Simulating frictional forces in a virtual reality environment, using message routes between objects

Dominic DeCesare
BSc (Hons) Mathematics and Computer Science
University of Bath
May 2004

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed: . . . . . .

Abstract

Currently virtual reality (VR) environments, such as those found in computer games, predefine the way the user of the VR environment can interact with it. The users choices are limited to what the designer chose to include and as there is often no obvious difference between an object that can be interacted with and one that cannot, so the user must experiment with his surroundings to find out which are which.

The focus of ongoing research at the University of Bath is to develop a method that allows the user to interact with everything in a VR environment, by defining the objects within the environments with respect to the physical rules that govern the behavior of objects in the real world. The most important factor is to have objects behave in a way that the user expects, so that they can interact with the virtual environment in a confident manner. It is noted that believable behavior does not necessarily mean mathematically accurate behavior so simplifications of the complex physical rules can be used instead. By creating a VR in this way, the behaviors permitted by the system are no longer decided during the design phase and so emergent behavior is possible.

The key method behind this approach is using non-hierarchical dynamic interaction graphs (an extension of bond graphs) linked by a messaging system. So objects interactions are not defined by their position in the scene but by how the objects relate and influence each other in the VR.

This project focuses specifically on the modeling of friction using dynamic interaction graph and a messaging system. A solution to the problem is attempted, but does not succeed in producing test simulations due to difficulties with the managing of the object data.

Acknowledgements

Thanks are due to the following people: My supervisor Prof Phil Willis for his support and advice throughout this project. Mr Florian Schanda, currently working on PhD in this area of research, for his suggestions and advice on the focus of this project.

I would also like to thank the family and friends who offered their support during the project.

Signed Declaration

Simulating frictional forces in a virtual reality environment, using message routes between objects

submitted by Dominic DeCesare

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with its author. The Intellectual property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:……..

Contents

Figures

1. Introduction

1.1 Motivation

Current virtual reality (VR) environments, whether they are simulations or games, are generated by first creating the 'world' i.e. the rooms, objects within the rooms, etc., and then adding the physical rules to the 'world'. Consequently all interactions in the world, whether they are initiated by user interaction or triggered events, are hardcoded in and the options available to the user are limited to those that have been included by the designer. However, in the real world you can interact with everything (limited only by the physical constraints of the world), and there are multiple approaches that can be undertaken to solve a problem, and multiple interactions that arise from just performing everyday tasks. Within currently produced VR many of the routine interactions commonplace in the real world are not present, and while they could be reproduced in a VR world the current design approach would require they were each individually coded in, and the designer would have to be aware of, and hardcode in, all the effects (or noticeable effects) relating to or caused by that interaction.

There is a current research project [1] investigating the issues in developing a system architecture that would allow all or almost all objects in a VR to affect and interact with each other, while conforming to a range of physical properties, which would allow the emergence of new behaviour not specified by the designer. This would produce a VR world that while not (and not needing to be) scientifically accurate, would behave in a manner sufficiently like the real world that users would be able to confidently interact with it, knowing both the available interactions and the consequences of those interactions.

An example of this would be trying to retrieve a glass from a high shelf in a kitchen to have a drink of water. In an old style VR world you would have one object, for example a stepladder, which would allow you to gain the height to reach the glass. In the desired VR you could use any object to get the height, for example by standing on a chair, but you could also use a different object to have a drink, for example getting a mug from elsewhere in the kitchen, or even just drink straight from the tap, bypassing the problem altogether.

1.2 Aim

This project will aim to examine a smaller subsection of the problem by producing a simulation demonstrating a believable representation of frictional forces between objects within a VR world, using a message based system.

1.3 Objectives

The objective is to produce a series of simulations that realistically demonstrate the effect of frictional forces at work, using the messaging system between the interacting objects to transmit the relevant information, while keeping the introduced load on the messaging

system as small as possible. Each additional simulation should expand upon the previous one, demonstrating the designed system works effectively across a range of complexity. There is the option of further development of simulations showing the effects of multiple sources of friction, variable friction, as well as demonstrating the various effects caused by friction (e.g. heating of the surfaces in contact, sound generated as the surfaces move past each other). To model friction realistically it will be necessary to examine a variety of sources of friction, but not to model the additional and often unidentifiable factors that increase the force required to get the object to move [2] or the effect of the deformation of surfaces in contact.

This will require an understanding of the GNU MAVERIK tool for producing the virtual reality environment, of C or C++ to implement the solution, of the messaging system interface and of the way frictional forces work and their effects. Further investigation into mathematical modelling of frictional and kinematic effects will also be required. While the existing research on the project has not investigated the modelling of friction, previous projects have studied the modelling of collisions using the aforementioned tools and messaging system. Along with the current research, this should provide an insight into the potential difficulties associated with this approach.

2. Literature Survey

2.1 Motivation and background

Virtual reality (VR) environments have been existence for many years in variety of forms, though the greatest motivation for its continued development has come from the computer games industry, it has many applications including simulations for both training purposes and to study behaviours in an easily controlled environment. The aim of this work regardless of the area in which it is being used, has been to simulate the world as accurately as possible either to create a greater sense of immersion and intuitive behaviour in games, develop more accurate and therefore more reliable training simulations or to increase the accuracy and validity of research and calculations based upon simulations of real world events. For the past few years the primary focus has been on the development or more realistic graphics, but recently the focus has shifted to the accurate modelling of physics.

The purpose of the research that this project is a part of is the development of a VE that "*behaves sufficiently like the real world*" [1], so that a user can manipulate it and be confident of the results of their actions. While current VR environments provide clear causes and events of specified interactions, this work aims to allow the "*emergence of new behaviour*" [1] as each object behaves as is expected and interactions are not limited to those predefined by the designer of the simulation. The proposed approach has the objects within in the VR aware of their locations and through the use of Dynamic Interaction Graphs (DIGs) links the objects to those they are in contact with. The DIGs also provide routes, which allow event messages to be passed, when an object is interacted with within the VR.

The focus of this project is to investigate if the proposed approach can be used to accurately represent the effects of friction with a VR simulation. The aim of this literature survey is to investigate the current research on physics in virtual environments, specifically with regards to the modelling of friction to provide a greater understanding of the nature of the problem. This will ensure the conclusions of the project have a greater validity and the meaning by placing them within the context of existing work. By investigating the approaches currently proposed within research papers it should be possible to identify the best methods to use to achieve the desired results and the project will benefit from the experience gained from the problems encountered in similar work.

2.2 Problems with calculating the physics

VR simulation can be described as a process of numerically solving a differential equation, which describes the evolution of a system over time [3]. The inclusion of physics within a simulation places an additional computational burden on the system. This additional burden is due to the increasing complexity of the differential equation that must be solved as it includes more and more effects with the simulation. In games this means that the "*physics has to compete with graphics, audio, multiplayer group communication, and non player AI for a heavily constrained time budget*" [4], the time

budget mentioned referring to the fact that each new state must be calculated within the frame rate of the game. When the computation time of the differential equation exceeds the threshold specified for the game, visible slow down occurs with the game, and it jumps from one state to a visibly discrete new state. Slow down inhibits interactivity with a simulation, and normally realism is sacrificed (either by reducing the graphic detail or the complexity of the physics) to ensure that the simulation runs at an acceptable speed. A commonly used solution is to have the physics solver using a model with a smaller polygon count than the graphical model [4], this however can be problematic with very complex models especially in collisions where the simulation appears to show the two objects intersecting, or not actually contacting when they collide.

The differential equations for physics (specifically rigid body dynamics) are most commonly solved using the Euler method, Midpoint method or a variety of Runge-Kutta and implicit methods [3]. With all these methods there is a trade off between computation speed and accuracy, where the less accurate methods may become unstable due to inaccuracies in the modelling, resulting in behaviours inconsistent with the users expectations.

In this research the focus is in producing believable physics not necessarily accurately calculating the physics, so that the world behaves and responds as expected. As such the methods that favour computation time over accuracy will be preferred, so that interactions with any object can be calculated quickly and result in a behaviour that agrees with the users experience in the real world.

2.3 Dynamic Interaction Graphs & Messaging

The key to achieving the aim of this project is to create the DIGs so that they can be efficiently created and searched, and to produce a message system which is fast and memory efficient using the DIG system. Each DIG represents a pair of objects in the scene which are capable of influencing each other as they are in contact or through other means, for example an object being magnetic can affect other objects without being in contact with them. As this is an ongoing research project previous works on the research have developed various ways of creating and storing the DIGs. The most successful of which ([5]) stored the DIGs as a 1D array, and proved that in doing so reduced the required memory that would be need if they were stored in a matrix configuration. As the system used in [5] for the DIGs has been to shown to work in the simulations produced as part of that work, it is worth noting though that the work did encounter limitations with the DIG design, and while the work focussed on collision detection, the difficulties encountered with multiple collisions may mean that similar problems could be encountered with multiple sources of friction.

The messaging system will send messages along the connections specified by the DIG graphs. An interaction with an object not only affects all objects it is directly connected to, but all objects that are physically related to it. A simple example is a stack of books. Removing the book from the bottom shouldn’t cause just the book above it to fall, but all the books in the stack to fall. This means that any message sent along a DIG’s connection

to an object should be propagated from that object to all the objects it is connected to via any number of DIGs. This means that an efficient algorithm for flooding the message throughout the simulation is required. An algorithm for sending the messages needs to ensure it avoids unnecessary message traffic by reducing the number of times an object is sent the message when it has already received it from a different route, while at the same time ensuring that each object in the system that should receive the message does, and the time delay between the message being sent and the last object receiving it ([6]) is as small as possible. Generally there is a trade off between ensuring complete coverage of all the objects required in the smallest time, and the number of unnecessary messages sent by the system. Ideally each object would receive the message once, and send it on only to those objects that have yet to receive this, but as the message will be propagating along several routes at the same time, it is entirely possible for an object to receive the same message from several directions. While it is possible to ensure that an object ignores any more messages it receives with the same point of origin and original sending time after the first, it is also important to ensure that any algorithm used is capable of dealing with the situation when an object receives multiple messages from all directions at the same time.

It is important that in creating the DIGs for a simulation that the adopted approach is as efficient as possible to reduce the overall load on the system, and the same is true for the passing of messages between the DIGs as well. While the systems developed in [5] are proven to work, they do have their limitations. Developing a message system based upon the one designed in [5] should provide a good starting point, but it will be necessary to optimise it for the specific problem, and to ascertain if the friction simulation could be integrated with the existing work.

2.4 Friction

The key focus of this project is the production of realistic representation of friction as both a retarding force on a moving object, and a constraining force on a static object. Coulomb's law describes the effects of friction in both of these instances. As explained in [7], the cause of friction is the same in both of these cases, and is a result of the combination of the roughness of the two surfaces in contact, as even 'smooth' surfaces are actually rough when viewed at very high magnification, and a result of the forming and breaking of atomic or molecular bonds between the two surfaces and on the edge of each of the surfaces. The resulting force due to the friction is different depending on whether the two surfaces are static or are in motion relative to each other. This is described in Coulomb's laws by two different coefficients of friction; one for static friction (μ_s) and one for motive, or kinetic friction, (μ_k). These coefficients of friction are entirely dependent on the materials of the two surfaces in contact, and the difference between the coefficients varies for each material pair. The table in **fig 1** is taken from [7] and lists the two coefficients for a variety of materials. It clearly shows that the difference between the two coefficients is as dependent on the materials as the coefficients are.

MATERIAL	μ_k	μ_s
Steel on Steel	0.7	0.6
Brass on Steel	0.5	0.4
Copper on Cast Iron	1.1	0.3
Glass on Glass	0.9	0.4
Teflon on Teflon	0.04	0.04
Teflon on Steel	0.04	0.04
Rubber on Concrete	1.0	0.8
Rubber on Wet (Concrete)	0.3	0.25

Fig 2.1: Frictional coefficients for common materials

Coulomb's law also states that the frictional force is dependant on the normal force between the two objects (F_n), but is independent of the area of contact between the two objects, and this fits experience as heavier objects are harder to drag than lighter ones. This results in the following two formulas for the frictional forces:

Static Friction: $\mathbf{f_{s,\,max} = \mu_s F_n}$

Kinetic Friction: $\mathbf{f_k = \mu_k F_n}$

The formula for static friction gives the maximum value of static friction on an object, the magnitude of force above which must be applied to the object to get it to move. Up to this point the static force is equal and opposite to the motive force being applied to the object, so the object does not move. The graph below (**Fig 2**), also taken from [6], describes the frictional force on an object relative to the applied force. It clearly shows that force required to get an object to move is less than the force require to keep an object moving and that once moving the frictional force acting on an object is constant.

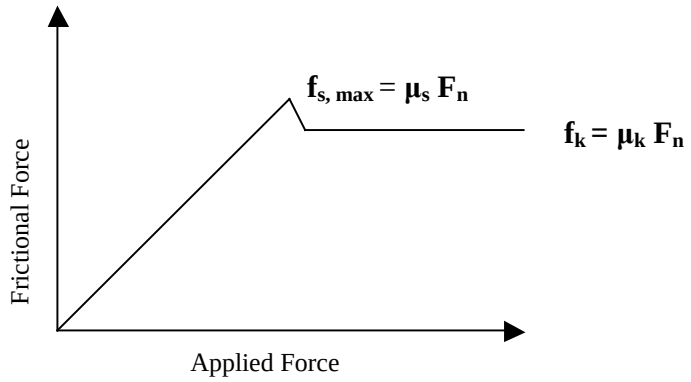


Fig 2.2 Graph of the Frictional Force vs. Applied Force for two objects in contact

The normal force in the simple case of an object resting upon another object in the real world is equal to the force of gravity acting upon the object, as otherwise the object on top would sink into the object below, and can be calculated from it as is shown in **fig 3**. **Fig 4** details a more complicated case where an object is sandwiched between two surfaces. In this example the value of the normal force due to gravity is equal to zero, but there is a normal force resulting from the object and the walls pushing on each other. However the frictional force can be easily calculated to have a magnitude equal to the force of gravity on the object, but in the opposite direction, otherwise the object would slide down.

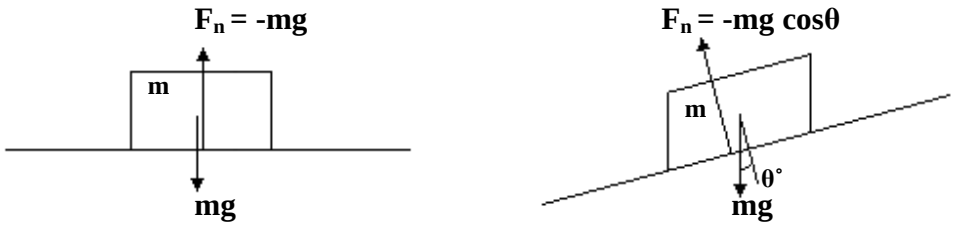


Fig 2.3 Calculating the normal force for an object on the flat and on a slope

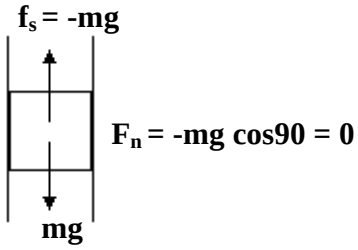


Fig 2.4 An object sandwiched between two walls

In the dynamic case where the object is rolling instead of sliding the friction between the two surfaces prevents it from sliding, and in this case it is the coefficient of static friction between those two surfaces that governs when the rolling object will start to slide. However, as described by [2], the slowing down effect, which is sometimes referred to as rolling friction, is actually a result of the deformation of the rolling object and the surface, and in the case of a wheel the friction of the wheel at the axel as well. The coefficient of the rolling friction is typically significantly smaller than that of the dynamic or static friction, for example the coefficient of rolling friction for a tyre on a road (rubber on concrete) is between 0.02 and 0.06 ([2]).

Modelling friction in VR environments has been investigated before. The problem can be divided up into three cases; the static case where the object is at rest, the dynamic case where the object is in motion, and the transition from the static case to the dynamic case.

Of these cases the transition between the static and dynamic states causes the most difficulty in modelling, as it requires that the methods for solving the static and dynamic case must be combined into a single algorithm. One of the key issues in modelling the effect of friction is in calculating the normal force (F_n), the standard way of modelling contact forces using the principle of constraint is an NP-hard problem, taking exponential time to solve ([8]) making it unusable for real time applications. With principle of constraint you first attempt to find a valid configuration with non-impulsive forces, before using impulsive forces, the calculation of determining whether a consistent solution exists using impulsive forces and it is this calculation that causes the bulk of the difficulty. The problems is made worse when you have to deal with states, which are inconsistent or indeterminate ([8]), i.e. where there is no valid set of contact forces or where the solutions to the problem result in there being multiple configurations for the object in question.

A variety of approaches have been developed to attempt to solve the problem. For dynamic friction in ([8]) Barraf suggests a method, where you use a combination of impulse and non-impulse forces to make up the contact force, and calculate the forces using Lemke's algorithm. Lemke's algorithm provides a way to solve the linear complementary problem (LCP), provided it is positive semi-definite, and the resulting vector give rise to either valid set of contact forces if they exist or a valid set of contact impulses if the configuration is inconsistent. This approach is then extended to static friction, by treating it as a dynamic friction, and requires the object has a very small tangential velocity or "crawl" [8] to maintain the static friction force.

Saeur in [9] develops this approach with the aim of sacrificing accuracy to allow it to be used in real time environment, 15 – 25 visualizations a second ([9]), linearizing the equations of motion and friction so that they can be solved. In order to linearize the cone of frictional force it is converted into a friction pyramid (**fig 5**), the linearized friction force is then added to the equation of motion by calculating its value for all of the vectors d_{kn}, Lemke's algorithm is then used to solve and the new linear and angular velocities are used in the calculation of the new state of the system.

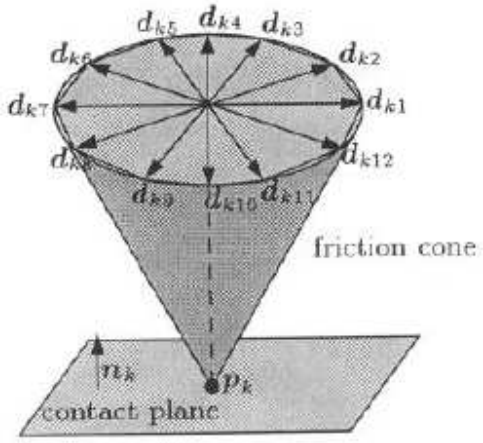


Fig 2.5 The friction cone, and the corresponding friction pyramid with 12 facets, in the case of a contact between two objects at P_k. Taken from [9].

In [10] Hart discusses the use of a similar approach but does not go into the same detail as Sauer in [9]. He notes that the use of hard constraints and the use of the velocity force LCP, can avoid the problems caused by the lack of solutions of differential algebraic equations, and acceleration force LCP, as well as the artificial stiffness of the penalty method.

Hilde [11] suggests a fast method of solving the dynamic equations of movement through the use of Euler's equations, and the Broyden's non-linear system solving method. In doing it is shown that an implicit rather than explicit integration method can be used for real time simulation. This allows the benefits of implicit integration such as its high stability and it's ability to deal with high stiffness ordinary differential equations. However, it is worth noting that it become necessary to reduce the time steps, and therefore to increase the simulation time, to solve very stiff ordinary differential equations.

With the exception of Barraf [8] the issue of static friction is not discussed in any particular detail, and Barraf leaves it as an area to be further investigated, and as such the issue of the change from the static to the dynamic case is also avoided. However all of these show that the accurate simulation of friction as described Coulomb's law is difficult problem that can have detrimental effects on the computation time. It is also important to note that in order to achieve stability in the integrations, a degree of accuracy has to be sacrificed. However the focus of this project is the development of "believable" physics, using a message system, for an interactive world, and as such computation time is the key. Therefore the effect of friction is more important in this project that modelling the causes, and either a simple approximation for the contact forces must be devised or an existing physics library must be used, to allow the production of friction force that behaves as expected.

2.5 Existing approaches

2.5.1 Open Dynamics Engine (ODE):

This is an open source C/C++ library for the simulation of rigid body dynamics [12], and has been used in commercial games as well as other VR applications. It can be used as a real time application, and has a very stable integration algorithm for solving the ordinary differential equations, though this algorithm only works with relatively large time steps and therefore has a limited accuracy. The developer admits that it needs work before it can be used for applications requiring a higher degree of accuracy and recommends it is not used in a situation when accuracy is important. This library does include two different approaches for modelling friction, though rolling friction and aerodynamic drag have to be simulated by the introduction of a damping force [13]. As it calculates the movement of an object, given an initial state and properties of the object, it could be used in the development of emergent systems. Both of these features make it a possible choice for the physics engine for the project, but as it is limited only to dynamics it would only allow a limited degree of behaviours. It does prove that the conventional approach to physics simulation does work. Though this project, and the overall research, is not focussed on physical accuracy, the effectiveness of this approach is worth considering. The main difficulty with studying ODE is the lack of documentation of the design of the system, meaning its use in the development of this project is potentially limited.

2.5.2 HAVOK 2:

HAVOK is a middleware software package designed to provide games developers with a physics library optimised for gaming both on PC's and on consoles. It provides developers with a variety of levels of accuracy allowing for both highly detailed physics when required or a series of less accurate systems to improve performance. It has been used in Half-Life 2 to generate a game where you are supposed to be able to pick up and move almost any object in the game (either by hand or using the gravity gun for large objects). As it is commercial software I have not, as of yet, been able to gain any more information on the specific approach they used to allow this interaction with any object, how they have approached the problem of modelling friction, or if it, in Half-Life 2, actually allows for emergent behaviour in how you approach and interact with the game.

2. 6 Evaluation of software and languages

2.6.1 VRML 2.0:

Is the standard language for creating virtual reality worlds, though as it only provides connections to other languages to define object behaviours it is not a complete solution. As VRML does use a system of nodes with routes linking them up it appears to provide a good starting point for the development, however as it requires that all interactions are pre-specified at the design stage it does not allow for the development of emergent behaviours which is the overall goal of the research this project forms a part of.

2.6.2 GNU Maverick:

Is a toolkit for developing 3D virtual environments, that has been used both in previous projects in this area, as well as by Prof. Willis in the creation of the simulations he has produced with regards to this research. As it has been developed as a tool kit, it makes a very small number of assumptions about the nature of the application ([14]), and provides several levels of tools allowing it to be used for different complexities of tasks so that new objects can be created, and MAVERIK can be extended as necessary. In using MAVERIK this project will be able to take advantage of the existing experience in using the tool in past projects, and the time saved in not having to produce a set of graphics tools from scratch for the project.

2.6.3 C/C++:

These have been the programming languages of choice for previous projects in this area. They are both easily available for free and fast due to their low level nature and so are ideal for the development of this work. Both languages are compatible with MAVERIK meaning either is a valid choice, however, as the bulk of existing work that this project will be expanding on, including the message code developed in [4], was written in C++ it is the sensible choice. In using C++ I gain the advantage of the experience gained through other work in the area, both successful approaches and information on the problems previously encountered.

2.7 Conclusion

The previous work on this project area has shown that the proposed system using dynamic interaction graphs and messages can be used to produce a virtual reality world with believable physics, specifically with regards to collision. The research into the current attempts at accurately modelling friction, and the Open Dynamics Engine, have shown that actual physics libraries can be created that are fast enough to produce realistic physics for interactive simulations, though in all cases the necessary performance is achieved through the sacrifice of accuracy. The key to creating believable friction with the approach suggested by Prof. Willis [1] must come through keeping the computation time, and resource cost of modelling friction as small as possible. Then it can be integrated into a much larger series of events being produced by interacting with the simulation, all of which must be propagated around the VR environment by the messaging system without causing the VR to operate and respond in less than real time. While it has been shown that accurate friction effects can be created and combined with collisions and other physical effects in real time simulations, the focus of this project must be to investigate if these effects can be simulated in a system that uses dynamic interaction graphs and a messaging system rather than the traditional scene graph approach. To create an interactive real time VR environment that can support fully emergent behaviours, this project must focus on the development of a simple approximation to the effects of friction, that appears to accurate to the viewer and behaves how they would expect.

3. Requirements

In order to correctly develop a system it is important to first analyse and then define the requirements of the system. As this is a research project rather than a software development project these requirements must be based upon the projects objectives. Therefore the feasibility of the approach being investigated by the project will be determined by how successfully the project meets its requirements.

3.1 Requirements analysis

The key objective of this project is to develop simulations demonstrating believable physics, which are based upon knowledge of how the real world behaves. Therefore a scenario-based approach [15] was selected to capture the requirements. As this project is concerned with determining the feasibility of the approach, discussed in [1], the requirements analysis focused on the behavior of an object as understood by someone viewing the simulation, rather than on the computational accuracy of the results. Additionally requirements were drawn from the project synopsis [1] as it details the approach that this project is investigating. The structure of this chapter is based upon standard models of systems requirements specifications as detailed in [15].

3.2 Requirements Specification

3.2.1 Functionality

A sufficient number of simulations must be developed to show that the message based approach can be used to model friction, both in the static and dynamic cases, and in the transition between the two. They must also demonstrate that the architecture can produce believable results of the behavior of all objects in the simulation as their influence of each other develops as the simulation progresses.

3.2.2 User Interface

The user interface with the simulations is not a priority as the potential users are those already versed with the research area, and the simulations are being developed to demonstrate friction forces, not interactivity. However, the user will need to have control of the viewpoint, and the simulations may allow the user to interact with the bodies, in these cases the interface can be provided through the use of the standard MAVERIK functions, as demonstrated by the examples provided with MAVERIK.

3.2.3 Environment

The primary requirement of the environment on which the developed software will be used is that the simulations will run at a suitable frame rate on an average specification PC, for example the computers available in the university library. It would be desirable for the core functions developed for the situations to be reusable for simulations of physical behavior outside the scope of this project, but it is not a vital design requirement.

The use of MAVERIK to create and manage the graphical part of the simulations, allows the software generated to be used on any machine with a functional MAVERIK distribution. It also helps to ensure that the software responsible for the physical simulation can be used in other projects on this area of research.

3.2.4 Friction

The simulations generated must demonstrate all of the expected frictional forces. This includes both the effects of friction on a static object, as a holding force, and on an object in motion, as a retarding force. Each simulation should also be able to cope with the transition between these states, which as was noted in [8] is a key problem in the modelling of friction. The code used to calculate the friction effects in a simulation should be simulation independent, otherwise the events in the simulation are being effectively hard coded and emergent behaviour will not occur. Therefore either code to calculate the friction effects must be created or an external physics library, for example ODE, must be integrated into the system design. As believability is more important than physical accuracy to this project a simpler constraint based friction modeller may be a better alternative, than a stand alone physics library.

To properly demonstrate that the message based approach can be used to create emergent systems, the simulations must also demonstrate believable frictional effects as the system progresses. This includes the effects of the application of impulses to objects already in motion, and of continuous, but not necessarily constant, forces to both static and moving objects. Additionally, the simulations should demonstrate the effects when the source of friction on an object changes.

If time allows additional friction effects, for example the transference of kinetic energy to heat and sound, can also be included to demonstrate the message based approach is feasible for the creation of various different but related physical behaviours.

3.2.5 Collision Detection

While not specifically the focus of this project the simulations may require additional functionality for the detection and resolution of collisions. This functionality may be provided with existing systems or the careful design of the simulations may allow simple approximations to be used, while still believably demonstrating the friction effects. It is worth noting, as mentioned in [16], that objects in impulse based systems have a tendency to bounce on a surface with decreasing motion, rather than being at rest. As such any collision detection system must have constraints that force an objects velocity perpendicular to a surface to be zero when the objects velocity is negligible.

3.2.6 Message System

A message passing architecture must be used, as this is the key principle of the approach being researched. The message passing system will need to be sufficiently fast that it does not have a detrimental effect on the frame rate of the system below an absolute

minimum of 15 frames per second [9]. The architecture must not be specifically created for the simulation of friction forces and must be used to transmit any object interaction that occurs. In conjunction with the DIGs the message system needs to ensure that when a message is sent detailing a change in the state of an object, every object that is affected by that change is informed of it. However it must also ensuring that objects don't react to, or transmits on, multiple instances of the same message propagated to them via different objects.

It is essential for the simulations to demonstrate that the modelling of friction can be achieved with as small a number of messages as possible. This is a requirement, as in a simulation where the objects simulate all the physical behaviour needed to accurately model their interactions, the message system must be responsible for passing all the information required for these interactions. At the same time these messages must be first propagated and then the information sent must be used to generate the correct responses before the next frame of animation can be displayed. As such if the friction modelling introduces a significant load on a message system then the approach may not be deemed as a success, as the frame rate of much more complex systems may be negatively affected as a result.

3.2.7 DIG Manipulation

The system must include functionality to manage the DIGs used to determine the interaction between objects in the system. This requires that each object is located in the same dynamic interaction graph as every object it influences or is influenced by. In the gear wheels example [17], the objects were placed in a dig together, if the objects were located next to each other in the grid. Objects in the 3D simulations this project will produce will have the full six degrees of freedom, which is significantly more complex than the gear wheels demonstration. However as the focus of the project is modelling friction, which only occurs between objects in contact, physical proximity is the only factor needing to be considered.

It is important that DIGs do not impose an ordering in a system, or result in objects being defined with respect to the other objects in the simulation. The only information a dig should store is which objects in the simulation can influence each other.

4. Design

As this is a research project that is investigating the effectiveness of the proposed method it is inevitable that some of the fundamental design decisions will result from problems that are encountered in the implementation. Therefore it is not completely possible to fully design the system before implementation. This section will cover the core functions the simulations will require, as well as the how the system is divided into components, and the way the components interface.

As the aim of this project is to create a series of simulations that build upon each other to demonstrate the various friction effects, but that require the same core functions it is sensible to decompose the system into a series of modules [15], with well defined interfaces, that can be reused for each simulation. As such C++ becomes the preferred language as it is object orientated, and the use of classes makes it simple to link the necessary data structures to the required functionality.

4.1 Modules

The following components are essential for the simulations, and were obtained by analysing the requirements:

- Object manager – Stores data required for the simulations and provides functions for the access and manipulation of this data. Links the MAVERIK data to that defined by the simulation for the object
- DIG – provides functionality to create, update and delete DIGs. Also needs to provide functions for iterating out the members of a DIG. Each value stored in a DIG refers to an object, and therefore the functions and data defined on that object.
- Message system – provides functionality to create, destroy messages, and pass messages. Needs to be able to provide a framework for a range of information to be passed.
- Friction – provides functions for calculating the effects of friction. This may be provided by ODE. Friction coefficients will be modified so that it is the sum of the friction values of the surfaces in contact so that a look up table of friction values isn't required to store the coefficients of friction between a variety of materials.
- Collision Detection and response – provides functions for calculating the results of collisions between objects. This may also be provided by an ODE.
- Graphical Output – provided by MAVERIK, which is an interface to the GL implementation. This allows the drawing of the simulation to be controlled by a series of simple function calls, therefore freeing the time the project would otherwise have to spend developing the graphics engine for the simulations

This structure of the system, both the modules and their connections is shown in the diagram (**Fig 4.1**) below.

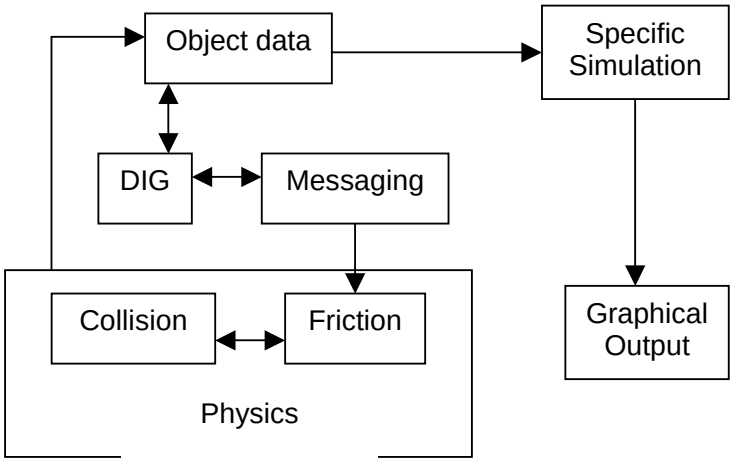


Fig 4.1 The structure of the whole system

Using these modules the program flow can be described diagrammatically, **Fig 4.2** below.

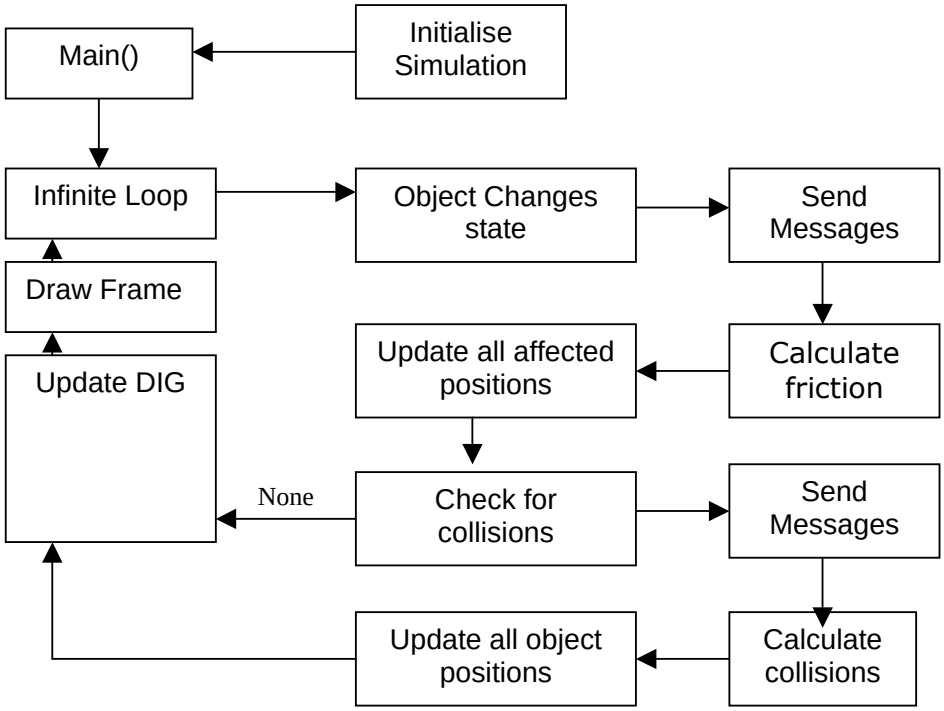


Fig 4.2 Flow of program

4.2 Object Data

The object data is stored in a class, which defines the variables needed for friction simulation and the management of messages. This class also defines a link to the MAVERIK data for that object. This means an object data instance can easily update the MAVERIK data for the rendering loop.

The object data will need to store the mass of the object, as well as the three friction coefficients for the objects possible motion. For the messaging system it needs to store the ID of the last message it received and for the DIG system the ID of the DIG graph to which it belongs.

4.3 DIGs

For memory efficiency the DIGs are stored in one dimensional list like structures, that have computationally efficient methods for accessing elements at the ends of the list, and by index. In that way the overhead of adding new objects to a DIG, and iterating through each member of the DIG can be kept to a minimum. To ensure that each object can be quickly send messages to those it influences when it changes state, or receives a message that must be propagated on, each object will have its own DIG graph. While this does add additional memory costs, as an objects is stored in several digs (each object pair takes double the memory, as they are each stored in the others DIG), this can be offset by storing only the ID of an object in a DIG, and accessing its data via the system manager. This also has the advantage that the DIG for an object can be passed to a function, which can then iterate over the members of the DIG rather than having to check the object for membership against every DIG.

The DIG class needs to include methods for deciding whether an object is under the influence of the other. This will be based on the bounding box approach, already implemented in MAVERIK, to evaluate when two objects are in contact and therefore need to be placed into a DIG.

4.4 Messaging system

The messaging structure should provide a series of message types that can be sent, each with specific data relevant to the action they are transmitting. In the case of friction, when an object receives a force that may result in a change of the objects velocity it must send a message detailing this to the objects connected to it, they in turn need to return their friction coefficients to that object.

Other messages relaying their mass may also be required, for example if there is a stack of objects the normal force between the base of the bottom object and the floor is based on the total mass pressing down into the floor of all objects in the stack, and hence, according to Coulomb’s laws, the friction force between the object on the bottom of the stack will be greater. This produces difficulties, as messages need to propagate away from and then back towards the original transmitting object for it to get the information

of all other objects that could affect it. It doesn't make sense to capture all of these in a DIG, because it would need to differentiate between those in contact, and those that are connected via objects in contact, which comes back to introducing a scene graph architecture where objects are defined in relation to their positions to each other, which is the very thing this research is investigating an alternative too.

A more efficient alternative would be too pass via messages the friction coefficients of all other objects that are in contact with the target object, whenever the digs are updated. However this would still require for the calculation of mass that the messages are propagated from all objects that are connected, either directly or via objects in contact with the target object, to the target object.

4.5 Friction library

This can be provided by either an external library, such as ODE if it proves suitable or, by a basic constraint based implementation. The static case would require a function that calculates the friction force on the object, and compares it to the motive force applied to the object, and if the static force is larger it prevents the object from moving and if smaller allows the object to move with a force equal to the difference between the friction force, and the impulse force. The motive case could be treated in a similar fashion, expect the motive force can be calculated from the objects velocity from the previous time step. Assuming a unit time step, the motive force is then equal to the mass times the velocity, and if the friction force is larger the object stops, and if it is smaller the object continues but with a force equal to the difference.

5. Implementation

One initial problem was encountered with the MAVERIK graphics package, which was its incompatibility with a variety of C++ compilers, including those it was recommended to be used with. In the end Visual Studio 2003 was obtained for use with it, and after time spent altering the MAVERIK libraries so they would compile, development was able to begin.

However, the key problem encountered in the implementation was accessing the data of the MAVERIK objects. The initial approach as detailed in the plan was abandoned as all attempts to access the MAVERIK data structures, via a pointer stored in the object data failed. Instead the development changes to include a class that stored both the object data and MAVERIK data as locally defined lists, with functions designed to access the data efficiently to take advantage of the list based types provided in C++. However this did not resolve the problem, and several attempts to store pointers to MAVERIK data were tried without success, these included variations of void pointers to the MAVERIK data, cast both to MAV_object and to the specific types of MAV_object, e.g. Mav_box. It was noted in this process that removing the MAV_SMS (list of objects) to the simulation function, stopped objects from being drawn. It was at this point at which the development of the simulations was stopped to produce the report, the system had evolved into the following state.

5.1 Modules

Following the difficulties encountered with the initial design, the set of modules required for the simulation was altered, with the addition of a simulation manager designed to tie the object data and MAVERIK data together, it became more efficient to transfer the functionality of passing messages to objects, and deciding when an object needed to be added to a dig to the simulation manager, as it stored all the digs, objects and messages defined with it's own domains.

- Simulation manager – maintains lists of all objects in the simulation, and should link the data of an object, required for the physics of the simulation, with the objects data for rendering (MAVERIK data of an object) and with the objects DIG graph. Also maintains the queue of messages waiting to be resolved in time step of this simulation. Needs to link to the messenger and DIG modules in order to pass messages between objects as required. Is responsible for updating the data of all objects in the simulation, calling the friction library to calculate friction effects on objects, calculating membership of DIG graphs and propagating messages between objects.
- Object manager – Stores data required for the simulations and provides functions for the access and manipulation of this data.
- DIG – provides functionality to create, update and delete DIGs. Also needs to provide functions for iterating through the members of a DIG. Each value stored in a DIG refers to an object ID, which is used by the simulation manager.

- Message system – provides functionality to create, destroy messages. Needs to be able to provide information for many different types of messages. However is no longer responsible for propagating messages.
- Friction – provides functions for calculating the effects of friction. This was provided by a simple constraint based version of friction as presented in chapter 4, based on the coulomb model. However was never tested due to the project not working.
- Collision Detection and response – this was not implemented, as it was not the focus of the research of the project, as collisions have already been covered before in both [5] and [16] and it would not assist in the study of the modeling of friction.
- Graphical Output – still provided by MAVERIK, which is an interface to the GL implementation. Boxes can be produced but cannot be connected to the libraries for DIGs, messages passing and friction calculation.

This structure of the system has now become, as shown in the diagram (**Fig 5.1**) below.

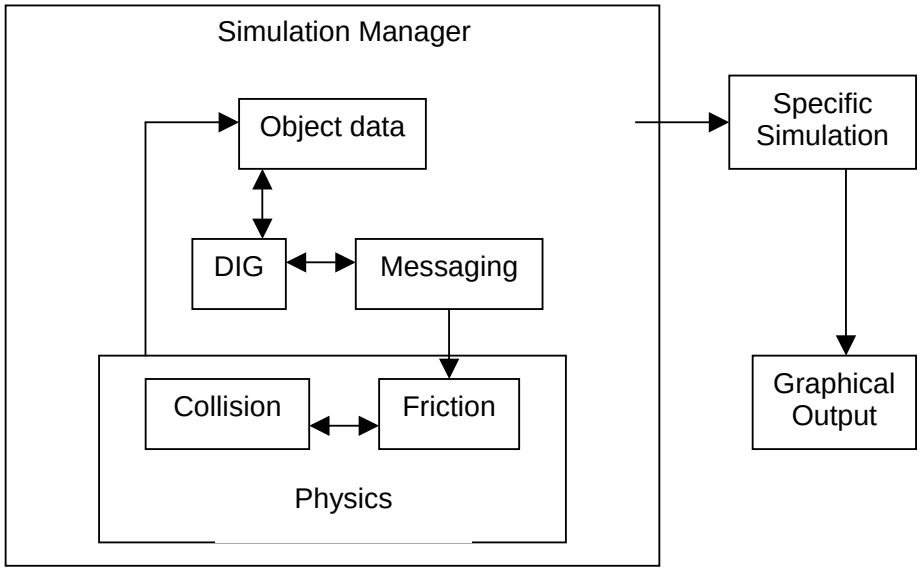


Fig 5.1 The structure of the whole system

Using these modules the program flow can be described diagrammatically, **Fig 5.2** below.

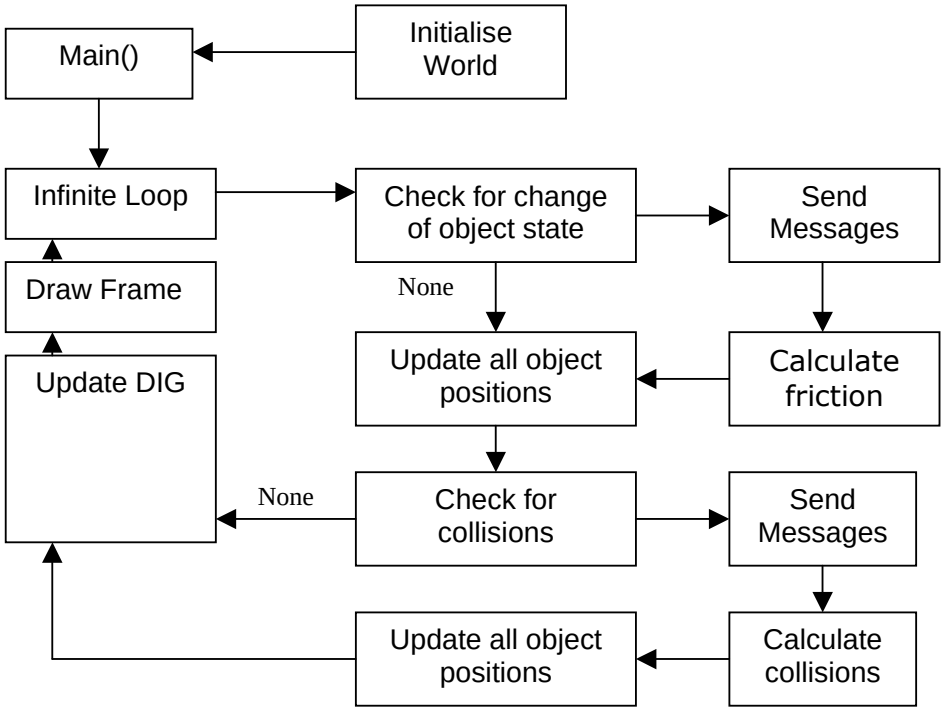


Fig 5.2 Flow of program

5.2 Simulation Manager

The simulation manager has three main roles, firstly to define the world with respect to physical quantities, secondly to link together the data needed for modeling friction with the data for rendering the objects and lastly to update all the objects in every frame of the simulation. By designing the system this way you can effectively link all the necessary parts of the messaging systems and DIG graphs together while controlling the access the simulation has to them. This also allows the various modules of the system to be reused as only the top level need be altered to include extra functionality that other simulations may require.

The simulation manager stores the values that apply to the world in general, and are independent of the objects in it. For these purposes it will be used to store the gravity in the VR simulations, but this could be easily extended to other include other information such as ambient temperature, or atmospheric viscosity.

It maintains the links between each of the objects' two data structures by maintaining parallel lists of the ID's of the objects. Each objects then ID represents its position in the list of objects data, and in the lists of MAVERIK data. Using efficient list type data structures allow fast access to the instances of the data objects. The individual data of these objects can then be assessed using either the relevant class functions in the case of the object data, or the callback functions.

Each rendering loop can be simplified in the main simulation program to a few calls to the simulation manager. This can then quickly iterate through its lists of the objects and pass their data to the other modules, which then calculate the new positions of objects, update the dig graphs and evaluate both friction effects and collisions.

This allows more complex simulations to be grown out of basic prototype ones, while avoiding redevelopment of significant parts of the libraries.

5.3 Object Data

The object data is stored in a class, which defines the variables needed for friction simulation and the management of messages. Data is stored separately from MAVERIK structures to allow easy access by the simulation manager.

5.4 DIGs

Maintained by the simulation manager, the module provides the functions for the creation and updating of DIGs. The DIGs are implemented as C++ double ended queues as they are the most efficient data structure for manipulation of values at both the ends of the list, and at indices of the list.

6. Testing Scenarios

The project focused on the developing of a series of simulations designed to demonstrate if the approach being researched could be used to create believable friction effects. The scenarios were designed so that each scenario was more complex than its predecessor, to test both how effective the approach could be, and if and how its performance changes as the complexity of the system being simulated grew. As the key focus of the research is to see if the suggested approach using a message system and DIGs allows the development of simulations with emergent behavior, it is important that the simulations are not scripted in any way. Each of the simulations was designed to use a human observer to judge the correctness of the object behaviors. As was noted in [16], while a human observer will not notice a numeric error in the simulation if the objects in it behave as expected, they will be able to spot errors that numerical analysis of the results would not detect, for example if spherical objects were sliding instead of rolling. The testing scenarios are detailed in Appendix B.

As the project did not achieve a working simulation there are no results of these testing to comment on.

7.Conclusions

As the project failed to meet its objectives few conclusions can be drawn. The investigation into related work has shown that friction can be implemented in virtual reality, and that similar works into this area of research, have show that VR using Newtonian dynamics can be created using DIGs and a message passing architecture. The key issue that needs to be addressed is what messages to pass and how to define the relations on objects for the calculation of the normal force, in a non-hierarchical architecture, where the system is designed to not express objects locations by how they are positioned in relation to each other.

The initial research detailed well the scope of understanding of the problem, however the failure of the implementation of the simulations to test the suitability of the approach meant the initial research can not be used to support any conclusions beyond that which can be inferred directly from it.

The key failure of the project was in working out the most effective way to link the message passing, DIG based architecture to the graphical display. If other ways of creating the graphical outputs of the simulation had been considered then perhaps the project would have achieved its objectives. The problem was unfortunately worsened by the focus on resolving the difficulty rather than investigating the other key areas of the intending research. More development could have been done into the messaging system, and DIG architecture that could have provided more information for conclusions even if the graphical issues had not been resolved.

As this project failed to meet its objective the investigation of friction with a message based architecture can still be done, and the original remit of this project could be expanded to include other friction effects such as the heating of objects, and the generation of sound. More complicated issues such as air resistance could also be investigated in relation to this. Additionally the issue of friction could be applied to directly to the models, as described in [16], to see how effective a fully message based approach is with a combination of Newtonian dynamics implemented.

Bibliography

[1]Willis, P. J., Proposed Research, Bath University, 2004. Included on CD.

[2] Nave, C. R., Static Friction, Kinetic Friction, Friction Plot, Rolling Friction, Hyper Physics, 2003, as available on
http://hyperphysics.phy-astr.gsu.edu/hbase/frict2.html

[3] Barraf, D., Interactive Simulation of Solid Rigid Bodies, Computer Graphics and Applications, pp 63 –75, vol. 15, issue 3, IEEE, May 1995.

[4] Low, C., Games and Physics: Design Issues, Internet and Mobile Systems Laboratories, HP Laboratory Bristol, HPL-2000-124, Hewlett-Packard Company, 2000.

[5] Zhang, L., A Message based Physical Behaviour Simulation System for Virtual Environment, MSc. Dissertation, University of Bath, 2004.

[6] Jonsson, J, and Vasell, J, A Comparative Study of Methods for Time-Deterministic Message Delivery in a Multiprocessor Architecture, Proceedings of 10th International Parallel Processing Symposium (IPPS'96), pp 392-398, IEEE, 1996.

[7] Tipler, P. A., Physics for Scientists and Engineers, 4th edition, WH Freeman and Company, 1999.

[8] Barraf, D., Coping with Friction for Non-Penetrating Rigid Body Simulation, Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques, pp 31 – 41, ACM Press, New York, NY, USA, 1991.

[9] Sauer, J, and Schömer, E, A Constraint-based Approach to Rigid Body Dynamics for Virtual Reality Applications, Virtual Reality Software and Technology, Proceedings of the ACM Symposium on Virtual Reality Software and Technology, pp 153 – 162, ACM Press, New York, NY, USA, 1998.

[10] Hart, G. D., and Anitescu, M., A Hard-constraint Time-stepping Approach for Rigid Multibody Dynamics with Joints, Contact, and Friction, Proceedings of the 2003 Conference on Diversity in Computing, pp 34 – 41, ACM Press, New York, NY, USA, 2003.

[11] Hilde, L., Meseure, P., and Chaillou, C., A Fast Implicit Integration Method for Solving Dynamic Equations of Movement, Proceedings of the ACM Symposium on Virtual Reality Software and Technology, Session: Algorithms, pp 71 – 76, ACM Press, New York, NY, USA, 2001.

[12] Smith, R, Open Dynamics Engine v0.5 User Guide, 2004, as available on http://ode.org/ode-0.5-userguide.html.

[13] RollingFrictionOrAerodynamicDrag, ODE Wiki Area, 2004, as available on http://ode.org/cgi-bin/wiki.pl?RollingFrictionOrAerodynamicDrag

[14] Cook, J., Howard, T., MAVERIK Programmer's Guide (version 6.2), Advanced Interfaces Group, University of Manchester, 2002, as available on http://aig.cs.man.ac.uk/maverik.

[15] Sommerville, I., Software Engineering, 6th Edition, China Machine Press, HZ Books, 2000.

[16] Richard Jones, Using Rigid-Body Mechanics And Dynamic Interaction Graphs To Create A Consistent And Believable Virtual Environment, BSc (Hons) Dissertation, University of Bath, 2004.

[17] Prof. Willis. P, MAVERIK example program (Gearwheels). Included on CD.

The following were referenced throughout the implementation and documentation process, respectively:

Stroustrup, B., The C++ Programming Language, 3rd Edition, Addison Wesley, 1998.

Dr Barry, A. M., The Project Dissertation, University of Bath 2004.

Appendix A

The code that appears on the following pages is all of the code that was written by the end of the project. It does not include the various unsuccessful functions and code designs that were rejected during the implementation. The codes include the core classes that were being developed for the planned scenarios, as well as what would have formed the main body of the simulation.

FrictionMessaging.cpp

```
#include "maverik.h"
#include <ode/ode.h>

#include <drawstuff/drawstuff.h>

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "Dig.h"
#include "Message.h"
#include "CoulombFriction.h"
#include "MavikObject.h"
#include "Mavik.h"
using namespace std;


#ifdef _MSC_VER

#pragma warning(disable:4244 4305)  // for VC++, no precision loss complaints

#endif

#define GRAVITY 1.0    // the global gravity to use

void defRect(MAV_rectangle *r) {

        r->width = 500.0; /* Size */

        r->height = 500.0;

        r->xtile = 3; /* Texture repeat tiling */

        r->ytile = 3;

        /* Orientation (RPY 0,-90,0) and position (XYZ 0,-2,0) */

        r->matrix = mav_rmtrixSet(0,-90,0, 0,-1,0);

        /* Use decal texture with index 5 */

        r->sp = mav_surfaceParamsNew(MAV_TEXTURE, 0, 0, 1);

        }

void defBox(MAV_box *b) {

        b->size.x = 2.0; /* Specify its size */

        b->size.y = 2.0;

        b->size.z = 2.0;

        b->matrix = MAV_ID_MATRIX; /* Position and orientation */

        b->sp = mav_sp_default; /* Surface parameters, i.e. colour */

}

void drawFrame(MAV_SMS *sms)
{
        mav_eventsCheck();
    mav_frameBegin();
        mav_SMSDisplay(mav_win_all, sms);
    mav_frameEnd();
        //printf ("frame rate: %4.2f\n", mav_fps);
}

int main(int argc, char *argv []) {
        MAV_rectangle gp;

        MAV_box box;
        MAV_SMS* sms;
        float gpprop = {0, 0.2, 0.27, 0.03};
        float boxprop = {10, 0.5, 0.45, 0.05};

        //      Dig worldname(0,  gp_dat.GetObjectID());


        mav_initialise(&argc, argv);
        sms = mav_SMSObjListNew();

        World sim(10);

        mav_paletteTextureSet(mav_palette_default, 1, "Marble_floor.ppm");

        defRect(&gp);
        defBox(&box);

        mav_SMSObjectAdd(sms, mav_objectNew(mav_class_rectangle, &gp));
        mav_SMSObjectAdd(sms, mav_objectNew(mav_class_box, &box));
        sim.AddObject(&gpprop, sms);
        sim.AddObject(&boxprop, sms);

        /* Use default mouse navigation */

        mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);

        while (1) {
        drawFrame(sms);
        }

}
```

```
World.h

#ifndef _WORLD_H_
#define _WORLD_H_

#include "maverik.h"
#include <string>
#include <deque>
#include <list>
using namespace std;

#include "WorldObject.h"
#include "Dig.h"
#include "Message.h"

class World {
private:

        deque<WorldObject*> objects;  //container of all objects
        MAV_SMS* smsptr;              //pointer to the simulation SMS
        deque<Dig*> digs;             //container of all digs
        list<Message*> messages;      //stack of all messages
        float gravity;                //world gravity

public:
        //constructor
        World(float grav);
        //add object to world
        int AddObject(float prop[], MAV_SMS* obj);
        //Gets object data
        WorldObject* UseObject(int pos);
        //Gets objects position
        int World::UseMAVObjectData(int pos);
        //Adds objects that influence each other to their digs.
        int World::UpdateDigs(void);
};

//constructor
World::World(float grav) {
        gravity = grav;
}

//add object to world
int World::AddObject(float prop[], MAV_SMS* obj) {
        int oldObj_no = objects.size();
        objects.push_back(&WorldObject(oldObj_no, prop[0], prop[1], prop[2], prop[3]));
        smsptr = obj;
        digs.push_back(&Dig(oldObj_no, oldObj_no));
        return 1;
}
//Gets object data
WorldObject* World::UseObject(int pos) {
        return objects[pos];
}
//Gets objects position
 int World::UseMAVObjectData(int pos) {
        /*Code to return data about Maverik object*/
        return 1;
}
//Adds objects that influence each other to their digs
 int World::UpdateDigs(void) {
          for(int i = 0; i < objects.size(); i++) {
                  for (int j = 0; j < objects.size(); j++) {
                          if (i != j) {
                                  /*Check for each of X, Y and Z if any object j is
                                  touching i. Do this by comparing the position of object
                                  i + half its size in each dimension to the position of
                                  object j +1/2 its size, in the same dimension. If the
                                  values are the same add j to i's dig*/
                          }

                  }
          }
          return 1;
 }
#endif
```

```
WorldObject.h
/*Class to set and get object data required for friction, and messaging*/
#ifndef _WORLDOBJECT_H_
#define _WORLDOBJECT_H_
#include "Maverik.h"
#include "CoulombFriction.h"

class WorldObject {
private:
        int objectID;           //Unique identifier of object
        int messageID;          //Identifier of last message it recieved
        int movetype;           //Type of movement 0 = static, 1 = sliding, 2 = rolling
        float objmass;                          //mass of object
        float staticfrictioncoef;               //object static friction coeffecient
        float slidingfrictioncoef;              //object sliding friction coeffecient
        float rollingfrictioncoef;              //object rolling friction coeffecient

public:
        //Constructor for type definiton
        WorldObject(void);
        //Constructor
        WorldObject(int objID, float mass, float stcoef, float slcoef, float rocoef);
        //returns friction coeffecient based on movement of object
        float GetFrictionCoef(void);
        //returns object mass
        float GetObjectMass(void);
        //returns ID of last message recieved
        int GetMessageID(void);
        //Returns ID of object
        int GetObjectID(void);
        //returns integer describing type of object movement
        int GetMoveType(void);
        //sets ID of last message recieved
        int SetMessageID(int messID);
        //Destructor
    ~WorldObject(void);


};

//Constructor for type definiton
WorldObject::WorldObject(void) {
}

//Constructor
WorldObject::WorldObject(int objID, float mass, float stcoef, float slcoef, float
rocoef) {
        objectID = objID;
        objmass = mass;
     staticfrictioncoef = stcoef;
        slidingfrictioncoef = slcoef;
        rollingfrictioncoef = rocoef;
}

//returns friction coeffecient based on movement of object
float WorldObject::GetFrictionCoef(void) {
        if (movetype == 0) {
                return staticfrictioncoef;
        }
        else if (movetype == 1) {
                return slidingfrictioncoef;
        }
        else if (movetype == 2) {
                return rollingfrictioncoef;
        }
        else {
                return -1.0;
        }
}

//returns object mass
float WorldObject::GetObjectMass(void) {
        return objmass;
}

//returns ID of last message recieved
```

```
int WorldObject::GetMessageID(void) {
        return messageID;
}

//Returns ID of object
int WorldObject::GetObjectID(void) {
        return objectID;
}

//returns integer describing type of object movement
int WorldObject::GetMoveType(void) {
        return movetype;
}

//sets ID of last message recieved
int WorldObject::SetMessageID(int messID) {
        messageID = messID;
        return 1;
}

//Destructor
WorldObject::~WorldObject(void) {
}

#endif
```

```
Dig.h
/*Dig object. Provides functions for the creation and manipulation of a DIG*/

#ifndef _DIG_H_
#define _DIG_H_
#include <deque>

class Dig {
private:
    int digID;          //Unique ID of DIG, also position of DIG list of DIGS's
    deque<int> DIG;     //DIG data structure, stores ID's of in objects
    int digsize;        //Size of dig
public:
        //constructor for object type definition
        Dig(void);
        //constructor for creating DIG
        Dig(int dgID, int objID);
        //function to update a DIG
        int update(int objid[]);
        //function to return ID of object in DIG at index
        int GetDigElement(int index);
        //function to getsize of DIG
        int GetDigSize(void);
        //destructor
        ~Dig(void);
};


//Empty constructor for defining, objects of type DIG
Dig::Dig(void) {
}

//function to create a dig - constructor
Dig::Dig(int dgID, int objID) {
        digID = dgID;
        DIG.push_back(objID); //First element of a DIG is the object it relates to
        digsize = 1;
        }

//function to update a dig, by adding or removing an object
int Dig::update(int objid[]) {

        int DIGObj = DIG.front();
        DIG.clear();
        DIG.push_back(DIGObj);
        for (int i = 0; i < digsize; i++) {
                DIG.push_back(objid[i+1]);
        }

        int digsize = sizeof objid + 1;
        return 1;
}
//function to get element in DIG
int Dig::GetDigElement(int index) {
                return DIG.at(index);
}

//function to get size of DIG
int Dig::GetDigSize(void) {
        return digsize;
}

//function to delete a dig - destructor
Dig::~Dig(void) {
}

#endif
```

```
Message.h
/*Class to create messages to be passed by system*/

#ifndef _MESSAGE_H_
#define _MESSAGE_H_
#include <string>

class Message {
private:
        int destobj;    //ID of target object
        int sourceobj; //ID of object sending message
        int messID;             //ID of message (also acts as timestamp for message)
        string command;         //Type of command being sent

public:
        //constructor for object type definiton
        Message(void);
        //constructor to create message type
        Message(int dest, int source, int ID, string com);
        //destructor
        ~Message(void);
};


//constructor for object type definiton
Message::Message(void) {
}

//constructor to create message
Message::Message(int dest, int source, int ID, string com) {
        destobj = dest;
        sourceobj = source;
        messID = ID;
        command = com;
}

//destructor
Message::~Message(void) {
}
#endif
```

```
/*CoulombFriction.cpp*/
/*Calculates simple friction approximation based on the coulomb model*/
#ifndef _COULOMBFRICTION_H_
#define _COULOMBFRICTION_H_

#include "mayerik.h"

MAV_matrix velocity;            //objects intial velocity
MAV_vector frictionforce;       //force of friction on object
float frictioncoef;             //coefficient of friction of object
float mass;                     //mass of object
int movement;                   //object movement 0 = static, 1 = sliding, 2 = rolling
float gravity;                  //gravity of world
MAV_matrix newvelocity;         //velocity of object, after friction calculated
float motiveforce;              //motive force on object
float staticforce;              //friction force opposing motion of static object
float frictforce;               //friction force on moving object
MAV_vector trans;               //translation terms of matrix

//function to calculate change in position after friction effects considered
MAV_matrix ObjectMovement(MAV_matrix vel, float fcoef, float m, int move, float grav)
{
   movement = move;
   frictioncoef = fcoef;
   mass = m;
   velocity = vel;
   gravity = grav;

   if (movement == 0) { //object static
        //calculate force of friction to be overcome
        staticforce = frictioncoef*mass*gravity;
        //calculate magnitude of motive force
        motiveforce = mass*mav_vectorMag(mav_matrixXYZGet(velocity));
        if (staticforce >= motiveforce) {
        //return no movement
            newvelocity = mav_matrixSet(0,0,0,0,0,0);
            return newvelocity;
        }
        else if (staticforce < motiveforce) {
            //set angular velocity
            newvelocity = mav_matrixScaleSet(velocity, (motiveforce-staticforce));
            //calculate velocity
            trans = mav_vectorScalar(mav_matrixXYZGet(velocity), (motiveforce-
staticforce));
            newvelocity = mav_matrixXYZSet(velocity, trans);
            return newvelocity;
        }
        else {
            return mav_matrixSet(0,0,0,0,0,0);
        }

   }
   else if (movement == 1 || movement == 2) { // object sliding or rolling
        //calculate magnitude of motive force
        motiveforce = mass*mav_vectorMag(mav_matrixXYZGet(velocity));
        //calculate magnitide of retarding friction force
        frictforce = frictioncoef*mass*gravity;
        //set angular velocity
        newvelocity = mav_matrixScaleSet(velocity, (motiveforce-frictforce));
        //calculate velocity
        trans = mav_vectorScalar(mav_matrixXYZGet(velocity), (motiveforce-frictforce));
        newvelocity = mav_matrixXYZSet(velocity, trans);
        return newvelocity;
   }
   else {
        return mav_matrixSet(0,0,0,0,0,0);
   }

}

#endif
```

Appendix B

These are the scenarios that would have been developed if the implementation had succeeded. Each scenario is designed to increase in complexity over the previous one.

Scenario 1 – Box with initial impulse

In this scenario the system would consist of a box on a flat plane. The user would be able to cause the box to experience an impulse of a magnitude they could decide. This is designed to test if basic friction effects can be modeled using the suggested approach.

The expected outcome is depending on the force applied the object would either not move, or move but slow down to a stop after a distance. The higher the force applied the faster the box should travel and the longer it should take to reach a stop.

Scenario 2 – Box on a slope

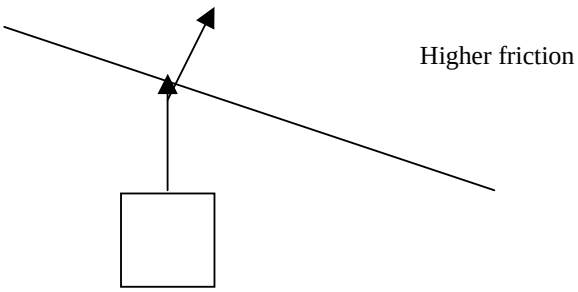
In this scenario the box would be resting on an inclined plane. The user would have control of the level of the planes inclination. This is designed to show if the approach can deal with friction when an object has a continuous motive force.

The expected outcome would be the box would not move, until the plane was sufficiently inclined. Once the box had started to move, the object would

Scenario 3 – Box on a plane with varying friction

In this scenario the ground plane would consist of two halves with different friction values. The box could be given either an initial impulse or a continuous motive force. This is designed to show if the system can cope with object passing from one objects influence to another objects influence, and the resulting change in the friction coefficient.

The expected outcome would be for the box to change speed it crosses to the different friction area, slowing if moving on to a higher friction surface and speeding if moving to a lower friction surface. If the box crosses the boundary at an angle, the path of its trajectory should bend just as it crosses the line.



Scenario 4 – Box resting on top of a second larger box

In this scenario the simulations consists of a box resting on the ground plane, with a smaller box on top of it. The box on top should be given an impulse force big enough to push it off the top box. This is to test whether the method for modelling friction changes when one object leaves one objects influence and enters another, as well as testing how well this form of friction links with collision.

The expected outcome is it should collide with the ground and possibly then slide on a little further.

Scenario 5 – Box resting on top of a second larger box with varying friction

This scenario is as before but the surface between the two boxes has a very high friction coefficient and the surface between the ground plane has a lower friction coefficient. Either the top or bottom box should be given an impulse sufficient to make them move, as well as an impulse insufficient to make them move. This is to test how effectively the message passing system deals with impulses on one object causing motion in another due to the resistance from friction between them.

The expected outcome it that the smaller box when pushed should slide off the lower box, but may also cause the lower box to slide with respect to the ground plane. Applying the impulse to the lower box should cause it to move with the upper box staying in the same relative position to it. If the friction between the two boxes is high enough, it may also be possible to move the lower box, by applying a force to the higher one.