are provided in this appendix for performing transformations about arbitrary lines, calculating the moment of momentum for a particle system and performing a singular value decomposition on a matrix (SVD). Finally some derivations to allow the implementation of a non-linear equation solver using the Newton-Raphson method are described.

C.1 Derivation of an Iota transformation matrix for rotating a point about a line

Equation C.1 shows the relationship between a point's position before a rotation ($\mathbf{r}$) and after ($\mathbf{R}$), about a unit direction vector $\hat{\mathbf{n}}$. It can be rearranged into a form given in Equation C.2.

$$\mathbf{R} - \mathbf{r} = \tan\frac{\theta}{2}\hat{\mathbf{n}} \wedge (\mathbf{R} + \mathbf{r}) \tag{C.1}$$

$$\mathbf{R} = \cos\theta.\mathbf{r} + (1 - \cos\theta)(\hat{\mathbf{n}}.\mathbf{r})\hat{\mathbf{n}} + \sin\theta.\hat{\mathbf{n}}) \wedge \mathbf{r} \tag{C.2}$$

If $\mathbf{R} = [X, Y, Z]$, $\mathbf{r} = [x, y, z]$ and $\hat{\mathbf{n}} = [l_1, l_2, l_3]$ are referred to the unit vectors

i, **j**, **k**, then a further rearrangement of relationship C.2 can be made to become

$$\begin{aligned}
[X,Y,Z] &= \cos\theta.[x,y,z] \\
&+ (1-\cos\theta).(l_1x + l_2y + l_3z).[l_1,l_2,l_3] \\
&+ \sin\theta.\begin{bmatrix} \hat{\imath} & \hat{\jmath} & \hat{k} \\ l_1 & l_2 & l_3 \\ x & y & z \end{bmatrix}
\end{aligned} \tag{C.3}$$

Equating corresponding components leads to

$$\begin{aligned}
X = &\quad [\cos\theta + (1-\cos\theta)l_1^2] && x \\
+ &\quad [(1-\cos\theta)l_1l_2 - l_3\sin\theta] && y \\
+ &\quad [(1-\cos\theta)l_1l_3 + l_2\sin\theta] && z
\end{aligned} \tag{C.4a}$$

$$\begin{aligned}
Y = &\quad [(1-\cos\theta)l_2l_1 - l_3\sin\theta] && x \\
+ &\quad [\cos\theta + (1-\cos\theta)l_2^2] && y \\
+ &\quad [(1-\cos\theta)l_2l_3 + l_1\sin\theta] && z
\end{aligned} \tag{C.4b}$$

$$\begin{aligned}
Z = &\quad [(1-\cos\theta)l_3l_1 + l_2\sin\theta] && x \\
+ &\quad [(1-\cos\theta)l_2l_3 - l_1\sin\theta] && y \\
+ &\quad [\cos\theta + (1-\cos\theta)l_3^2] && z
\end{aligned} \tag{C.4c}$$

These relationships can be put into a more compact form by the introduction

of the three matrices $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ which are defined by

$$\mathcal{A} = [a_{ij}] = \cos\theta.\mathbf{I} + (1 - \cos\theta).\mathcal{B} + \sin\theta.\mathcal{C} \tag{C.5a}$$

$$\mathcal{B} = [b_{ij}] = \begin{bmatrix} l_1^2 & l_1 l_2 & l_1 l_3 \\ l_2 l_1 & l_2^2 & l_2 l_3 \\ l_3 l_1 & l_3 l_2 & l_3^2 \end{bmatrix} \tag{C.5b}$$

$$\mathcal{C} = [c_{ij}] = \begin{bmatrix} 0 & -l_3 & l_2 \\ l_3 & 0 & -l_1 \\ -l_2 & l_1 & 0 \end{bmatrix} \tag{C.5c}$$

Using these matrices, Equation C.4 can now be written as

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathcal{A} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{C.6}$$

It has been necessary to adapt this equation, extending it to work with a rotation vector whose length represents the angle θ. This allows the result of a cross product to be input into the equation to be derived, and also reduces the equation to be a function of three unknowns from four. The length of a vector L,

$$\mathbf{L} = \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix}, \tag{C.7}$$

is given by $\sqrt{L_1^2 + L_2^2 + L_3^2}$ which corresponds to the angle θ. The unit direction

vector $\hat{\mathbf{n}}$ becomes

$$\hat{\mathbf{n}} = \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} / \sqrt{L_1^2 + L_2^2 + L_3^2} \tag{C.8}$$

$$= \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} / \theta \tag{C.9}$$

$\mathcal{A}$ becomes significantly more complex as a result, as shown in Equation C.10.

$$\begin{aligned} \mathcal{A} = & \cos\left(\sqrt{L_1^2 + L_2^2 + L_3^2}\right) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ + & \frac{1-\cos\left(\sqrt{L_1^2+L_2^2+L_3^2}\right)}{L_1^2+L_2^2+L_3^2} \begin{pmatrix} L_1^2 & L_1L_2 & L_1L_3 \\ L_1L_2 & L_2^2 & L_2L_3 \\ L_1L_3 & L_2L_3 & L_3^2 \end{pmatrix} \\ + & \frac{\sin\left(\sqrt{L_1^2+L_2^2+L_3^2}\right)}{\sqrt{L_1^2+L_2^2+L_3^2}} \begin{pmatrix} 0 & -L_3 & L_2 \\ L_3 & 0 & -L_1 \\ -L_2 & L_1 & 0 \end{pmatrix} \end{aligned} \tag{C.10}$$

C.2 Derivation of moment of momentum

This derivation starts with the fundamental equation

$$\mathbf{G} = \dot{\mathbf{H}} \tag{C.11}$$

$$H = \sum \mathbf{r} \wedge m\dot{\mathbf{r}} \tag{C.12}$$

$$= \sum \mathbf{r} \wedge m\omega \text{ for a rigid body} \tag{C.13}$$

$$= \left(\sum mr^2\right)\omega - \sum (m\mathbf{r}.\omega)\mathbf{r} \tag{C.14}$$

It is not immediately clear how to separate ω out from this since $\mathbf{r}$ is not constant for the summation. Studying the individual components of the expression provides a solution:

$$\begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} = \sum mr^2 \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} - \sum m(x\omega_x + y\omega_y + z\omega_z) \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$= \sum mr^2 \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} - \sum \begin{pmatrix} mx^2\omega_x + mxy\omega_y + mxzw_z \\ mxy\omega_x + my^2\omega_y + myzw_z \\ mxz\omega_x + myz\omega_y + mz^2w_z \end{pmatrix}$$

$$= \sum m(x^2 + y^2 + z^2) \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} - \sum \begin{pmatrix} mx^2 & mxy & mxz \\ mxy & my^2 & myz \\ mxz & myz & mz^2 \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix}$$

$$\Rightarrow \mathbf{H} = \begin{pmatrix} A & -F & -E \\ -F & B & -D \\ -E & -D & C \end{pmatrix} \omega$$

where

$$A = \sum m(y^2 + z^2) \qquad D = \sum myz$$

$$B = \sum m(z^2 + x^2) \qquad E = \sum mzx$$

$$C = \sum m(x^2 + y^2) \qquad F = \sum mxy$$

Moments of Inertia Products of Inertia

From this we have derived the well-known inertia equation, allowing inertia to be calculated for bodies composed of particles.

C.3 Singular value decomposition

Singular Value Decomposition (SVD) is a technique whereby a series of three matrices **U**, **W** and **V** have the property:

$$\mathbf{A} = \mathbf{U}.\begin{pmatrix} w_1 & 0 & \dots & 0 \\ 0 & w_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & w_n \end{pmatrix}.\mathbf{V}^T \tag{C.15}$$

A is considered to be square for the purposes of this discussion, though the theory can be extended for other cases. The decomposition is useful because the values of w_1, w_2, ... give an indication of how near-singular the matrix is and hence diagnose precisely what the problem is with matrices which cannot be inverted. While performing a SVD will diagnose the problem, it does not however provided a solution.

Inverting a (square) matrix once it has been decomposed is relatively straightforward, being simply

$$\mathbf{A}^{-1} = \mathbf{V}.[diag(1/w_j)].\mathbf{U}^T \tag{C.16}$$

Matrices which are near-singular will have one or more small values w, while one which is actually singular will have zeros. Clearly $1/w_j$ in these cases will yield a value approaching ∞ which is why problems arise when inverting. A *condition number* can be used to gauge how *ill conditioned* a matrix is — very large values will cause problems.

A simple solution to inverting ill-conditioned matrices is often possible: in cases where $1/w_j$ are considered "near-infinity" they can simply be replaced by zero. This removes components which "pull" solutions of towards ∞ along a direction caused by round-off errors.

C.4 The Newton-Raphson technique

This section provides a derivation and the code for solving simultaneous equations using Newton's Method. It describes the algorithm used to solve a series of N relations which are to be zeroed by finding appropriate values for $x_{1...N}$.

$$F_i(x_1, x_2, \ldots, x_N) = 0 \qquad\qquad i = 1, 2, \ldots, N \tag{C.17}$$

C.4.1 Solver algorithm

In the neighbourhood of $\mathbf{x}$, each of the functions F_i can be expanded in the Taylor series

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\delta F_i}{\delta x_j}\delta x_j + O(\delta\mathbf{x}^2) \tag{C.18}$$

A matrix of partial derivatives appearing in Equation C.18 is the Jacobian Matrix J

$$J_{ij} = \frac{\delta F_i}{\delta x_j} \tag{C.19}$$

The Jacobian for a set of four simultaneous equations would therefore be computed from the following partial derivatives:

$$J_4 = \begin{pmatrix} \frac{\delta F_1}{\delta x_1} & \frac{\delta F_1}{\delta x_2} & \frac{\delta F_1}{\delta x_3} & \frac{\delta F_1}{\delta x_4} \\ \frac{\delta F_2}{\delta x_1} & \frac{\delta F_2}{\delta x_2} & \frac{\delta F_2}{\delta x_3} & \frac{\delta F_2}{\delta x_4} \\ \frac{\delta F_3}{\delta x_1} & \frac{\delta F_3}{\delta x_2} & \frac{\delta F_3}{\delta x_3} & \frac{\delta F_3}{\delta x_4} \\ \frac{\delta F_4}{\delta x_1} & \frac{\delta F_4}{\delta x_2} & \frac{\delta F_4}{\delta x_3} & \frac{\delta F_4}{\delta x_4} \end{pmatrix}$$

Equation C.18 may be written in matrix notation as

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J}.\delta\mathbf{x} + O(\delta\mathbf{x}^2) \tag{C.20}$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, a set of linear equations may be obtained for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely

$$\mathbf{J}.\delta\mathbf{x} = -\mathbf{F} \tag{C.21}$$

This equation may be rearranged to be

$$\delta\mathbf{x} = -\mathbf{J}^{-1}.\mathbf{F} \tag{C.22}$$

Now this equation may be solved by an LU (Lower and Upper triangle) decomposition, and the computed $\delta\mathbf{x}$ added to the solution vector $\mathbf{x}$.

$$\mathbf{x}' = \mathbf{x} + \delta\mathbf{x} \tag{C.23}$$

where $\mathbf{x}'$ is the new $\mathbf{x}$.

$\delta\mathbf{x}$ is known as the "Newton Direction", and is the *initial* direction which will reduce the value of all the equations, but can fail to converge to a solution if

simply added to $\mathbf{x}$ as in Equation C.23. §C.4.2 provides a solution to this.

C.4.2 Linesearch algorithm

The linesearch algorithm attempts to find a reasonable distance to travel along the "Newton Direction". It works by investigating the impact of accepting the whole step, and then back tracking if need-be until it reaches an acceptable step. Without going into any detail, a value λ is computed to be

$$\lambda = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \tag{C.24}$$

which is used during the backtracking operation. The details of how this is derived can be found in [PTVF92].

Appendix D

Integration of C/C++ with Perl

Perl has extensive support for integration with C and C++, whether it be to allow C to call Perl routines, or vice versa. In this section the technique by which the simulator and MAVERIK have been interfaced within Iota is described. It is necessary to write an 'XS' file which describes function prototypes and is pre-processed into true C/C++ for compiling.

Besides the XS file itself, an additional file, known as `typemap` is required which instructs the pre-processor on how to convert to and from Perl data types. A standard version is supplied which allows all the standard C types (e.g. integers, floats, strings etc) to be mapped. The `typemap` file has three sections. The first is a table of C/C++ types, and an arbitrary name for the data type it represents.

```
TYPEMAP
int          T_IV
unsigned     T_IV
long         T_IV
char         T_CHAR
char *       T_PV
...
```

Similar C types can be mapped to the same tag, allowing their conversions to be handled in a similar fashion. The default tag names have been constructed as follows: `T_` for Type, `I`, `CHAR` and `P` for Integer, Character and Pointer, and `V` for

Value. The following definition block, preceded with an `INPUT` contains C code to coerce a Perl type into a C type.

```
INPUT
T_IV
            $var = ($type) SvIV ($arg)
T_CHAR
            $var = (char) *SvPV ($arg, na)
T_PV
            $var = ($type) SvPV ($arg, na)
...
```

The `Svs` stand for Scalar Value, to be contrasted with Arrays and Hash Tables. A similar section is then required for converting C types back to Perl.

```
OUTPUT
T_IV
            sv_setiv ($arg, (IV) $var) ;
T_CHAR
            sv_setpvn ($arg, (char *) &$var, 1) ;
T_PV
            sv_setpv ((SV *) $arg, $var) ;
...
```

Now that these relationships have been built, the task of writing XS code is made much simpler. An example C function such as

```
int Validate (System* this_system, int verbose)
{
        ...
        status = ...
        ...
        return status ;
}
```

could be restructured as an XS file with little modification, to become

```
int
Validate (system, verbose)
        System* system
        int verbose

CODE:
```

```
        ...
        RETVAL = ...
        ...

OUTPUT:
        RETVAL
```

The `xsubpp` programme manipulates this file, using the `typemap` file, to C which can be compiled. `xsubpp` can also create Perl methods for C++. Functions for converting objects must be explicitly added to the `typemap` file.

```
TYPEMAP
System*  O_OBJECT

INPUT
O_OBJECT
        if (sv_isobject($arg) && (SvTYPE(SvRV($arg)) == SVt_PVMG)
          & correct_type ($arg, \"$type\"))
            $var = ($type)SvIV ((SV*)SvRV($arg));
        else
        {
           warn (\"${Package}::$func_name(): $var is not a $type\") ;
           XSRETURN_UNDEF;
         }

OUTPUT
O_OBJECT
        sv_setref_pv ($arg, CLASS, (void *) $var) ;
```

The `INPUT` section does a number of consistency checks to prevent bad accesses to data structures. The XS file may now map Perl object methods to C++ ones. For example:

```
System*
System::new ()

void
System::DESTROY ()

void
System::display ()

int
```

```
System::my_id ()
CODE:
        RETVAL = THIS->my_id ;

OUTPUT:
    RETVAL
```

will create the constructors, destructors and other methods for an object called `System`. There are a number of points which can be made about this example. Not all the methods have an explicit function body: these cases will be assumed to bind to a C function or C++ method with the same name and arguments. A special case of this is the `System::DESTROY` method which, because it is the Perl destructor, will `delete` the object which in turn will call `System::~System ()` for class cleanup. The implementation still needs to be done with care: some underlying C structures may be static, some may be `malloc`ed and others may be local to a C function. Some caution and understanding of the underlying C is required to find the demarcation between C's memory management and Perl's.

In common with many scripting languages Perl is responsible for its own memory management, not the user. That is, it keeps reference counts to allocated memory items such as variables and when the last reference is finished with the memory is released. This makes manipulation of objects with pointers or references much less error-prone as neither dangling pointers nor unreferenced memory are possible within Perl's memory area. An object's `DESTROY` method will only be called when its last reference is removed. This means that users cannot fall into the traps which they are exposed to when manipulating the underlying C++ objects directly. Naturally, this is a huge benefit to a scene description language.

D.1 An example of passing a function reference to C

This section describes the basis on which Perl callbacks are built. A routine `perl_call_sv` allows a call to be made to a Perl function and removes all the complexity which would otherwise be required to cope with the alternate ways of making subroutine calls as well as providing a handle to a subroutine regardless of the methods used to create it.

An example shown in Figure D.1 taken from the `perlcall(3)` manual page, illustrates this nicely. In this example, the `CallSubSV` has been implemented as an XS module as shown in Figure D.2.

```
CallSubSV("fred") ;  # Text, assumed to be function name
CallSubSV(\&fred) ;  # Explicit address of function
$ref = \&fred ;
CallSubSV($ref) ;    # Function address held in variable
CallSubSV( sub { print "Message from perl\n" } ) ;
                     # Inlined anonymous code call
```

Figure D.1: An example illustrating some of Perl's function call mechanisms

```
void
CallSubSV(name)
    SV *    name
    CODE:
    PUSHMARK(sp) ;
    perl_call_sv(name, G_DISCARD|G_NOARGS) ;
```

Figure D.2: C XS module which accepts a function and calls it

Taking a copy of the `SV *` passed is all that is required to retain the subroutine handle in question to implement callbacks into Perl. Another function, the 'execute callback' function can then initiate the callback through the preserved `SV *` sometime later.

Appendix E

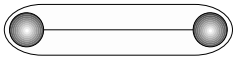
Simple user scripts

Systems, bodies, particles, connections and cosmetic connections are all instantiated with a method called `new`. To give an overall impression of the Perl API at this level, we present three simple scripts and describe how they work.

The first example, shown in Figure E.1, creates a system with one body falling under gravity. The default system contains one body composed of just one particle, and both are assigned to variable names. The position of `$particle1` defaults to the origin, and that of `$particle2` is set explicitly to `(30, 0, 0)`. The direction in which gravity acts is set and the system is simulated.

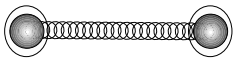
This simple example illustrates a number of key points, firstly when a new system is created the constructor of a body is implicitly called which in turn implicitly creates a particle. Secondly, further particles are created explicitly and added to the body.

Consider now a similar scene, but with a force function between the two particles instead of a cosmetic bond. The new script describing this scenario is shown in Figure E.2. Recall from Figure E.1 that both particles existed within the same body as it was rigid. In this example however, the particles move within the restrictions imposed by the force function. Moreover, the predefined force function is passed as a textual string to the `new` method of `Bond`. This string is



```
# Create system
$system = new System ;
# Set gravity vector
$system->gravity (Direction (0,-1)) ;

# Name default body
$body1 = $system->bodies(0) ;
# Name default particle
$particle1 = $body1->particles(0) ;

# Create, name and set the position of
# a new particle
($particle2 =
  new Particle)->position (Position (30,0,0)) ;
# Create a cosmetic bond between
# particle1 and particle2
new CBond($particle1, $particle2) ;

# Event Loop
# Interaction and rendering would go inside this
# loop too
while (1)
{
  $system->advance_frame () ;
}
```

Figure E.1: A Perl script and corresponding scene



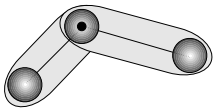
```
# Create system
$system = new System ;
# Set gravity vector (z defaults to 0)
$system->gravity (Direction (0,-1)) ;

# Name default body and its particle
$body1 = $system->bodies(0) ;
$particle1 = $body1->particles(0) ;

# Create a new body
$body2 = new Body ;
# Name body2's default particle and
# set its position vector
$particle2 = $body2->particles(0) ;
$particle2->position (Position (30,0,0)) ;

# attach a predefined force callback function called
# force_function1 to act between particle1 and
# particle2
new Bond($particle1, $particle2, "force_function1") ;

# Event Loop
while (1)
{
  $system->advance_frame () ;
}
```

Figure E.2: A Perl script describing two particles connected by a force function

matched with a force function of the same name and the appropriate computation is carried out.

The force function can be defined in a Perl subroutine anywhere in the simulation file or imported from another package and passed to `new` using any of the mechanisms described in Appendix D.1.

Now suppose a user wants to change the script further to describe a scene with two bodies connected by a hinge the modification to the script is trivial. In this case, the hinge is simply created by a `combine` call. The positions of the particles combined are not required to be identical as the solver will be called upon to impose and maintain the hinge constraint.



```
$system = new System ;
$system->gravity (Direction (0,-1)) ;

$body1 = $system->bodies(0) ;
$particle1 = $body1->particles(0) ;
($particle1a =
new Particle)->position (Position (-30,-35,0)) ;
new CBond($particle1, $particle1a) ;

$body2 = new Body ;
$particle2 = $body2->particles(0) ;
$particle2->position (Position (0,10,0)
($particle2a =
new Particle)->position (Position (35,-25,0)) ;
new CBond($particle2, $particle2a) ;

# Construct a hinge particle
combine ($particle1, $particle2) ;

# Event Loop
while (1)
{
  $system->advance_frame () ;
}
```

Figure E.3: A Perl script describing two bodies connected by a hinge

Appendix F

Results of user perception tests

This appendix contains the answers given by a sample of fifteen test subjects to the question; "What was your impression of the motion?" for each case study. The following set of tables summarise each test subjects responses to each case study. Where there were two examples of the same case study (e.g. Newton's cradle and interactive Newton's cradle) the more complicated one was used for the trial. Note that this is relevant in the following cases:

Newton's cradle - the interactive Newton's cradle was used for the user trials

Articulating chain - the interactive example was used for the user trials

Rigid molecule - the case study showing a portion detaching off the model was shown to all the test subjects

Each volunteer gave their permission for the following comments to be recorded and agreed that they were an accurate representation of their assessment of the case studies.

Name	Case study	Comment
Subject 1	Newton's cradle	Motion is plausible
	Jacob's ladder	Motion is believable
	Shoal of fish	Motion is plausible
	Undersea model	Motion is plausible
	Articulating chain	Motion is brilliant
	Rigid molecule	Motion is believable in the context of the inertia change
	Deformable molecule	looks like rubber
	Bucky ball	looks like a heart pulsating
	AIG laboratory	Cool!

Table F.1: Subject 1's comments

Name	Case study	Comment
Subject 2	Newton's cradle	Motion is plausible most of the time
	Jacob's ladder	I think that is plausible
	Shoal of fish	Yes that looks like a shoal of fish
	Undersea model	Yes I think that is plausible the seaweed is nice
	Articulating chain	Sufficient for me to accept as a chain
	Rigid molecule	Yes that looks realistic
	Deformable molecule	Looks stretch
	Bucky ball	Looks springy
	AIG laboratory	It is realistic

Table F.2: Subject 2's comments

Name	Case study	Comment
Subject 3	Newton's cradle	The model could be richer but the motion seems fine
	Jacob's ladder	I think that looks really good even the way it hits and bounces off - I can't fault it
	Shoal of fish	I can believe that to be the motion of a shoal of fish. The motion of individual fish could be a bit richer
	Undersea model	I believe that to be faithful
	Articulating chain	The motion is a bit too heavily damped but it seems realistic to me ... good, I like it
	Rigid molecule	I can't fault it
	Deformable molecule	Yes that looks good. Reminds me of an elastic material, sagging and a bit like jelly
	Bucky ball	Yes that is good
	AIG laboratory	Yes same as before, same models

Table F.3: Subject 3's comments

Name	Case study	Comment
Subject 4	Newton's cradle	That is quite nice. The elasticity between the mouse and ball is nice but a bit unintuitive. You seem to model the motion of the balls accurately
	Jacob's ladder	I can't fault the motion, it is realistic. I couldn't tell the difference from the real thing
	Shoal of fish	It looks fairly realistic when they flock towards the mouse pointer from a distance. It does look realistic
	Undersea model	Yes that could be a shallow water simulation. It gives a realistic impression of fish feeding in water
	Articulating chain	I like that behaviour, the way it copes robustly with large forces looks realistic. The joints also look realistic
	Rigid molecule	That looks very realistic
	Deformable molecule	Looks like a grocery bag, it is believable
	Bucky ball	They are all realistic, I'm finding it hard to think of comments ... can't really criticise them
	AIG laboratory	Very immersive, looks more tangible and more interesting than a static scene

Table F.4: Subject 4's comments

Name	Case study	Comment
Subject 5	Newton's cradle	Seems to be behaving like a Newton's cradle. A richer model of string would have been nice
	Jacob's ladder	This reminds me quite nicely of the real thing
	Shoal of fish	Looks fish-like, it is realistic enough
	Undersea model	The seaweed looks wriggly and seaweedish. It seems reasonable
	Articulating chain	Nice, I like that, that's good. I particularly like being able to attach and detach portions
	Rigid molecule	Yes that looks realistic, for a stick and ball model
	Deformable molecule	A bit abstract but it doesn't seem unreasonable for modelling soft objects
	Bucky ball	You can do hearts!
	AIG laboratory	Now that is pretty good, the models look even more believable in context and the limitations somehow become less of a problem
Comment: The interactive ones are more interesting because you can't do that with a cunning movie		

Table F.5: Subject 5's comments

Name	Case study	Comment
Subject 6	Newton's cradle	Not damped enough but apart from that it seems natural
	Jacob's ladder	Very realistic, especially the way in which it sways. Looks large because of the inertia
	Shoal of fish	That is just about believable enough. It is adequate, more than adequate
	Undersea model	That is OK, the seaweed looks like several straight bits with hinges but it does look OK
	Articulating chain	That is very good, probably the best one motion wise. Very realistic
	Rigid molecule	Very good, very realistic
	Deformable molecule	Looks very realistic but it went still a bit too quickly
	Bucky ball	I don't know about that one, it gives the impression of the camera moving in and out rather than a structure pulsating
	AIG laboratory	The cradle was very good, both models seem to behave just as well in context. It would have been nice to see the Jacob's ladder held by an avatar

Table F.6: Subject 6's comments

Name	Case study	Comment
Subject 7	Newton's cradle	Seems to be very accurate. Seems natural and the interface is quite intuitive
	Jacob's ladder	I like the way that the propagation of ripples doesn't seem to affect performance. It still behaves realistically even though the model is more complex
	Shoal of fish	Seems right
	Undersea model	It's fine, very nice
	Articulating chain	I like the oscillations, it is very good
	Rigid molecule	That is quite natural
	Deformable molecule	Yes, I think it is fine
	Bucky ball	It is like a heart
	AIG laboratory	More realistic, this really shows the potential of the kinds of things you can do. Could I play for a bit longer please?

Table F.7: Subject 7's comments

Name	Case study	Comment
Subject 8	Newton's cradle	Yes, I'm convinced
	Jacob's ladder	That looks very realistic
	Shoal of fish	That is amazing
	Undersea model	I like that, I could believe that especially with a blue background and some coral
	Articulating chain	It does look realistic, seems to be slightly damped a bit too much. I'm looking for things the real thing should do because the motion is so good
	Rigid molecule	That is pretty realistic once you work out what is going on. It doesn't look wrong
	Deformable molecule	Looks OK, it's a bouncy castle
	Bucky ball	Convincing but a bit too regular a structure to be interesting
	AIG laboratory	That's pretty good. I'm very impressed by the Newton's cradle but the Jacob's ladder falling off the screen is a little odd

Table F.8: Subject 8's comments

Name	Case study	Comment
Subject 9	Newton's cradle	Yes, I'm impressed. All the subtle movements help to make it convincing
	Jacob's ladder	I'm convinced by this one as well
	Shoal of fish	It's pretty plausible
	Undersea model	I like the way one of the fish comes in and takes a bit of seaweed. I'm convinced by that one as well
	Articulating chain	Apart from being a bit heavily damped it is convincing
	Rigid molecule	You can see the motion changes as the portion breaks off, it appears to be realistic. It isn't doing anything obviously wrong
	Deformable molecule	Yeah it's realistic, the lack of collision detection is a problem though
	Bucky ball	It looked like the image size was changing rather than the nodes moving apart
	AIG laboratory	As convincing as they were on their own. The context helps to put the models into perspective. It is easier to follow the motion with other reference points in the scene to compare against

Table F.9: Subject 9's comments

Name	Case study	Comment
Subject 10	Newton's cradle	Yes that's quite good that
	Jacob's ladder	Yes it looks fairly realistic. Once it has reached the end of its activity it gradually straightens out as one would expect ...except it doesn't appear to hang completely straight but that may or may not be correct
	Shoal of fish	I think it's quite good, quite realistic
	Undersea model	Yes, I think that is quite good
	Articulating chain	Yes, I thought that was quite good too. It illustrates disassembly and assembly in a dynamic way. It doesn't take account of assembly constraints but then that isn't the objective of the exercise
	Rigid molecule	Interesting illustration of varying mass and inertia. This sort of thing is important as a result of some activity
	Deformable molecule	Quite realistic but also shows difficulty in modelling complex organs which would require more processing power
	Bucky ball	Yes, that is quite interesting the system clearly has alot of potential. The approach is different to alot of others
	AIG laboratory	That is quite impressive, the sub models actively behaving in a complex environment simultaneously
Comment: It pulls alot of concepts together		

Table F.10: Subject 10's comments

Name	Case study	Comment
Subject 11	Newton's cradle	Yeah it seems fine, I the consolidation of a held set of balls with a colliding set a bit surprising
	Jacob's ladder	Yes, plausible
	Shoal of fish	Yes, that is plausible
	Undersea model	The motion of the fish seemed a bit too erratic but apart from that it was OK. I found it a bit difficult to see the fish dive down and pick up the seaweed
	Articulating chain	That is OK but I would expect the links to rotate in more than one axis. I believe it to be acceptable though.
	Rigid molecule	Yeah seems plausible
	Deformable molecule	Yeah all of the actual motion seems fine. Lack of collision detection is quite noticeable
	Bucky ball	Yes it seems fairly plausible it is a bit difficult to tell without some context
	AIG laboratory	Seems to be the same motion as before, seems fine. It makes the laboratory model more interactive

Table F.11: Subject 11's comments

Name	Case study	Comment
Subject 12	Newton's cradle	It's lovely, I love it. It's really nice. It's really real in a way it seems to respond to whatever I am doing. It is very real
	Jacob's ladder	Nice, I love it. Unfortunately I have never seen this toy before so it is a bit more difficult for me to interpret the movements. It seems realistic
	Shoal of fish	What I like here is that you can see the fish from different angles and this gives the feeling of reality. It seems like the fish play with each other as they swim. This is how fish do swim, I like this one
	Undersea model	I like this, I like seeing the individuality in the fish, showing that one fish has different behaviour
	Articulating chain	I think this is a very good example. I found the highlighting of selected links and attaching and detaching very good. I could see the potential of this for virtual magic tricks
	Rigid molecule	The motion is really good
	Deformable molecule	Yes, it is really real. I can see how gravity can change the shape of the object so it is really real
	Bucky ball	My mind went to the universe which expands and shrinks. The motion was excellent
	AIG laboratory	It looks lovely for me, I did not need to see the toys in this context because I could appreciate them on their own but I like them in the lab model too. The toys made the environment more interesting because you could interact with them

Table F.12: Subject 12's comments

Name	Case study	Comment
Subject 13	Newton's cradle	Yes, that is reasonable. It has a soft feel to it somehow that I wouldn't have expected ... from the interaction
	Jacob's ladder	Yes, that looks plausible
	Shoal of fish	Yes, that looks very good
	Undersea model	Yes, that looks pretty good again, maybe the seaweed is a little less realistic than the fish but there is a definite sense of depth there
	Articulating chain	Yes, that is pretty nice
	Rigid molecule	Yes, that's good, it might have been helpful to see what it was hanging from but yes it is fine
	Deformable molecule	Yes it is believable, might be clearer if connections were thinner but it looks realistic
	Bucky ball	At a faster frame rate it looks like it is pulsating but at a slower frame rate it looks like the camera position is moved in and out. It might have been nice to add some variation in the oscillations so that it doesn't appear so uniform
	AIG laboratory	Yes that's good

Table F.13: Subject 13's comments

Name	Case study	Comment
Subject 14	Newton's cradle	In terms of letting go of the balls the simulation behaves as one would expect. The interaction doesn't work in quite the way I expected
	Jacob's ladder	I think that is fine. It just looks like the real thing in slow motion
	Shoal of fish	That is fine
	Undersea model	The behaviour of the fish is very nice, the frame rate is a little too fast
	Articulating chain	It is very realistic apart from the fact that you feel you should be able to move hoops within each other
	Rigid molecule	Yes that is realistic
	Deformable molecule	I don't know what to relate this to, it gives the feeling of being a slightly rubbery thing dangling from a point
	Bucky ball	It's fine
	AIG laboratory	That's fine

Table F.14: Subject 14's comments

Name	Case study	Comment
Subject 15	Newton's cradle	That's realistic in terms of the motion. The balls don't appear to touch exactly but the motion is quite calming really. I could see this being used in a physics teaching system
	Jacob's ladder	The motion seems correct. It is difficult to see what is holding the toy though
	Shoal of fish	Yes, I could believe they are virtual fish
	Undersea model	That is impressive
	Articulating chain	The movement seems realistic
	Rigid molecule	The motion looks realistic. The position which it was rotating about was difficult to identify
	Deformable molecule	The motion is plausible
	Bucky ball	I found this to be more realistic at a slower frame rate
	AIG laboratory	The motion adds quite alot to the environment
Comment: An impressive set of demonstrations in terms of the range of stuff shown		

Table F.15: Subject 15's comments

Bibliography

[ABB+95] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Publications, Philadelphia, second edition, 1995.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *To appear in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, October 2000. ACM.

[AFH99] Bowen Alpern and Susan Flynn-Hummel. Issues in High-Performance Programming in Java. High Performance Computing Systems (HPCS99 Tutorial), June 1999. `http://www.research.ibm.com/jalapeno/papers/hpcs.99/present.html`.

[AFP+95] J. Auslander, A. Fukunaga, H. Partovi, J. Christensen, L. Hsu, P. Reiss, A. Shuman, J. Marks, and J.T. Ngo. Further experience with controller-based automatic motion synthesis. *ACM Transactions on Computer Graphics*, 14(4):311–336, October 1995.

[ALI] Alias/Wavefront. Headquarters: 210 King Street East. Toronto. Ontario. M5A 1J7. Canada. `http://www.aliaswavefront.com`.

[Ban86] A. Bandura. *Social Foundations of Thought and Action: A Social Cognitive Theory*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, 23(3):223–232, 1989.

[Bar92a] David Baraff. Dynamic simulation of non-penetrating rigid bodies. PhD thesis, Cornell University, 1992.

[Bar92b] Ronen Barzel. *Physically-Based Modelling for Computer Graphics*. Academic Press, San Diego, CA, 1992.

[BB88] Ronen Barzel and Alan Barr. A modelling system based on dynamic constraints. *ACM Computer Graphics*, 22(4):179–188, August 1988.

[BEA] Sun Microsystems Inc. 901 San Antonio Rd. Palo Alto. CA 94303 USA.
`http://java.sun.com/beans/`
`http://java.sun.com/beans/docs/spec.html`.

[Bea96] David M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *4th Annual Tcl/Tk Workshop*, Monterey, CA., July 1996.

[BH95] W. Barfield and C. Hexdrix. The effect of update rate on the sense of presence within virtual environments. *Virtual Reality: The Journal of the Virtual Reality Society*, 1(1):3–16, 1995.

[BHG91] D.E. Breen, D.H. House, and P.H. Getto. A particle-based computational model of cloth draping behavior. In *Scientific Visualization of Physical Phenomena*, pages 113–134. CG International, June 1991.

[BHG92] D.E. Breen, D.H. House, and P.H. Getto. A physically-based particle model of woven cloth. *The Visual Computer*, 8(5-6):264–277, June 1992.

[BHW94] D.E. Breen, D.H. House, and M.J. Wozny. Predicting the drape of woven cloth using interacting particles. *Computer Graphics*, 28(2):365–372, 1994.

[Box98] Don Box. *Essential COM*. Addison-Wesley Publishing Company, Menlo Park, CA, 1998.

[Bro] Peter Broadwell. `http://www.plasm.com/~peter/cyber24/cyber24.html`. Premiered at SIGGRAPH 1985.

[CDR96] U. Cugini, P. Denti, and C. Rizzi. Design and simulation of non-rigid materials handling systems. *Mathematics and Computers in Simulation*, 41(5-6):587–593, 1996.

[CHK98] J. Cook, R. Hubbold, and M. Keates. Virtual reality for large-scale industrial applications. *Future Generation Computer Systems*, 14:157–166, 1998.

[CLA] `http://members.aol.com/luthercode/clay/index.html`.

[COL] `http://www.coltvr.com`.

[Col98] Technofile Column. Real-time reaction. *New Scientist*, 2130:15, April 1998. Toshikazu Takada, Senior Researcher.

[Cow73] J.M.G. Cowie. *Polymers: Chemistry and Physics of Modern Materials*. International Textbook Company Limited, Aylesbury, 1973.

[Cox] Ray Cox. C++ v. Java A Comparison with C++. `http://members.aol.com/luthercode/clay/index.html`.

[CW97] Peter M. Chapman and Derek P.M. Wills. Towards a unified physical model for virtual environments. In *Proc. 4th UK VR-SIG Conference*, pages 130–139, Brunel University, UK., November 1997.

[CW98] Peter M. Chapman and Derek P.M. Wills. Modal analysis: A physical model for virtual environments. In *Eurographics UK*, pages 175–182, Leeds University, UK., March 1998.

[DBMS79] Jack J. Dongarra, James R. Bunch, Cleve B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Publications, Philadelphia, 1979.

[DG94] M. Desbrun and M.P. Gascuel. Highly deformable material for animation and collision processing. In *Proc. 5th Eurographics Workshop on Animation and Simulation*, Oslo, September 1994.

[DG95] Mathieu Desbrun and Marie-Paule Gascuel. Animating soft substances with implicit surfaces. *Proceedings of SIGGRAPH 95*, pages 287–290, August 1995.

[Dis37] Walt Disney's. Snow white and the seven dwarves. Animated film (83 minutes), 1937. `http://www.filmsite.org/snow.html`.

[DIV] `http://www.sics.se/dive/dive.html`.

[DMNS97] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, and Patrik Steyaert. Design guidelines for 'tailorable' frameworks. *Communications of the ACM*, 40(10):60–64, October 1997.

[DS83] J.E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, pages 147–152. Prentice-Hall Series in Computational Mathematics, 1983.

[Edw99] Phil Edwards. Language barrier. *UNIX & NT News*, pages 39–42, June 1999.

[EE98] Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, Redmond, WA, 1998.

[Ell96] S.R. Ellis. Presence of Mind: A Reaction to Thomas Sheridan's "Further Musings on the Psychophysics of Presence". *Presence: Teleoperators and Virtual Environments*, 5(2):247–259, 1996.

[Fau96] François Faure. An energy-based method for contact force computation. *Computer Graphics Forum (Proceedings of Eurographics 96)*, 15(3):357–366, August 1996.

[Fau99] François Faure. Fast iterative refinement of articulated solid dynamics. *IEEE TVCG*, 1999.

[FPRS98] S. French, K.N. Papamichail, D.C. Ranyard, and J.Q. Smith. Design of a decision support system for use in the event of a nuclear emergency. In F. Javier Giron and M. Lina Martinez, editors, *Applied Decision Analysis*, pages 2–18. Kluwer Academic Publishers, Boston, 1998.

[FPS+00] Y. Fabre, G. Pitel, L. Soubrevilla, E. Marchand, T. Géraud, and A. Demaille. An asynchronous architecture to manage communication, display, and user interaction in distributed virtual environments. In *Virtual Environments*, pages 105–113, 2000.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[FvFH90] J. Foley, A. vanDam, S. Feiner, and J. Hughes. *Computer Graphics Principles and Practice.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[FWT98] T. Fernando, Prasad Wimalaratne, and Kevin Tan. Interactive product simulation environment for assessing assembly and maintainability tasks. In *Proceedings of the 5th UK-VRSIG Conference*, Exeter University, September 1998. `http://www.dcs.ex.ac.uk/ukvrsig98/pap2_04.htm`.

[FWT99] T. Fernando, Prasad Wimalaratne, and Kevin Tan. Constraint-based virtual environment for supporting assembly and maintainability tasks. In *Proceedings of ASME Computers in Engineering Conference*, Las Vegas, Nevada, September 1999.

[Gas92] Jean-Dominique Gascuel. Displacement constraints: a new method for interactive dynamic animation of articulated solids. In *Third Eurographics Workshop on Animation and Simulation*, Cambridge, England, September 1992.

[GH97] Mashhuda Glencross and Toby Howard. Mirages. In *Eurographics UK 15th Annual Conference*, pages 223–235, Norwich, England, March 1997. Mashhuda Khote at time of writing.

[Gia84a] Douglas C. Giancoli. *General Physics*, volume 1. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Gia84b] Douglas C. Giancoli. *General Physics*, volume 2. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Gle88] James Gleick. *Chaos.* Sphere Books Ltd., London, 1988.

[GM98] Mashhuda Glencross and Alan Murta. Multi-body simulation in virtual environments. In Richard Zobel and Dietmar Moeller, editors, *Simulation — Past, Present and Future. 12th European Simulation Multiconference*, pages 590–594, Manchester, England, June 1998.

[GM99] Mashhuda Glencross and Alan Murta. A virtual Jacob's ladder. In *Graphicon 99*, pages 88–94, Moscow, August 1999.

[GPP86] Vadim Gerasimov, Alexey Pajitnov, and Dmitry Pavlovsky. Tetris. Computer Center of the Russian Academy of Sciences, 1986. `http://vadim.www.media.mit.edu/Tetris.htm`, `http://www.tetris.com`.

[Gri87] John Gribbin. *In Search of Shrödinger's Cat.* Corgi Books, London, 1987.

[Gro] Object Management Group. The common object request broker: Language mapping specifications. `http://www.omg.org/technology/documents/formal/corba_language_mapping_specifica.htm`.

[Gro99] Object Management Group. The Common Object Request Broker: Architecture and Specification Revision 2.3.1/IIOP. Formal/99-10-07, 1999. `http://www.omg.org/technology/documents/formal/corba_2.htm`.

[Guv99] Sinem Guven. Virtual lego (BSc 3rd year report). Technical report, Department of Computer Science, University of Manchester, 1999.

[HA95] Matthew Holton and Simon Alexander. Soft cellular modelling: A technique for the simulation of non-rigid materials. In *Computer*

Graphics: Developments in Virtual Environments, pages 449–460. Academic Press, 1995.

[Hay96] Alden M. Hayashi. The Developer of Java Speaks About its History, and Future Direction. *Application Development Trends*, pages 14–17, February 1996.

[HKG+98] Roger Hubbold, Martin Keates, Simon Gibson, Alan Murta, Steve Pettifer, and Adrian West. MAVERIK Programmers Guide. Edited by Jon Cook and Toby Howard. Technical Report MPG v4, University of Manchester, October 1998. Draft.

[IC87] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulations with kinematic constraints, behaviour functions and inverse dynamics. *Computer Graphics*, 21(4), Jul 1987.

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[Kol92] Craig E. Kolb. Rayshade user's guide and reference manual. `http://www-graphics.stanford.edu/~cek/rayshade/doc/guide/guide.html`, Jan 1992.

[KSB] Hartmut Keller, Horst Stolz, and Thomas Bräunl. `http://www.ee.uwa.edu.au/~braunl/aero`.

[KSZB95] Harmut Keller, Horst Stolz, Andreas Ziegler, and Thomas Bräunl. Virtual mechanics: Simulation and animation of rigid body systems with aero. *Simulation*, 65(1):74–79, 1995.

[Lan96] H.P. Langtangen. Introduction to solvers for nonlinear algebraic equations in diffpack. Technical Report Diffpack v1.4 Report Series, University of Oslo, December 1996.

[LD95] John Lasseter and Steve Daly. *TOY STORY: The Art and Making of the Animated Film.* Hyperion, New York, first edition, 1995.

[Lew71] P.G.T. Lewis. *SMP FURTHER MATHEMATICS SERIES II Vectors and Mechanics (Draft Edition).* Cambridge University Press, London, 1971.

[LIG] NewTek's LightWave 3D. Headquarters: 8200 IH-10 West. Suite 900. San Antonio. TX 78230. `http://www.newtek.com`.

[LJR+91] Annie Luciani, Stéphane Jimenez, Olivier Raoult, Claude Cadoz, and Jean-Loup Florens. A unified view of multitude behaviour, flexibility, plasticity and fractures: balls, bubbles and agglomerates. In *IFIP WG 5.10 Working Conference*, Tokyo, Japan, April 1991.

[LKC93] Wen-Bang Liu, Ming-Tat Ko, and Ruei-Chuan Chang. Sequential-goal constraints for computer animation. *The Journal of Visualization and Computer Animation*, 4(3):153–163, 1993.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly and Associates, second edition, 1992.

[LWA94] I.P. Logan, D.P.M. Wills, and N.J. Avis. Deformable objects in virtual environments. In *Proceedings of the UK VR-SIG*, Theale, Reading, 1994.

[Mae96] George Maestri. *Digital Character Animation.* New Riders, Indiana, 1996.

[MAX] Discreet's 3D Studio MAX. Headquarters: 10 Duke Street. Montreal. Quebec H3C 2L7. Canada. `http://www2.discreet.com`.

[MEM] `http://www.immersive.com`.

[Mey99] Bertrand Meyer. Rules for component builders. *Software Development*, 7(5):26–30, May 1999.

[MGH+98] A. Murta, S. Gibson, T.L.J. Howard, R.J. Hubbold, and A.J. West. Modelling and rendering for scene of crime reconstruction: A case study. In *Proceedings of Eurographics UK*, pages 169–173, Leeds, March 1998.

[Mil88] Gavin S. P. Miller. The motion dynamics of snakes and worms. *Computers and Graphics*, 22(4):169–178, 1988.

[MM99] Alan Murta and James Miller. Modelling and rendering liquids in motion. In *Proceedings of WSCG*, pages 194–201, Plzen-Bory, Czech Republic, February 1999.

[MP89] G. Miller and A. Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, 1989.

[MPPW94] F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling A Procedural Approach*. Academic Press Limited, Cambridge, Massachusetts, 1994.

[MRT] `http://www.cs.ualberta.ca/~graphics/mrtoolkit.html`.

[MS] Microsoft Corporation. One Microsoft Way. Redmond. WA 98052-6399. `http://www.microsoft.com/com/tech/com.asp`, `http://www.microsoft.com/com/tech/DCOM.asp`.

[Mus89] F.K. Musgrave. Prisms and rainbows: A dispersion model for computer graphics. In *Proceedings of the Graphics Interface '89 — Vision Interface '89*, Toronto, Ontario, June 1989.

[Nab] Thierry Nabeth. Centre for Advanced Learning Technologies, INSEAD, France. `http://www.insead.fr/CALT/Encyclopedia/ComputerSciences/AI/aLife.htm`.

[NAG] `http://www.nag.co.uk/numerical_libraries.asp`.

[New86] I.S. Newton. *Philosophiæ Naturalis Principia Mathematica.* Londini, Jussu Societatis Regiæ ac Typis Josephi Streater. Prostat apud plures Bibliopolas. Anno MDCLXXXVII., Trin. Coll. Cantab. Soc. Matheseos, Julii 1686. Translated by Andrew Motte 1729, `http://members.tripod.com/~gravitee/toc.htm`.

[OGL] `http://www.opengl.org`.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Ous96] J. Ousterhout. Additional information for scripting white paper, 1996. `http://www.sunlabs.com/people/john.ousterhout/scriptextra.html`.

[Ove91] C. Van Overveld. An Iterative Approach to Dynamic Simulation of 3-D Rigid-Body Motions for Real-Time Interactive Computer Animation. *The Visual Computer*, 7:29–38, 1991.

[Ove94] C.W.A.M. Van Overveld. A simple approximation to rigid body dynamics for computer animation. *The Journal of Visualization and Computer Animation*, 5(1):17–36, 1994.

[PB88] J.C. Platt and A.H. Barr. Constraint methods for flexible models. *Computer Graphics*, 22(4):279–288, August 1988.

[Pet99] Stephen Robert Pettifer. *An Operating Environment for Large Scale Virtual Reality.* PhD thesis, Department of Computer Science, 1999.

[Pla92] John Platt. A generalization of dynamic constraints. *CVGIP: Graphical Models and Image Processing*, 54(6):516–525, November 1992.

[PMT82] Star Trek II: The Wrath of Khan. Paramount Pictures, Motion picture (110 minutes), 1982.
`http://movie-reviews.colossus.net/movies/s/st2.html`.

[POV] `http://www.povray.org`.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C The Art of Scientific Computing*. Cambridge University Press, Cambridge, second edition, 1992.

[Ree83] W.T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *Computer Graphics*, 17(3):359–376, July 1983.

[Rey] Craig Reynolds. `http://hmt.com/cwr/boids.html`.

[Rey87] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, July 1987.

[RMI] RMI Specification. Sun Microsystems Inc. 901 San Antonio Rd. Palo Alto. CA 94303 USA.
`http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html`.

[ROD+98] Scot Thrane Refsland, Takeo Ojika, Tom Defanti, Andy Johnson, Jason Leigh, Carl Loeffler, and Xiaoyuan Tu. Virtual great barrier reef: A theoretical approach towards an evolving, interactive vr environment using a distributed dome and cave system. In *Proceedings of Virtual Worlds*, pages 323–336, Paris, France, July 1998.
`http://www.evl.uic.edu/aej/reef/reefpaper.html`.

[RSG] Mitchel Resnick, Oliver Strimpel, and Tinsley Galyean. The virtual fish tank. `http://www.tcm.org/html/fishtank`, `http://el.www.media.mit.edu/groups/el/project/fishtank`.

[SB90] F.P. Sayer and J.A. Bones. *Applied Mechanics A modern approach.* Chapman and Hall, London, 1990.

[SB91] Wolfgang Sohrt and Beat D. Brüderlin. Interaction with Constraints in 3D Modeling. *International Journal of Computational Geometry and Applications*, 1(4):405–425, 1991.

[SCE] `http://gd.tuwien.ac.at/graphics/sced/sced.html`.

[SF93] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. *Computer Graphics*, 9(5):369–376, August 1993.

[SG95] Dieter Schmalstieg and Michael Gervautz. Towards a virtual environment for interactive world building. Technical Report TR-186-2-95-08, Vienna University of Technology, 1995.

[Sim90] Karl Simms. Particle animation and rendering using data parallel computation. *Computer Graphics*, 24(4):405–413, August 1990.

[Sla99] Mel Slater. Measuring Presence: A Response to the Witmer and Singer Questionnaire. *Presence: Teleoperators and Virtual Environments*, 8(5):560–566, 1999.

[SOF] Softimage 3D. Headquarters: 3510 St-Laurent Blvd. Montreal. H2X 2V2. Canada. `http://www.softimage.com`.

[SP96] C. Szyperski and C. Pfister. Workshop on component-oriented programming, summary. In M. Muhlhauser, editor, *Special Issues in*

Object-Oriented Programming – ECCOP'96 Workshop. Dpunkt Verlag, Heidelberg, 1996.

[SS00] M. Slater and A. Steed. A virtual presence counter. *Presence: Teleoperators and Virtual Environments*, 9(5), October 2000.

[ST91] R. Szeliski and D. Tonnesen. Surface modeling with oriented particle systems. Technical Report CRL 91/14, Cambridge Research Lab, December 1991.

[Str87] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[Sut65] I.E. Sutherland. The ultimate display. In *IFIP Congress*, volume 2, pages 506–508, 1965.

[SZ90] Peter Schröder and David Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 23–31, Snowbird Utah, March 1990.

[Szy98] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. ACM Press, Addison Wesley, Harlow, London, 1998.

[Szy99] Clemens Szyperski. Components and objects together. *Software Development*, 7(5):33–41, May 1999.

[Tan] `http://www.evl.uic.edu/benjamin/`.

[TF88] D. Terzopoulos and K. Fleischer. Modeling inelastic deformations: Viscoelasticity, plasticity, fracture. *Computer Graphics*, 22(4):269–278, August 1988.

[TJ] `http://www.cis.upenn.edu/~hms/jack.html`.

[Ton91] D. Tonnesen. Modeling liquids and solids using thermal particles. In *Graphics Interface Proceedings*, pages 255–262, 1991.

[TPF89] D. Terzopoulos, J. Platt, and K. Fleischer. From goop to glop: Heating and melting deformable models. In *Graphics Interface*, pages 219–226, June 1989.

[U.S98] U.S. Department of Defense. *High Level Architecture Interface Specification*, version 1.3, draft 1 edition, April 1998. `http://www.dmso.mil/projects/hla`.

[Vin98] John Vince. *Essential Virtual Reality fast.* Springer-Verlag, 1998.

[Wat99] J. A. Waterworth. *Virtual Reality in Medicine: A Survey of the State of the Art.* Department of Informatics Umeå University, July 1999. `http://www.informatik.umu.se/~jwworth/medpage.html`.

[WCS96] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl.* O'Reilly and Associates, second edition, 1996.

[WFB87] Andrew Witkin, Kurt Fleischer, and Alan Barr. Energy constraints on parameterized models. *Computer Graphics*, 21(4):225–232, July 1987.

[WGW90] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–22, March 1990.

[WH91] Jakub Wejchert and David Haumann. Animation aerodynamics. *Computer Graphics*, 25(4):19–22, 1991.

[WHN+98] Anders Wallberg, Pär Hansson, Bino Nord, Jonas Söderberg, and Lennart E. Fahlén. *"The Mimoid and Blob" Projects Presentation/Poster.* ACM MultiMedia '98, Bristol UK., 1998.

[Wil90] Jane Wilhelms. Dynamics for computer graphics: A tutorial. In *SIGGRAPH Course Notes 8 Human Figure Animation: Approaches and Applications*, pages 85–115. ACM, Dallas, 1990.

[Woo99] Benjamin Wooley. Rules of the game. *Personal Computer World*, pages 272–273, May 1999.

[WTT95] Yi Wu, Daniel Thalmann, and Nadia Magnenat Thalmann. Deformable surfaces using physically based particle systems. In *Computer Graphics: Developments in Virtual Environments*, pages 205–215. Academic Press Ltd, 1995.

[WW90] Andrew Witkin and William Welch. Fast animation and control of non-rigid structures. *Computer Graphics*, 24(4):243–252, August 1990.

[WW92] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques Theory and Practice.* Addison-Wesley and ACM Press, Wokingham and New York, 1992.

[ZPF+93] Michael J. Zyda, David R. Pratt, John S. Falby, Chuck Lombardo, and Kristen M. Kelleher. The software required for the computer generation of virtual environments. *Presence*, 2(2):130–140, 1993.