Virtual physics for virtual reality/games modelling

D P Jones

BSc (Hons) Computer Software Theory

2005

Abstract

Physical modelling is a popular field due to its heavy use in computer games and VR. This technology has progressed to very advanced levels, such that incredibly realistic effects are included in games. Despite this apparent realism, there may be some aspects of the physical modelling process that are ignored as they would bring little benefit to the game. This project will involve a study and attempt to implement on of these lesser used physical models in the form of a temperature model. The aim is to display that exchanges of thermal energy can be modelled with a program.

Acknowledgements

Thanks to:

My supervisor, Prof. P Willis
My personal tutor, Dr. A M Barry

Contents

Virtual physics for virtual reality/games modelling

Submitted by……………………

____

Project Proposal
Virtual physics for virtual reality/games modelling

Original Brief

Basic physics is what makes the world behave in a familiar way. Most VE/games worlds concentrate on the graphics and geometry, and then add the physics afterwards. This makes it hard to get the world to behave as expected. Previous projects in 2004 examined how to model the physics; for example to get collisions and mechanisms to behave as expected. This project will examine ho to 'model' a virtual world on physical principles: in particular, objects should sit on other objects because of action-reaction, not because of some arbitrary transform.

Problem

It is a fairly typical approach with, for example, games to focus on graphical appeal before realistic physics. Although many recent games have advanced notably in some areas, such as dynamic shadows, very few make any use of real physics. Some games in particular, such as the Thief series, make great use of light and shadow as part of the game playing experience, and, specifically, Thief 3 actually makes use of dynamic shadows and light sources. Another example of use of light physics in games is Aliens vs. Predator. This game allowed players to actually destroy most light sources, thus leaving areas in complete darkness.

It has become a growing trend in games, first person shooters in particular, to feature breakable objects, e.g. windows, crates, vehicles and even, occasionally, walls, however it is rare for a game to offer any further interaction with the environment, other than leaving bullet holes in the wall!

One fairly recent game is notable for its slightly more interactive levels; Red Faction. This game allowed players to actually destroy walls and floors in the levels to create alternative routes to the end goal, however even this did not offer totally accurate or complete interaction with everything in the levels. There are some fairly realistic anatomy models used with enemies in such games, but this tends to simply limit how many bullets they can take in certain locations.

An obvious, but often overlooked, effect is the inability to look down and see the player character's body (from a first person perspective). This feature is implemented by very few games indeed. A good example of one is Jurassic Park: Trespasser. This game allowed a player to see their own body. Moreover interaction with the game world was done through controlling one of the character's arms; with the elbow, wrist and hand all controlled independently. In this example there was some quite realistic physical interaction with the game world, e.g. being able to press different buttons on a phone keypad and being able to pick up and throw rocks and branches on the floor.

In general modelling accurate and detailed physics within a game world tends to be overlooked, when the effect we wish to generate can simply be scripted. If a key is turned in a car, we expect the engine to start. This can be fairly easily scripted by simply linking the engine sound effects and the moving vehicle to the key. If we consider a more detailed physical model, the player might, for example, need to ensure the gear stick were in neutral and that there was fuel in the car before it would start.

First person games have a lot in common with Virtual Reality, in that the intention is to try and immerse the user, or player, in a 3D world. Although the interface may differ, the actual creation of the 3D world is not likely to be very different, since the goal of most games is to make a believable 3D environment.

For the purpose of this project, the aim will be to construct a 3D model of some kind of system, such as a mechanical system. An example could be a physically accurate 3D model of an engine, allowing interaction such as removing the battery, or restricting the fuel line. Another example could be a reservoir connected to a dam which generates electricity and flows into a stream which then operates a watermill.

Project

To construct an interactive environment, one room, containing a number of devices that demonstrate a higher level of user interaction than normal. For example an office containing items like a desktop fan with multiple speed settings, or other common electrical devices. Electricity flow will be modelled in the environment, so wall mounted power sockets must be implemented. Also a light source and controlling light switch will be implemented. Alongside these there can also be a main fuse controlling the electricity supply to the complete environment.
Depending on the time taken to complete the project, it can be extended to include accurate modelling of gravity and permit interaction with objects such as shelves or furniture that may be featured in the environment.
The electricity model will have a base value, e.g. 10, that will be supplied to electrical devices. In the case of a fan, the different speed settings could correspond to how the electrical current reaching the motor can be restricted.
Modelling every mechanical system in perfect detail is unnecessary, since this would only be important if the user can see it in action. Any such system that the user can actually see should be modelled in full, or as completely as possible, however if the user is not watching the system in action, it does not require such a high level of detail.

Requirements

- Identify common physical interactions in such an environment so that they can be correctly modelled.
- Create a working model of an electricity flow throughout the environment with a base value that will be supplied to all electrical devices.
- Create a visually, and possibly mechanically, accurate model of any electrical device to be featured in the final environment.
- The environment should be sufficiently complex to appear as though everything is realistically modelled; however the environment should be simplified where possible to reduce the required machine specs.
- The environment will need rigorous testing to ensure the user cannot perform some action to cause any error.

Potential Expansion

- Accurately model gravity and allow for any consequences this will cause between items in the environment.
- Implement a physically accurate light model allowing dynamic shadows. A (slightly extreme) example of this could be the effect of a swinging light bulb.

Objectives

- Research common physical reactions that will be applied in the 3D environment.
- Identify complete and specific requirements to create the desired end result.
- Carefully design the environment and how items will interact.
- Work out exhaustive Test Plan to create a working end result.
- Code and test the environment, recording each version of code and identifying modifications made during testing.
- Evaluate final result and attempt to identify future directions or any area in need of obvious improvement.
- Maintain a complete record of all texts, books and sources of information used for, and relevant to, this project.

Resources

- Texts on 3D modelling and physics are readily available on the internet and books on this topic are fairly common.
- There will be access to previous projects related to this topic during the project.
- As well as the project supervisor, a PhD student attempting a related project will also be available for consultation.

Introduction

As described by the abstract, the relevant area of study relating to this project is to do with physical modeling for virtual reality and/or games. Given the current level of physical models in many recent games, it is obvious that this is not an uncommon subject. We can take a good example in the well known Half Life 2. A game featuring very impressive and realistic visual effects, as well as interactive ones. The ability to break boxes, windows and other solid items, and the realistic lighting effects result in a very realistic experience. Since this project is of a much lower level, rather than trying to cover the same sort of ground with breakable objects, or realistic lights, this project will focus on some less common areas. In various action games, similar levels of interaction to those described above are quite common, however since these games are action based, many details present in real world situations may be omitted since they are not essential to the game experience. This is fine in games; as such things would most likely be needless extra features, e.g. electricity. Although there are places where light switches can be turned on and off in Half Life 2, it is highly unlikely the game bothers to model the wire connecting the switch to the light and the power supply. If we go into this in more detail, we could mention that the material and length of the wire could be modeled, so that the brightness of the bulb controlled by the light switch, determined by the power supply, the properties of the wire, the switch and the bulb itself is calculated accurately. This level of detail is not likely to be necessary in a game, or probably even in virtual reality, as this kind of model is not something that has many visible aspects. However, it seems something of a shortcut to partly model a system such as this, e.g. a light switch connected to a light, without going into much detail.

Another similar example is temperature. Very few games make use of models of temperature, or, indeed, many forms of energy exchange. For this reason, the main focus of this project will be to create a visual temperature model. The aim will be something like an infrared camera view; hot things appear very bright, cold things are dark. Additionally, things that are near to sources of heat will be warmer than things that are far away. At least some attempt will be made to model the exchange of thermal energy. In order for this to be achieved, modeling of solids and material properties should be implemented.

Modeling of additional features, such as light and light based effects will only be attempted if there sufficient time.

Literature Review

Original Problem

Basic physics is what makes the world behave in a familiar way. Most VE/games worlds concentrate on the graphics and geometry, and then add the physics afterwards. This makes it hard to get the world to behave as expected. Previous projects in 2004 examined how to model the physics; for example to get collisions and mechanisms to behave as expected. This project will examine ho to 'model' a virtual world on physical principles: in particular, objects should sit on other objects because of action-reaction, not because of some arbitrary transform.

Literature Review

As stated in the original brief, this project branches into Virtual Reality (VR) which is the subject of a number of papers. Many papers are concerned with the basics of what VR is, creating an artificial environment in which to immerse a user. A fairly good description of the basis of VR is "create the effect of interacting with things, not with pictures of things ($1)". "Rather than attaching physical properties to geometric shapes, we directly design and manipulate objects with intrinsic dynamic properties such as stiffness, mass, angular momentum. Flexible and rigid bodies can interact in the same simulated physical environment, responding to collisions, fluid velocity fields, friction and gravity ($5)" is a more low level way of describing VR. Also, it should be noted that "it is the interface, not the content that characterises virtual reality ($1)", better explained as "VR is the term used to describe advanced methods of involvement and interaction for humans with a computer-generated graphical (usually 3D) environment ($2)". The most important part of VR is how the user interfaces with the computer, requiring special equipment that gives direct sensory input. In this way, it becomes obvious of the merits of VR for simulation and training purposes. The most important aspect with such VR simulations is realistic physical behaviour. Unless everything within the VR world behaves just as the real world equivalent would, this will defeat the object of simulating it. Not having access to advanced VR equipment for this project, it will focus instead on the simulation aspect. This aspect also ties slightly into 3D computer games, in that some simulation is often used but correct modelling is rarely applied when something like an in-game script and graphical effects can replicate the desired result, yet be far less demanding on the machine; "Real-time physical simulation requires significant computing resources ($3)". However, from the game point of view, fast and smooth graphics are more important than correct physics, though with powerful enough machines this is not an issue.

The training uses of VR have been considered in many safety critical areas, e.g. "Fire-fighter command training virtual environment" (Tazama U. St. Julien, Chris D. Shaw, 2003) and "Medical applications of virtual reality" (Russ Zajtchuk, Richard M. Satava, 1997). The obvious merit of VR in these specific applications is that there is no real danger to either the trainee or any subjects or victims involved. More than this, VR training in this form is potentially far faster than hand training people to do the same jobs. This being particularly notable in the case of, say, a surgeon, who would usually need many years of medical training before being allowed into the professional field, as evidenced by "virtual reality has been used to train surgeons in the new procedures and can determine their competence level before going 'live' (Ota et al (1995))" ($4). The comment was in relation to a paper by Delingette et al (1995) which presented a craniofacial surgery simulation test bed which makes extensive use of virtual reality techniques. Also Merril, Raju and Merril (1995) presented a prototype virtual reality surgical simulator for laser iridotomy training. The aim of this being to help physicians acquire necessary skills without patient risk ($4). There many examples of papers on proposed VR training in medical fields, e.g. Zeigler et al (1995a and 1995b) and Coleman (1995) are just a couple. As mentioned earlier, VR training extends beyond medical applications, a few examples of which are a Zelter, Pioch and Aviles (1995) paper on training for a US Navy submarine deck officer ($4), and a Hollands and Mort (1995) paper on VR to visualise the simulation of manufacturing systems ($4), which helped users unfamiliar with the system to understand it and those who were familiar with it to easily recognise it. Another example of the use of VR in simulation comes from Suzuki et al (1995) describing a driving simulator that allows the user to drive through a virtual town ($4). This leads on to the mention of ROADNET (Katz 1995), a proposed system which utilises distributed interactive simulation technology to support a network of vehicle simulators ($4). Similarly Stevens and Neal (1995) proposed use of a distributed interactive simulation to support air traffic control simulation ($4). Misumaki et al (1995) discussed an indoor distribution simulator system based on virtual reality technology. The system was designed to give the user the feeling of touring a substation for routine inspection. It was proposed that the validation of equipment operating procedures, training of unskilled workers and work schedule preparation could be improved by the use of this system ($4). It is obvious from some of these examples that VR training and simulations cover more than just safety critical areas. A common use of simulation is in science which could be beneficial if used in the case of dangerous or expensive experiments, where components can be accurately modelled then the experiment performed many times in a virtual environment. An example of VR actually applied in a scientific field is molecular modelling; allowing users to actually see the interaction of molecules in order to better understand it. The overall point here is that VR has uses in a vast number of areas, thus there are many papers on its proposed uses in these. Here is a brief list of a few papers on varied fields involving VR simulation; Sims (1995) paper on VR applications for experimentation with evacuation scenarios, Bethel Jacobsen and Holland (1995) paper on the combination of simulation with VR and visualisation tools to aid training in environmental remediation, Bienvenue, Curtis and Thakkar (1995) paper on VR in the US classroom for teaching and learning, Wakefield and O'Brien (1995) paper on VR for planning construction activities, Bowen-Loftin (1995) paper on VR for training NASA astronauts and ground-based personnel, and Mastaglio and Callahan (1995) paper on a VR environment called the Close Combat Tactical Trainer, or CCTT ($4), and a large number of papers on scientific applications of VR by authors such as S. Bryson, C. Cruz-Neira and assorted co-authors each, among many.

Since VR can be used in so many areas to create realistic and, above all, believable models of real systems, it should be obvious that it can be of great use in the media.

VR to some degree is common within 3D games, but not often complete. VR can potentially be applied in films, for example as a method of generating realistic SFX but at lower cost than staging them for real. It also has a use in the creation of fully 3D animated movies, where simple physical reactions could cascade into on screen effects, "using physics to determine the motion of objects adds realistic effects to animations. Animations are automatically created from an initial set of physical properties and models that are visually appealing and communicate information about how different models interact in a simulated environment ($5)", e.g. a bullet striking a gas tank causing an explosion; this could be initiated by a human character simply being instructed to fire said bullet and then recording the resulting sequence.

As mentioned, a major use is in computer gaming. Newer games often strive to provide the user a more interactive and involving experience by presenting them with increasingly convincing and realistic game worlds. The main limiting factor in this is through the actual user interface method. Full VR interface has a far more in-depth experience, but the challenge of building the actual interface is far greater. "Conventional interfaces typically contain only virtual computer-generated objects. Virtual reality interfaces, on the other hand, may contain both virtual and physical objects that coexist and exchange information with each other, as in the case of augmented reality systems. The need to properly align virtual and physical objects constitutes an additional challenge in virtual reality interface design ($6)". A more in depth note, "In conventional interfaces, objects usually exhibit passive behaviours. In general, they have predetermined behaviours that are activated in response to user actions. Therefore, communication patterns among objects are usually deterministic. Virtual reality interfaces contain both real world-like objects and magical objects that exhibit autonomous behaviours. Unlike passive objects, autonomous objects can change their own states. They can communicate with each other and affect each other's behaviours and communication patterns. Therefore, designing object behaviours is more challenging in virtual reality interfaces ($6)". Details of the design involved in the creation of a VR interface can be found in "VRID: A Design Model and Methodology for Developing Virtual Reality Interfaces", a paper by V. Tanriverdi, R.J.K. Jacob ($6), which also references a large number of papers related to the main topic here.

The methods of modelling VR are highly relevant to this project, since the use of real VR interfaces is not possible. "We refer to three types of models: a *geometric model*, a *physical model* and a *surface model*. A *geometric model* is simply a mathematical description of an object. The mathematical description could be surface definitions, networks of vertices and edges, sets of volumetric primitives as in the case of CSG, or any other geometric entity. A *physical model* is the combination of a geometric model with physical properties or attributes of an object at an instant of time. Each physical attribute is usually coupled with a specific geometric primitive. Physical models are input to a simulated environment where they react with forces and other physical entities. A *surface model* is a subset of the *geometric model*. It is a representation of the external surfaces of an object. The *surface model* is important for visualising the objects that are being simulated ($5)". The paper by P. Sweeney, A. Norton, R. Bacon, D. Haumann (1991), Modelling Physical Objects for Simulation ($5) is of great interest in this case. Also, the paper by M. Cavazza, S. Heartly, J. Lugrin, M. Le Bras, Qualitative Physics in Virtual Environments ($3) describes the creation of VR environment at fairly low level. One thing to note is that the project detailed in this paper made use of a game engine, in particular the engine from Unreal Tournament 2003 (UT2003), a fairly recent game which is quite well known for its use of 3D. The use of Qualitative Processes (QP) is described in this paper, which are predefined processes that some virtual environment (VE) objects can be involved with. The QP's are triggered by the user, when specific preconditions have been satisfied. "The environment remains interactive while QP's are active, as QP's themselves are simulated in user real-time, updating the environment's object's properties as landmark values are reached for physical properties ($3)". Another paper that describes low level development of simulation design and implementation is GMSS Graphic Modelling and Simulation System by R.R. Willis and W.P. Austell.

A subject closely tied with VR is the notion of augmented reality (AR). This involves elements of VR overlaying real world environments. As with full VR, the interface required for AR is very complex, but does not involve immersing the user in a completely simulated world. AR allows the user to see the real world around them, but it simulates additional objects or entities to 'augment' this. The interface must be able to accurately relate simulated objects to real ones, and to keep track of the user location with the AR environment at all times. Because it overlays entities into a real setting, it is highly important that the AR model of the environment correctly copies the real environment; failure to do this can result in AR objects appearing in unusual, or even impossible, places, among other obvious bugs. While not as common as VR, AR has a number of papers relating to it; Interactive Augmented Reality Techniques for Construction at a Distance of 3D Geometry by W. Piekarski and B.H. Thomas (2003) describes some methods involved in AR at high level. First Person Indoor/Outdoor Augmented Reality Application: ARQuake by B.H. Thomas, B. Close, J. Donoghue, J. Squires, P. De Bondi and W. Pierkarski (2002) explains the development of an interpretation of the well known computer game, Quake, as an AR game. While much of the paper deals with the interface between user and AR world, and how this is achieved, it is a good example of a method of immersing the user in a slightly different kind of 'realistic' game world.

Sources

$1: Virtual Reality in Scientific Visualisation

Steve Bryson

$2: Virtual Reality and Simulation
Martin Barnes
1996

$3: Qualitative Physics in Virtual Environments
M. Cavazza, S. Heartly, J. Lugrin, M. Le Bras

$4: Virtual Reality and Simulation: An Overview
R. Macredie, S.J.E. Taylor, X. Yu, R. Keeble
1996

$5: Modelling Physical Objects for Simulation
P. Sweeney, A. Norton, R. Bacon, D. Haumann
1991

$6: VRID: A Design Model and Methodology for Developing Virtual Reality Interfaces
V. Tanriverdi and R.J.K. Jacob

The ACM Digital Library (http://portal.acm.org)

Requirements Document

Requirements Draft

This is, initially, a list of all the things this project should aim for, but not necessarily all the things it will do. Some aspects may be too complicated to implement and some may simply take too long. Occasionally requirements may interfere with each other, in which case the requirements themselves will need to be altered.

- **Construct a 3D environment**
- **Construct 3D models of common real world items**
- **Allow easy placement of any number of items in world**
- **Allow (some) user interaction with 3D objects**
- **Model air currents**
- **Model electric current flows**
- **Model electrical devices to apply electric current**
- **Implement diffuse and specular lighting**
- **Implement shadows**
- **Implement reflections**
- **Model physical interaction between objects correctly**
- **Implement Dynamic shadows**
- **Model gravity and gravitational effects correctly**
- **Construct an accurate temperature model**
- **Provide a thermal view mode for the user to experience this feature**
- **Model user interface to behave in a realistic manner**
- **Model user as a world object to allow physically correct interaction**
- **Provide user quality settings in order to use simulation on lower spec machines**
- **Construct objects with appropriate modelling technique**
- **Consider the type of material for models when applying lighting**
- **Consider the type of material for models when applying temperature**
- **Consider the type of material for models when applying gravity**

Requirements Analysis

- **Construct a 3D environment**

A 3D environment is essential, as this is the basis for the project. Solid modelling is most appropriate for this situation, so the CSG method will be considered to construct the world. When considering the temperature model, a surface model seems far more logical for a realistic appearance, however such a model would not model physical properties as well as solid modelling.

- **Allow (some) user interaction with 3D objects**

If possible, user should be allowed a wide range of interaction options; however this will depend on time and difficulty constraints. Ideally we would like the user interface to lend itself to any imaginable interaction, e.g. interface through a physical model of a hand which then interacts with other models as normal.

- **Model air currents**

Since we are considering modelling common items, one such item may be a desktop fan. This can demonstrate the model of electric current and we can also use this to demonstrate air currents to some effect. This would be most obvious if something like cloth simulation were applied in this project; however that may be too complex to implement. More simply, we can just apply it in the case of very light items. We can consider, for future improvements, trying to implement a cloth simulation, so that we could try to build a more realistic environment, e.g. a 3D model of the inside of a house, where we would need a cloth simulation for items such as curtains which would more impressively demonstrate the modelling of air currents.

- **Model electrical devices and electric current flows**

Allow user to interact with switches, etc, in order to see electricity model in evidence. We want to include some models of everyday electrical appliances for this purpose. Since the simulation is based on reality, such objects would be commonplace. The main feature of this would be to allow the user to turn light sources on or off at will. To some extent, we should like the electrical applications to respond to an electricity supply; however this would mostly be minor details, not essential to the feature.

- **Construct an appropriate lighting model**

Simple lighting effects like shadows, specular and diffuse lighting should be achievable through a simple ray tracing method, however the Radiosity method may lend itself to some of the desired aspects of this project.

- **Implement reflection and refraction**

This is possible using the ray tracing rendering method; however these are not so simple with the Radiosity technique.

- **Model physical interaction between objects correctly**

If objects collide, we expect them to behave as real world objects would; taking collision as the most basic interaction we can come up with. In this way, the solid model is an obvious choice. We can give it a material that has several properties, e.g. density, then use the physical model with these for realistic effects, e.g. combining volume and density will give us a mass for an object, which is applicable in the case of a gravity model.

- **Implement Dynamic shadows**

This may be too difficult to implement, but may be attempted anyway. In order for this to work, it seems likely we will need to construct a model for the user's body. Since we have mentioned we would like interaction to take place through a hand(s) model, this is not an unreasonable expectation.

- **Model gravity and gravitational effects correctly**

This may be too difficult to implement, so is unlikely to be attempted. The inclusion of gravity complicates physical interactions between models in that it introduces a downwards force that must be constantly kept in mind. If correctly modelled, we can easily perform accurate throwing and dropping of objects. This, in conjunction with calculations for the object material and dropping height, can allow us to consider breakable objects. If this cannot be implemented it should be noted for future improvements.

- **Construct a temperature model**

The main feature of this project. This may be quite similar to the Radiosity rendering method and, as such, may be implemented similarly; however a difficulty with this is the Radiosity technique leans towards surface models and is less effective for solid models. Another problem is that Radiosity is less effective in real-time, as this would be. We would need to construct an effective combination of surface and solid models and combine Radiosity and ray tracing to model temperature as we desire. A temperature only simulation would not be impossible, assuming it is implemented in a similar way to Radiosity. We simply use only surface models and assign colours based on the temperature values; however this would severely restrict all physical object interactions, which is the main requirement of this project. Ideally we want to preserve interactions, for a more real simulation, and allow this temperature model, for a more complex physical simulation.

- **Provide user visual effects of temperature model**

As described, this may be implemented in a similar fashion to Radiosity, where the colours the user sees will be determined by the temperature value of objects. This can probably be modelled quite closely with a variation on the Radiosity technique; however this technique is hard to apply in the case of solid modelling, which is our main focus here. We could allow the user to take a "thermal image", i.e. give the option to render the user's view with the temperature model and save an image that displays the world as we would like the thermal view to appear. This will likely be implemented first in order to try and build up to a fully accurate thermal view.

- **Construct objects with appropriate modelling technique**

Using a surface model would probably give a nicer look to the 3D world; however a solid model would lend itself to more realistic physics. In the case of thermal vision we would prefer a surface model, as this works well with Radiosity, however, as mentioned; this also limits physical interactions between objects, which remove a lot of realism from the simulation. It is more important to model these interactions than additional features, such as the thermal view, so our first consideration would be the solid model, however the temperature model helps move the simulation towards even more accurate physical properties and appearance, thus we should try to find a method of combining solid and surface models to some extent, in the hope this will function with all desired features we can implement.

Functional Requirements

These requirements are the essential features that will allow the project to work in the most basic form. We are considering these features to be the minimum to be implemented in order to have created our 3D environment, many of which the 3D world cannot be created without.

- **Construct a 3D environment** - This is precisely what it says. The basis of this project is creating a virtual world, so this is obviously a requirement.
- **Construct 3D models of common real world items** - We want a realistic setting, so we will need to model at least a few common household objects, including some that can demonstrate the more advanced features we hope to implement. If possible, we would like to construct a large enough selection that the user can find most objects they may desire available for our 3D world. In practice, we cannot offer them ALL objects they may desire, as this is a ridiculous request, but we hope to allow them to define new models to add to the world.
- **Allow placement of any number of items in world** - We would like for our world to be modified/expanded as the user desires. At the very least we should allow them add more of our defined objects wherever they desire them in the world.
- **Construct objects with appropriate modelling technique** - We have not yet defined what is 'appropriate', and our definition of such may change later during the implementation of more advanced features of the project, however, on the most basic level, we will expect objects to exist in the 3D world, and we will expect them to look and behave like solid objects reliably.
- **Construct a temperature model** - Similar to electricity, modelling this also leads towards a realistic simulation. We expect objects to change their temperature based on the distance from a heat source. If some kind of real-time or time based solution can be achieved, we can even watch object temperatures actively changing in response to heat sources.
- **Consider the type of material for models when applying temperature** - We will give this feature higher priority than its light model counterpart, since we are strongly focusing on creating a temperature model in this project. This is only really important if a time based solution can be established, as material properties for a temperature model will mainly determine how long an object takes to change its temperature.
- **Provide a means of demonstrating this feature to the user** - To demonstrate a temperature model it seems obvious we need to be able to see a difference in temperatures between models. Creating a temperature based view, e.g. similar to an infrared camera, would demonstrate the temperature model in use.

Non-Functional Requirements

These are the features that we would like to have implemented in our ideal simulation, but do not consider them essential to our simulation. We give them a lower priority than functional requirements, but we should still expect them to be included in the final simulation. Merely being non-functional does not mean they can be omitted, it simply means they have no impact on our definition of a working 3D world.

- **Model electric current flows** - This is one of the features that assist in demonstrating a realistic model. While a complete and accurate model of electrical currents could be quite challenging, a basic model with on/off settings should not be a problem.
- **Model electrical devices to apply electric current** - If we can implement an electrical model, we obviously need some way to show its use. Creating models of some form of electrical equipment seems an obvious choice. Even better if they can display some visible effect from the electricity model, e.g. light bulbs, electric heaters.
- **Allow (some) user interaction with 3D objects** - Since this project is focusing on creating a realistic environment for the user, we would like some level of interactivity. At the least, users would be allowed to move around freely, though we would prefer to also be able to push solid objects, however if we are going to model the user as an object, this will be the only kind of interaction allowed, though the consideration of 'where' to push then becomes far more important.
- **Model physical interaction between objects correctly** - While absolutely precise calculations for interactions are not essential, we would like something close that will at least cause the world to display these interactions as we expect them to appear.
- **Model user as a world object to allow physically correct interaction** - At the very least, we want the user to behave as solid, i.e. not walking through objects. We will expect more than this, though, since we require physical interaction to be modelled sufficiently well, and by considering the user as an object, we expect the user to follow these same rules as a consequence.
- **Implement lighting** - A complicated system of lighting is not essential for the project, but it will look rather bland without such. Although we require lighting in order to see the 3D world, we would like a more advanced lighting model to be implemented that will demonstrate some realistic lighting effects as well.
- **Implement shadows, diffuse and specular effects** - While these are an important part of building an accurate light model, this is a very common feature in 3D modelling. Since our focus here is to model some less common features, we give the prettier lighting effects a fairly low priority.
- **Implement reflections** - A reflective model is quite possible while we consider a ray tracing method to displaying our model, however as described above, we give low priority to "pretty" lighting effects.
- **Consider the type of material for models when applying lighting** - This is fairly easy to model by storing specific ordered data about different materials, then applying it when displaying the image, but this is also grouped with the low priority of lighting effects, as it would mainly serve to work with reflective and, diffuse and specular effects.

Optional Requirements

These are features that we give the lowest priority, either because they are considered too difficult or time consuming to be implemented. Given sufficient spare time during the project, there is a chance these will also be attempted, however we do not consider them either essential to our working simulation or necessary for our simulation to be considered successful.

- **Model air currents** - This could be a highly challenging feature, similar to modelling gravity. It would involve considering ways to model some kinds of object that respond to air currents, e.g. cloth models, which could be very complex indeed.
- **Implement Dynamic shadows** - This is also a challenging feature. Obviously only possible to implement in a real-time solution, this would, above all, demonstrate a very advanced lighting model. Since our focus is not on creating a light model, however, we will not assign much importance to this feature.
- **Model gravity and gravitational effects correctly** - Realistic modelling of gravity is very challenging and above the level of this project. Naturally, a gravity model is needed for a complete realistic model; however this is another feature that requires a real-time solution to be worth implementing.

- **Consider the type of material for models when applying gravity** - If gravity is modelled well, we can make use of different material properties to determine the weight of each object, thus how they react with gravity.

Test Plan Draft

Are models constructed?
- Does the primitive array(s) contain primitive data?
- Does the object array(s) contain object data?
- Does the world array(s) contain world data?

Are models correct?
- Are primitives stored correctly?
- Are objects stored correctly?
- Is the world stored correctly?

Is world displayed?
- Do primitives appear?
- Do objects appear?
- Is anything missing from the world?

Is display correct?
- Are primitives oriented correctly?
- Are objects constructed and oriented correctly?
- Does the world appear correct?

Does electricity model work?
Do relevant objects respond to electricity?
Does temperature model work?
Does temperature model work correctly?
Do material properties work?
- Can primitives be assigned different materials?
- Do materials have different properties?
- Do different properties cause different behaviour?
- Do materials have correct properties?

Design Document

Since this project is focusing on creating a solid, 3D world, rather than a world that merely looks 3D, is seems obvious that our choice for implementing the world uses solid modelling techniques over surface modelling. While we do want our world to look at least vaguely realistic, we are more concerned with it possessing realistic qualities. The benefits of manipulating 3D models with solid modelling compared with the difficulty of such activities with regards to surface modelling makes it an easy choice. We can note, however, that a surface model would lend itself to creating a more visually pleasing light model; we have already mentioned that we are more concerned with qualities than appearance, however. Also, our theoretical temperature model may perform better when applied to a surface model than a solid model, as it involves calculating the exchange of a kind of energy throughout surfaces in the world, very similar to the working of Radiosity, which is known to work well upon surface models. We can assume that a technique bearing similarities to Radiosity would therefore behave well in some environment that complements Radiosity. As we have given priority to a solid model approach, in order to head in the direction of a realistic model, we will need to devise a temperature model that can be applied in such an environment. A completely accurate temperature model may well involve as much calculation as Radiosity, however a simplified approach should be possible to implement and can be considered to include in a real-time or time-based solution.

We are taking the approach of creating a 3D model from scratch. In this way it is hoped that building it up allows us to make it simple to modify various aspects, and attempt to demonstrate the use of realistic physical properties. Our main focus is in making a temperature model, however to reach this point we also need to consider how to create physical models of solid objects.

Primitives

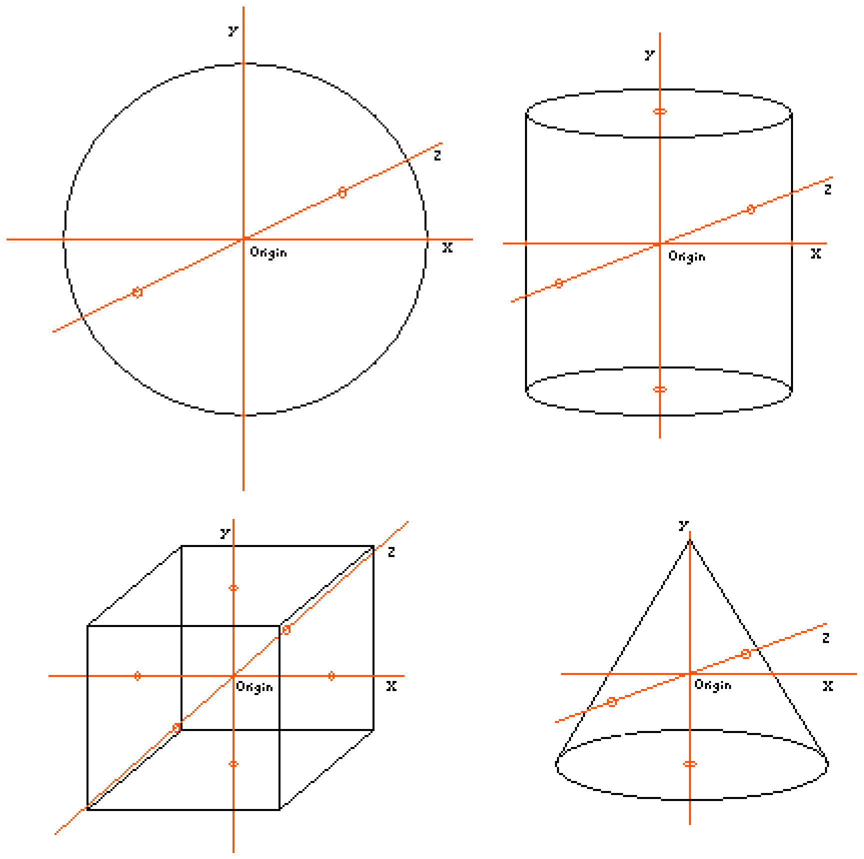
The most basic mathematical shapes available. We can create object models through various combinations of different primitives. Here is a starting list of primitives for our simulation:

- **cylinder**
- **sphere**
- **cube**
- **cone**

If others come into use during the design of our simulation, we shall add them to this list. Each of these can be defined in mathematical terms in a 3D space. For each, we will expect a centre coordinate to be supplied in the form (x, y, z). An array of 3 numbers is the simplest way to express this. We will assume the numbers are also in the order x y z, so input will be expected in this order too. For each of the primitives listed so far, we can also expect at least one value to define dimensions. Obviously, a cylinder and sphere can be defined with a radius and a radius-height pair. In the case of a cone, we shall assume the radius refers to the base, i.e. where the horizontal cross section is widest, and also assume that the top comes to a point, i.e. it approaches a radius 0. The cube will require a value similar to a radius, though not the common definition of the term. We shall call the value a radius anyway, for the sake of continuity. The "radius" of the cube will be the distance from the centre coordinates to the centre point of any one side. We now have our centre and radius. Naturally we need these values to correspond to one another.

We can fairly easily define these primitives by taking a very general approach. We will define a sphere by saying the centre is naturally at the origin and the radius is 1. This is very basic and easy to remember. Similarly, this approach works with the other primitives. A cylinder can be assumed to have a radius of 1 and a height of 1, and have centre point at the origin. Very similar to a cylinder, we have a cone primitive. We can define it in precisely the same way, and simply tell the difference by keeping a record of the type of each primitive. Our other primitive type, the cube, can be defined in almost the same way as a sphere. The main difference being we will assume each edge has a length of 1, rather than radius of 1. The vertexes that identify the corners of a cube are also easy to remember by this definition, as each vertex is a point where the coordinates are all either 1 or -1. We should note that we are not ignoring the individual dimensions of the primitive here simply that we are aware that the cube we have described is defined by a single unit dimension. If we are to consider all the dimensions that define the shape, it is obvious that we are no longer treating it as a cube, but as a "cuboid". There are two methods of defining such a shape; either we can treat it as a primitive type, or we can use our "cube" primitive type to define a "cuboid" object. The latter is a needlessly complicated method of treating this issue, but we will look at both methods, nonetheless.

Having defined a cube primitive, we should also look at a more general type of primitive, the cuboid. We can consider cuboids to be a superset of cubes, as a cube is obviously just a special case of cuboid where every edge is equal. This implies we need not make any distinction between cube and cuboid, and can adjust our list of primitive types accordingly.



Sphere, Cylinder, Cuboid and Cone.

We have considered how to make our primitives from a modelling perspective, but we also need to look at how to define them from a computer program perspective. This has multiple solutions such as defining a new data type for "primitive", or possibly a carefully ordered array or series of arrays.

This choice may depend on the programming language used, but we can still consider how to construct and use these data structures in a general way. A series of arrays is a very straightforward approach to storing primitive data. Keeping an array for the type, another for the centre points, one for rotation and one to define the dimensions. We can do without the need for an identifier if we are careful when storing primitives. So long as the array indices match, we use a value in the index as an identifier for that primitive. If we use what was mentioned above and assume the primitives have dimension values of 1 and are centred on the origin, we can use an array for the scaling values of the primitive to define its dimensions. This is very easy to follow and implement, but will require more code than creating a new data

structure would. It may become a problem if we permit too many variables in a primitive. For a simple model, this technique is suitable, though basic.
e.g.

```
define array primitive_types[number of primitives];
define 2Darray prim_centres[number of primitives][3 centre coordinates];
define 2Darray prim_rotation[number of primitives][3 values for rotation about x axis, y axis, z axis];
etc.
```

A new data type, "primitive" being created would need the following fields: identifier and type. An identifier allows us to search for particular primitives in lists of the same type. A type field tells us how the primitive is defined, and thus how to handle it. If we were to store every primitive in some kind of list using this data type, we would need a bit more. We could include fields for centre point, dimensions and rotation. In this way we can keep track of every primitive in a single list. If we include enough fields, we can store all the relevant data for each primitive in a single entry.
e.g.

```
define data type primitive
(           identifier field;
            centre array[3] field;
            rotation array[3] field;
...
)
```

Objects
We use the term "object" in this context to refer to a shape made up of one or more primitives. We intend the word "object" here to refer to a model of some real world item, although it is not limited to this definition. As we have described above, primitives can be defined in code using either a series of arrays or a new data type. The same holds true of objects, although obviously an object data type will hold slightly different fields than a primitive. Most notably, it should hold a list of all the primitives that compose the object.

Cuboids
As mentioned when discussing primitives, there are two ways we could model a cuboid. We will look at constructing it from a number of smaller cubes. Initially, we will want to construct the cuboid from as large cubes as possible, in order to keep down the number of primitives. In some cases the cuboid may divide into some number of identical cubes, which would be convenient. Obviously, if we consider small enough cubes the cuboid will always divide into a number of identical ones, however this does nothing to limit the number of primitives in the cuboid.

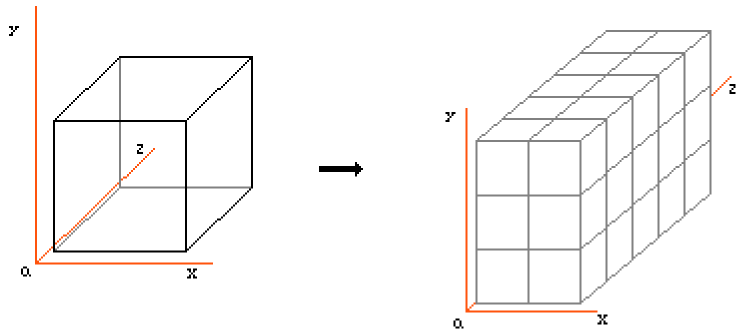
In general, the best approach would be to approximate the shape with some number of the largest cubes we can fit into it, then we fill out any missing parts with smaller cubes. Overall, this is somewhat of a complex way of constructing a simple shape, so it may best to treat a cuboid as a primitive.

Primitives in Objects
We have said that objects are composed of some number of primitives. Additionally, these primitives each have their own orientation, independent of the orientation of the object. This is why we should keep separate records of scale, rotation and translations that are performed on primitives and objects.

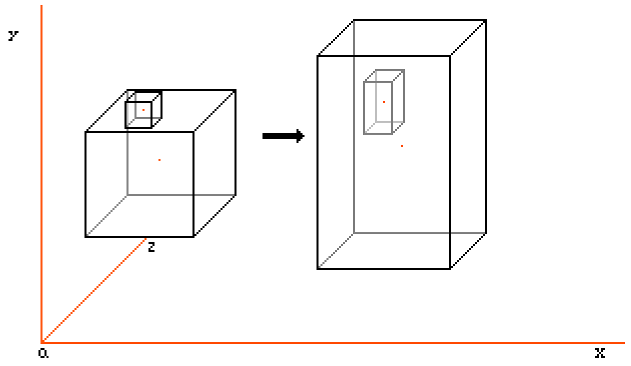
Rotation, Translation and Scaling
When we try to model a real world item it may be quite easy to see how it could be split into various primitive types, however it seems highly unlikely that these primitives will all be aligned and positioned in some ordered manner. We may need to rotate, scale or translate some primitives in order to construct some models. In a similar manner, the same can be said of objects. We may wish to perform these same operations on objects in order to try and model a real scenario.

Translation
This is a very simple operation. It merely involves adding a certain value to the x, y and z coordinates of any primitive or object in order to change its position. In the case of objects, translation can be applied to each primitive to achieve a correct effect.

Scaling
In this context, we will use to word "scaling" to refer to deformation of a primitive about any of the 3 main axes. As such, we will store it as a list of values in some specific order. The usual order is x, y, z so we shall take it as such. This means that scaling requires a list of scale values, e.g. (a, b, c). Uniform scaling is easily achieved by using a list of equal values, e.g. (a, a, a), as each axis will be scaled independently. This means the list (2, 3, 5) will work as follows; the x value is doubled, the y value is tripled and the z value is multiplied by five. If we think of a cuboid, initially defined as a uniform cube with dimensions of 1, then this scaling list will give us a very different shape.



Scaling of an object is slightly more complicated. Since the object is not "solid" itself, rather just a grouping of primitives, if we were to take scaling the same way as with primitives and scale every primitive with these values, we would encounter a problem in that the primitives have changed size, but their centre points have not moved.

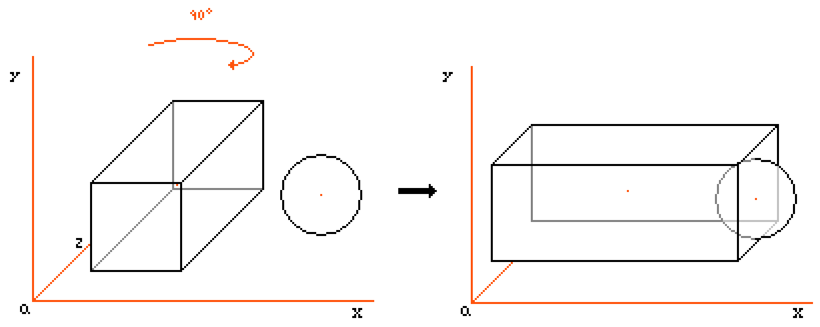


Since the relative shift in primitive centre point has not been taken into account, the larger primitive has entirely covered up the smaller one.

This problem can be solved. If we can define a cuboid with a volume that encompasses the whole object, and we know dimensions that define this shape, we can scale this cuboid and adjust the primitive positions based on their original locations within the bounding cuboid we defined before scaling.

Rotation
For primitives, rotation is fairly simple. If we need to do any kind of calculation on the primitive, we can simply translate it to the origin and "un-rotate" it, assuming we have the values of the original rotation. So long as we equally rotate any other item, e.g. a ray, we are using in conjunction with this primitive, the relative position and facing of the item is preserved, thus hopefully making the calculation easier.

Similarly to scaling, rotation is slightly harder for objects. A similar problem occurs if the object rotation is directly applied to the primitives inside.



Assuming these two primitives are part of the same object, and the object is rotated as described, then applying the rotation directly to each primitive causes these particular two to collide. To correctly rotate the object we need to take a vector going from the object centre to each primitive, and then apply rotation to this vector. The resulting vector should give the correct centre point for the primitive. Then we can also apply the rotation to the primitive. Applied to every primitive in the object, this causes the object to rotate in a "solid" manner, keeping the relative positions of all primitives.

As we described in scaling, this transformation can also change the centre points of primitives. The standard order is to scale, rotate and then translate, in modelling. So long as we are concerned with maintaining the relative positions of all primitives that compose the object, scaling and rotation can be performed without problem.

Physical Properties

We have covered the construction of models, but what about some of the smaller details? We desire to create a realistic world model, so we must have objects that have realistic properties. Storing data for these properties is easy, as we described for constructing primitives and objects. Adding a few fields to the data structures or simply creating a few new arrays for these values is quite trivial. However the values we assign to them are important. We can quite easily work out what fields would be useful by thinking about real materials. For example, consider, say, steel. We could store data on its colour, how shiny the surface is, the density, how well it conducts heat and/or electricity, etc. This works with any material we can imagine, as well as tying into some of the features of our world model. The "shininess" of a material could be a value between 0 and 1, taking one extreme for a mirror surface and the other for a matt surface. Colour is obvious. In programming terms it should be an array that stores the RGB values for the colour we want displayed. The heat and electricity conducting properties of a material are rather relevant in this project. Our heat property will dictate how quickly or slowly a material takes in and releases heat. This is very important in conjunction with a model of temperature. Similarly, conducting electricity. For a detailed model of electricity, we would probably want to model electrical cables, and thus we would need this field to tell us some value for the resistance of a cable based on the length. The density of a material is likely to only be really useful when modelling gravity. Finding the volume of a primitive combined with its density gives us the exact mass. Naturally, the mass of an object would be the total mass of every primitive

Light

Since we have said that our aim to visually model temperature, we will give small priority to modelling light. Light models are very common in games and virtual reality, so it is highly unlikely a project of this level will contribute any improvement to the field. Nonetheless, modelling light is included as one of our additional physical features, and ties in quite closely to the electricity model we have mentioned.

After building up models in our world, we obviously need to consider lighting, otherwise we won't see anything! In general there are two approaches to 3D light models; ray-tracing and

Radiosity. Radiosity is a technique that works best with surface models, and, most importantly, does not function in real-time, which is something we desire. We will instead want to employ some method of real-time ray-tracing. Ray-tracing itself is a fairly simple technique which can be implemented in real-time without great difficulty. Opposed to the Radiosity approach, ray-tracing lets us create visual effects such as reflections, partial reflection and refraction; all far more difficult with Radiosity. On the other hand, we can achieve far more realistic and visually appealing shadow effects with Radiosity.

Ray-tracing is implemented by the technique of firing rays from some point significantly relative to the user viewpoint and taking the objects these rays hit to determine visual data. The only calculations that are required are the ray direction and determining which solids it strikes. For visual data, we usually calculate which of these solids is nearest to the ray source, i.e. is "in front" of the others, and take that as what the user sees.

Reflection is handled easily by a ray-tracer simply by taking firing secondary rays from reflective surfaces. We treat the surface as a temporary viewpoint and determine what is "seen" by the surface. Then we can simply relay this back to the actual viewpoint as what is seen on the surface. One thing that should be noted is that some surfaces are more reflective than others, some are not reflective at all. A consideration we should make when dealing with the material of a surface is some value that will tell us what level of reflection to display. In this way we can also apply the value as a factor when determining the visual data upon the reflective surface. Since many surfaces may be only partially reflective, the image is reflected at only half quality. By allowing reflected images to degrade in this way, we avoid awkward situations of large numbers of reflections that may slow down the rendering process. The theoretical case of two parallel mirrors is an example; the image will be reflected an infinite number of times between them. Since the process does not stop, it will obviously cause problems. If we do not permit the idea of a "perfect mirror", this problem is avoided.

Refraction can be handled in a similar manner to reflection. Taking a refractive medium, likely to be a solid, and firing a secondary ray from the point on the surface where the initial ray hits, in a direction determined by the refractive index of the material. This process can be repeated every time a ray meets the surface of a material with refractive properties. Again, we may need a value for different materials that determines the level of refraction. If we say 1 is fully opaque, there is no refraction whatsoever, and 0 allows the ray to pass without changing its course. A value around 0.5 may give us an effect like the distortion caused by looking through a fish tank, i.e. a "displaced" of the image behind the fish tank. The closer to 1 this value becomes, the less detail will be visible through the material, the closer to 0, the less detail will be lost. As with reflection, if we consider the case of an very large number of very thin panes of glass, all in parallel and look through them side on, even though individual panes will appear to cause no refraction, a sufficient number will prevent us from clearly seeing the image on the other side of them, so we shall include the method of slightly degrading the image with each refraction calculation it goes through.

If we are considering refraction, however, there is also the matter of total internal refraction to be addressed. While, with care, this could be avoided in our simulation, we may likely to intentionally implement this feature to add more realism. Since this process follows defined mathematical rules, it should be relatively simple to model with a program.

Temperature

In order to construct a good model of temperature we need to consider the exchange of heat energy between every surface and every other. Straight away, this should sound familiar; very like the process of modelling light exchanges with Radiosity. We can see that Radiosity offers us a very good initial approximation to a temperature model, but we again run into the problem of long calculation, and the fact that Radiosity seems to work best with surface modelling. Simply calculating a temperature involves only manipulation of values we have given to primitives in the model.

Solution 1:

How do we model temperature? As mentioned, Radiosity appears to lend itself well to such a model, but we want a real-time solution. If we think of a point heat source in a room, then, assuming there are no air currents, the heat will spread out uniformly, i.e. a sphere. If we think of a normal room, i.e. a box, with a point heat source in the centre and relate real experience to this, we know that the room will not feel warm instantly. We also know that it will feel warmer nearer to the heat source than away from it. Since our sphere will represent an area or "cloud" of increased temperature, the temperature value associated with this sphere will be determined by the heat source. We know that as the source emits constant heat, the area around it that has a temperature increase will grow larger. We can say that anything outside of this sphere will not be affected by this heat source, and for anything inside the sphere we can say that the effect is inversely proportional to the distance from the centre. We consider the effect of the heat source at any point in the room to be the total heat value of all such spheres that intersect that point.

We may also consider that when an object has its temperature increased, it will release this heat over time. The length of time taken for such heat to be gained and lost is determined by the properties of the material of the object, e.g. stone takes a long time to heat up compared with something like plastic.

Even though we are not likely to use it, we can still look at Radiosity as an approach to modelling temperature since it is a legitimate method of creating a temperature model, and look at such an implementation may be beneficial in implementing a working temperature model. Radiosity works by calculating the total light energy values reaching each surface from each other surface. In this respect, we can see that it may involve a massive number of calculations even in just a small room. This works on the assumption of a fixed amount of light energy, which causes a problem when we look at real-time. Since light levels could be constantly changing in a real-time simulation, these calculations would have to be repeated almost constantly in order to remain accurate, and due to the massive amount of calculation being done, it will obviously impact on the simulation's performance. Since we are concerned with simply how much energy (light) is reaching a surface, rather than its appearance which is determined later, we can simply do the same for temperature. One consideration is that thermal energy will encounter resistance in air, which is not a problem with light. On some level, this could be considered similar to the case of a ray tracer passing a ray through translucent material, however Radiosity is not as adept as ray tracing for such techniques. Since Radiosity focuses on "patches" making up a surface, it uses the coordinates of the centre of these patches to find out where light can reach the surface from. If the coordinates for these patches are known, they can be used to calculate the distance between two surfaces for the purpose of determining whether heat from one can reach the other. This overcomes one problem, but also adds more calculations to the model. When an object is heated up it will naturally release heat until it reaches a "normal" temperature, usually room temperature. In the case of light energy, some surfaces were flagged as light emitting, and others simply reflected some existing light. The problem here is that given enough heat energy, any surface will emit heat for a time. Such a feature not a problem with Radiosity, since it assumes it is not used in real-time. This problem appears more easily approached with solid models, than with the surface models commonly used with Radiosity. Using solid models, we can use the material of objects and their volume to determine how long they take to warm up and how well they retain heat.

<u>Solution 2:</u>
An alternative temperature model could be constructed by focusing on the relative locations of objects with each other. Since we will have a list of all existing objects in the simulation, we can work through this list to pick out any with a temperature value greater than the room temperature, and/or look for a "heat source" flag that could also be incorporated. If this is found, we can calculate how much thermal energy is passed to other objects based on the distance separating the heat source and target. The use of our "heat source" flag will be to tell us whether the temperature of an object will decrease as thermal energy is released. If a heat source warms up some other object and is left alone, depending on the separation between them, the object will either increase in temperature until it matches the value of the source, remain at a stable value, or will cool down at a reduced rate. If we remove the source at some point in time, the warmed object will only cool down at some rate determined by the material type. If we do this in a ray tracing manner, we need only calculate the temperature value of objects that are hit by rays.

As a very simple model, this second approach can be applied to object models, and we can say that the object as a whole has uniform temperature. To make it a little more advanced, we can consider each primitive individually, thus allowing some parts of an object to be warmer than others. To be most accurate, we could use this approach, combined with a ray tracing type method that will allow the "front" of a primitive to appear warmer than the "back". By combining these methods we have primitives that may be very close to a heat source, thus we have the whole primitive being quite hot with the side nearest the heat source being hotter still. For primitives that are far from any heat source the temperature is likely to appear uniform, as, at a large enough distance, it won't matter which side is "facing" the heat source.

Gravity

Gravity is a physical property that is known to be somewhat troublesome to model. While we are unlikely to actually construct a working model of this, we can look at how it may be implemented in order that it could be considered for future improvement of this project. If we assume our simulation begins in a "stable" state, i.e. nothing will happen until the user provokes it, then we only need to worry about applying gravity when a reaction occurs between objects. Thinking of an example of two objects, e.g. spheres, colliding, working out the momentum of each before and after collision is fairly simple if we assume a level, flat surface, meaning we only need to determine the motion in two dimensions. If we think of a sloping surface, however, things become a little more complicated. Since gravity will be a constant factor on both spheres on such a surface, it will inevitably affect the direction and speed of their movement. Additionally, gravity is taken into account with moving objects when considering the case when they might reach the edge of a high surface and fall. Some materials, in this case, will cause objects to bounce. Due to the effect of gravity, a falling object striking the floor can be modelled just like a collision between two objects, calculating the height of such a bounce based on the force of the impact. Considering different floor materials also determines the effects of objects hitting the floor, i.e. a very hard floor and a fragile object may cause the object to break, while a bouncy object hitting a soft floor will greatly limit the bounce produced.

The mention of the effect of different materials with respect to gravity leads us on to think about how we can determine how to correctly apply gravity to objects based on their material properties. If we store a density value for all materials we can then calculate a weight value using the volume and material of primitives in an object. Having this, we can then determine how gravity is applied, i.e. whether the object is heavy or light. Additionally, we may be able to work in the application of air resistance on a falling object, which would naturally depend on its shape. A decent air current model, assuming it can be applied here, would let us model things such as gliders, and even planes, with real and accurate physics.

As mentioned, a precise model of gravity is probably too complex to be implemented in this project, however we can try to approximate the effect of gravity to a limited extend. We would expect an object being pushed beyond the edge of a surface to fall. This can be fairly easily checked for by considering it in object space and determining if the object and surface have any contact. To make it slightly more accurate, we could say that if the centre of the object is beyond the edge of the surface, i.e. it is more than half over the edge, the whole object will fall. Even if we do not model the fall precisely, we still have a partially accurate effect.

Air Currents

In some ways similar to the modelling of gravity, air currents can be modelled as a force acting in some direction but without any link to a solid model. The difference between air currents and gravity is that the force from air currents has a very limited area, whereas the model of gravity would simply be a constant world effect. Air currents could be modelled by defining a point, points or area as a starting location and then having a force act in a specified direction. The effect of collision with any object in the path of the current would depend on the force of the air current and the mass of the object. Overall, modelling this kind of feature is too difficult for a project of this level, but can be considered as one of the possible future additions.

Design Decisions

A number of features discussed in the requirements section have been left out of the design for the project we aim to create, either because of the difficulty or because we simply do not desire that feature.

Light Model

Unusually for a ray tracing based project, we are ignoring the modelling of light in favour of another form of energy model. Since we are still ray tracing, it would be possible to adjust the code sufficiently to model light without too much trouble. However since this is not much of a problem, and, as we have stated, light modelling is very common, we can consider this trivial.

Electricity Model

This feature has been omitted in favour of an attempt to model temperature which can give us a better visual end product. To fully display an electricity model, as well as having common electrical appliances being modelled, we would also have to model a fairly comprehensive light feature where the bulb is treated as another electrical appliance.

Gravity

As mentioned previously, constructing a gravity model is something too difficult for a project of this level. While we will make a weak approximation, i.e. objects falling off surfaces, etc, we can only consider a fully detailed gravity model to further extend the project into a complete simulation.

Air Currents

Similarly to gravity, air current modelling is too complex to be attempted in this project. It may be noted for future improvements, as this is also an example of a rarely occurring feature in physical models.

Implementation

Despite what is written in the requirements and design sections, not all the intended features of this project were implemented, either due to time or difficulty restrictions. This section describes the detailed design of the program that was actually written.

The program written was intended as a form of ray tracer that would produce a kind of thermal image of a solid model. Ideally we would like to have a choice of a light based or thermal image for output, but as we are considering the thermal image to be more important we will give it higher priority. The language used for building the program was C++, since this was simply the most familiar language to the programmer. The decision to create the program like a ray tracer was due to the similarity of the Radiosity method and the theoretical temperature model. A real time solution would not be possible if including some number of calculations on the scale of Radiosity, so a "static image" was created with a time variable.

Primitives

For defining primitives a method of a series of arrays was used. Separate arrays for the type, material, centre point, bounding sphere radius and temperature were created. We allowed the user to set the number of primitives in the model. Calculating the number of primitives in each object should be necessary to create the objects, and knowing the total number of objects therefore makes it trivial to calculate the number of primitives in the world. All the arrays relating to primitives were either named with "primitive" or used a name beginning "p_". It is easy to identify the purpose of every array in this fashion, as their names also serve to identify the data they hold. We are assuming that all the primitive data for each new entry will be added at the same time, such that the index of each array will match up to the same primitives.

e.g.	p_space_centre[n][3]	(where n is a primitive identifier)
	primitive_materials[n]	(where n is a material identifier)

The scaling and rotation were created as two-dimensional arrays that stored the values that the primitives were multiplied and rotated by respectively. In the case of rotation, it was assumed values were stored in the x, y, z order, i.e. rotating about the x, y and z-axis respectively. The data for the primitive bounding sphere used for ray tracing was stored as two arrays, one holding the centre point coordinates and the other for the radius of the sphere. The centre point was easily calculated by addition of the relevant primitive and object translations. The radius was calculated based upon the individual primitive dimensions and the scaling factors.

e.g.	p_rotation[n][3]	(where all n is a primitive identifier)
	p_scale[n][3]	
	p_space_radius[n]	

Additionally, related to the primitives, we also stored values for material types. The data we have considered storing is the reflective quality, the density, the colour and the "temperature variable". In the case of the colour and reflective quality, these only apply in conjunction with a light model, but we can note that these details are very easily added for future reference should we decide to expand the project with a light model. Similarly, density is a quality that will only be of use when considering the mass or weight of models, most likely when simulating gravity.

e.g.	material_temperature_property[n]	(n is a material identifier)

Although we have scale and translation applied to primitives, they are not complete functions. Scaling of a sphere is taken as uniform, using only the first scale value supplied. Scaling a cylinder or a cone allows us to adjust the height and/or the radius. A cuboid is the only primitive to use all three scaling values.

Objects
Objects were defined by a series of arrays also. Notably, the "object_primitive" array stores a list of every primitive that composes each object by keeping record of the index number relating to that primitive in the series of primitive arrays. For the objects themselves we keep arrays of the bounding space radius, centre and object translation. Rotation and scaling of objects turned out to be more complex than expected and were not implemented in time due to this.

e.g. object_primitive[m][n] (m is an object identifier, n is a short array of primitive identifiers)
o_space_centre[m][3] (m is an object identifier)
o_space_radius[m]

The centre of the object bounding sphere is easily determined by the translation on the object. The radius of the bounding sphere, however, is done in a very simplistic manner. We simply take the sum of the radii of all the primitive bounding spheres within the object, and double it, to ensure it has a large enough bounding sphere to encompass all the primitive components. This is making the assumption that all the primitives composing an object are in actual physical contact with one another, i.e. meaning they are close enough to each other that they will not pass outside the bounding sphere defined in this way. By over estimating the size of the bounding sphere, we ensure that a ray is never wrongly assumed to miss, but we also increase the number of possible hits, knowing that a majority of these will not hit any part of the object itself.

Ray Tracing
We have defined the coordinates of the source of the rays, i.e. the theoretical "eye" of the viewer, and have defined a projection plane, i.e. the theoretical screen on which the image is displayed. Naturally the "screen" is a rectangular shape and we always assume it is "in front" of the eye.

e.g. eye[3] (contains the x, y, z coordinates of the ray source)

We use this data to start ray tracing. We direct a ray to the top-left corner of the projection plane, what is, in theory, the first pixel on the screen. We can quite easily calculate the length of the current ray with some simple vector maths. We can now use this to get a ray unit vector, or direction vector. Having this direction sorted, we check for intersection with the bounding spheres of each object in the world. The test for intersection is quite simple; we assume there exists a scalar that, when applied to the ray direction vector, causes it to intersect the "object space" (bounding sphere). Substituting the unit vector into the equation to define a sphere, and then using the quadratic equation, we can find the value(s) for the scalar. So long as we have a real value, the ray has intersected the sphere. An imaginary value, or zero, for the scalar means it missed, or is a tangent to the sphere. Having confirmed that ray passes within the object space, we will then compare it against the component primitives of the object.

Similarly to the object space, we can use the quadratic method to determine whether the ray intersects the primitive space. Assuming the ray hits, we then have to determine whether it truly did reach the primitive. Each primitive type needs to be handled in a slightly different manner. A sphere is very simple, as we have already done this test. We know that the ray would hit a sphere primitive, so we can then solve the quadratic equation to give us the value for the scalar that projects the ray onto the correct coordinates on the surface of the primitive. We know a sphere is easy to test for, however the other primitives are less simple. The other primitive types require us to undo rotation first. This can quite easily be done by applying the inverse (360 degrees minus the value of rotation) of the rotations to the unit ray vector. After doing this, we need to consider the type of primitive again. A cylinder is tested for by checking for intersection between the ray and 2D circle of the cylinder. If it connects we simply check the ray falls between vertical limits. A cone is dealt with in a similar manner. Although we attempted this test with a cone, the difficulty of the operation was underestimated. We can perform the test for a cylinder using the base radius and height of the cone. We can then test for intersection with the circle, as we did with the cylinder. If successful, we need to calculate radius of the cone at any given height. This is quite a difficult task. It can be closely approximated by projecting the ray to the centre of the cone circle. We can then take the height at which the ray hits to determine the radius at that exact point. The difficulty occurs if the ray is fired from a position lower, or higher than the point it hits at the circle centre. As the ray hits passes through the surface at an angle, the height we have found for a rough intersection is not quite correct. Lastly we have a cuboid. This is simpler than cone. We can quite easily intersect the ray with the parallel planes that make up the sides of the cuboid, then we merely need to see if the intersect point is within the limits of the cuboid's dimensions. Lastly we determine which of the planes intersected is the closest to the ray source.

Temperature Model

For this project we desire a temperature model that will give us roughly accurate temperature behaviour without being overly complex. With this in mind, the second design seems the best choice. If there is no initial heat source, our world can be assumed to begin in a stable state, i.e. we can set all elements to a standard "room temperature". In this case we can still run our temperature model; however it will simply check for heat sources and, if none are found, will be required to do less work than otherwise. It is important to note that the more time we leave the heat source in existence, the greater change it will cause in the world.

Our model is going to work by first checking for primitives with some value different than the room temperature. In the case that none are found, the model needs to do nothing. If such primitives are found, we shall have the model work through them in a linear fashion. The temperature change for the current primitive is calculated based upon a number of factors; the number of heat sources, the distance they are from the primitive, the material of the primitive and the time. We will say that any heat source will give out a temperature value equal to the difference between its temperature and the room temperature. We will divide this by some factor of the distance between the primitive and the heat source, such that the temperature falls off quite rapidly as the distance increases, i.e. a non linear fashion. This can be repeated for each heat source in the world, keeping track of the total amount of heat energy being passed to the primitive. Next we take into account the primitive material. Since we are only using a simple relationship here, we will say that the relevant material quality is a number between 0 and 1, and use it as a simple multiplier for the energy being passed on. The limits, 0 and 1, refer to materials that do not heat up and heat up extremely quickly respectively. Lastly we look at time. We will, again, use a simple relationship and say that this amount of energy is passed to the primitive every second (assuming time is kept in seconds). This is, again, a simple multiplier.

We will make a temperature array in the primitive array series to store relevant data for this model. We can define some variable at the start as room temperature, e.g. say 23 degrees C. This can be either a double or integer, a double being more precise, but an integer producing more obvious visual changes. Initially we can define all primitives to have a temperature value of room temperature. If we desire any to be different, we can input that manually. Next we will assume we have a heat source. We will give it some temperature greater than room temperature, e.g. 100 degrees C. Since the model looks for any primitive with a temperature greater than room temperature, anything more than room temperature will be treated as a heat source. This may cause some needless calculations, e.g. a heat source of 24 degrees C, while room temperature is 23 degrees C may not even cause a visible change in temperature for any primitives, however the calculations will still be performed. Since this is a ray tracing method, not real time, this is not a problem, but it becomes an issue when we think of expanding the project into a real time program.

Source Code

```
//Final Year Project
// D P Jones- ma1dpj

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "stuff.h"

int max_objects; // max number of objects in the world
int p_per_o; // max number of primitives per object
int max_primitives = (max_objects*p_per_o); // max number of primitives in the world
int materials; // number of types of materials

//double material_reflective[materials]; // the reflective property of a material
//double material_density[materials]; // the density of a material
double material_temperature_property[materials]; // the response of a material to
temperature

int p_material[max_primitives]; // an array of the material type of every primitive
int p_type[max_primitives]; // an array of the shape of each primitive
// arrays for the rotation/scaling/translation operations performed for every
primitive
double p_rotate[max_primitives][3];
double p_scale[max_primitives][3];
double p_translate[max_primitives][3];
// arrays that define the bounding sphere (p_space or "primitive space") for every
primitive
double p_space_centre[max_primitives][3];
double p_space_radius[max_primitives];

// a record of every object and which primitives compose that object
int object_primitives[max_objects][p_per_o];

double o_translate[max_objects][3]; // translation operations for every object
// arrays that define the bounding sphere (o_space) for each object
double o_space_centre[max_objects][3];
double o_space_radius[max_objects];

double room_temperature; // the normal temperature for non-heat sources

int heat_source[max_primitives]; // a reference array that identifies which primitives
are heat sources
double primitive_temperature[max_primitives]; // the initial temperature of every
primitive

// values that define the area and position of the plane of projection
double p_plane_w;
double p_plane_h;
double p_plane_centre[3];
double eye[3]; // "eye" is used to describe the source of the rays

double time; // the amount of time from theoretical initialisation of the world
// defines the pixel resolution of the image being made
int image_w;
int image_h;
int main(int argc, char **argv)
{

int ray_hits;
// a value to keep track of the specific primitive hit by each ray

// These loops calculate and store the values for the radius of both primitive and
object //bounding spheres
for (int j=0;j<max_objects;j++)
{
	int i = 0;
	while (object_primitives[j][i]!=-1)
	{
		switch (p_type[i])
		{
		case 1: //sphere
		p_space_radius[i] = p_scale[i][0];
		break;
		case 2: //cone
		case 3: //cylinder
		p_space_radius[i] =
sqrt((p_scale[i][0]*p_scale[i][0])+((p_scale[i][1]/2)*(p_scale[i][1]/2)));
		break;
		case 4: //4 sided pyramid
		break;
		case 5: //5 sided pyramid
		case 6: //cube(oid)
		p_space_radius[i] =
sqrt((p_scale[i][0]*p_scale[i][0])+(p_scale[i][1]*p_scale[i][1])+(p_scale[i][2]*p_scal
e[i][2]));
		break;
		}
// The actual centre point of each primitive is calculated and stored
		p_space_centre[i][0] = p_translate[i][0]+o_translate[j][0];
		p_space_centre[i][1] = p_translate[i][1]+o_translate[j][1];
		p_space_centre[i][2] = p_translate[i][2]+o_translate[j][2];
		o_space_radius[j] = o_space_radius[j] + p_space_radius[i];
		i=i+1;
	}
	o_space_radius[j] = o_space_radius[j]*2;
	// The centre point of each object is calculated and stored
	o_space_centre[j][0] = o_translate[j][0];
	o_space_centre[j][1] = o_translate[j][1];
	o_space_centre[j][2] = o_translate[j][2];
}

// The dimensions of each pixel on the projection plane
double pixel_w = p_plane_w/(double)image_w;
double pixel_h = p_plane_h/(double)image_h;
// A structure to store the values that dictate what each pixel displays
double image[image_w][image_h];

// These coordinates identify the location of the current pixel/initial ray source.
//	The starting values identify the top left corner of the screen
double Z = p_plane_centre[2];
double X;
double Y = p_plane_centre[1]+(p_plane_h/2);

double temp_ray[3]; // A temporary record of the unit vector for the ray after //we
"undo" the translation/rotation on primitives
double temp_eye[3]; // A temporary record for the ray source after "undoing"
//rotation/translation
double scalar; // A value that projects the ray unit vector to intersect the //surface
of a primitive

for (int h=0;h<image_h;h++)
{
	X = p_plane_centre[0]-(p_plane_w/2);
	for (int w=0;w<image_w;w++)
	{
// The initial ray is actually defined here using the coordinates of the centre of the
//lens and the current pixel
		double initial_ray[3];
		initial_ray[0] = X - eye[0];
		initial_ray[1] = Y - eye[1];
		initial_ray[2] = Z - eye[2];
		double length =
sqrt((initial_ray[0]*initial_ray[0])+(initial_ray[1]*initial_ray[1])+(initial_ray[2]*i
nitial_ray[2]));
		double unit_ray[3]; // The direction vector for the current ray
		unit_ray[0] = initial_ray[0]/length;
```

```
                unit_ray[1] = initial_ray[1]/length;
                unit_ray[2] = initial_ray[2]/length;
                X = X + pixel_w;
                for (int a=0;a<max_objects;a++)
                {
                        ray_hits = -1;
// Initially we assume the ray hits nothing, so, unless overwritten, this value tells
us //the ray has missed
// Values for a, b and c variables used to solve the quadratic equation we use to help
//determine whether a ray intersects a primitive or object space
                        double o_q_a = 1.0;
                        double o_q_b = ((2*unit_ray[0])*(eye[0]-
o_space_centre[a][0]))+((2*unit_ray[1])*(eye[1]-
o_space_centre[a][1]))+((2*unit_ray[2])*(eye[2]-o_space_centre[a][2]));
                        double o_q_c = (((eye[0]-o_space_centre[a][0])*(eye[0]-
o_space_centre[a][0]))+((eye[1]-o_space_centre[a][1])*(eye[1]-
o_space_centre[a][1]))+((eye[2]-o_space_centre[a][2])*(eye[2]-o_space_centre[a][2])))-
(o_space_radius[a]*o_space_radius[a]);
                        double o_quadratic = sqrt((o_q_b*o_q_b)-(4*o_q_a*o_q_c));
// Check for intersection of the ray with the bounding sphere for the object
                        if (o_quadratic > 0.0)
                        {
                                double t1 = sqrt(((o_space_centre[a][0]-
eye[0])*(o_space_centre[a][0]-eye[0]))+((o_space_centre[a][1]-
eye[1])*(o_space_centre[a][1]-eye[1]))+((o_space_centre[a][2]-
eye[2])*(o_space_centre[a][2]-eye[2])));
// The distance from current object space to the eye (centre of projection)

                                double t2 = sqrt(((o_space_centre[a][0]-
p_plane_centre[0])*(o_space_centre[a][0]-p_plane_centre[0]))+((o_space_centre[a][1]-
p_plane_centre[1])*(o_space_centre[a][1]-p_plane_centre[1]))+((o_space_centre[a][2]-
p_plane_centre[2])*(o_space_centre[a][2]-p_plane_centre[2])));
// The distance from object space to the centre of the screen (plane of projection)
// If the object is "in front", the eye should always be a greater distance away than
the //screen

                                if (t1 > t2)
                                {
                                        for (int b=0;b<p_per_o;b++)
                                        {
                                                int current_primitive =
object_primitives[a][b];
                                                double p_q_a = 1.0;
                                                double p_q_b = ((2*unit_ray[0])*(eye[0]-
p_space_centre[current_primitive][0]))+((2*unit_ray[1])*(eye[1]-
p_space_centre[current_primitive][1]))+((2*unit_ray[2])*(eye[2]-
p_space_centre[current_primitive][2]));
                                                double p_q_c = (((eye[0]-
p_space_centre[current_primitive][0])*(eye[0]-
p_space_centre[current_primitive][0]))+((eye[1]-
p_space_centre[current_primitive][1])*(eye[1]-
p_space_centre[current_primitive][1]))+((eye[2]-
p_space_centre[current_primitive][2])*(eye[2]-p_space_centre[current_primitive][2])))-
(p_space_radius[current_primitive]*p_space_radius[current_primitive]);
                                                double p_quadratic = (p_q_b*p_q_b)-
(4*p_q_a*p_q_c);
                                                int temp_ray_hits;
                                                double temp_scalar;


                                                if (p_quadratic > 0.0)
                                                {
// These are used for the calculations involved with undoing rotation and translation
on //primitives
                                                        double
X1,X2,X3,X4,Y1,Y2,Y3,Y4,Z1,Z2,Z3,Z4,t_q_a,t_q_b,t_q_c,t_quadratic;

        switch(p_type[current_primitive])
                                                        {
                                                                case 1:
                                                                //sphere
                                                                temp_ray[0] =
(unit_ray[0]-o_translate[a][0])-p_translate[current_primitive][0];
                                                                temp_ray[1] =
(unit_ray[1]-o_translate[a][1])-p_translate[current_primitive][1];
                                                                temp_ray[2] =
(unit_ray[2]-o_translate[a][2])-p_translate[current_primitive][2];
                                                                temp_eye[0] =
(eye[0]-o_translate[a][0])-p_translate[current_primitive][0];
                                                                temp_eye[1] =
(eye[1]-o_translate[a][1])-p_translate[current_primitive][1];
```

```
                                                                      temp_eye[2] =
(eye[2]-o_translate[a][2])-p_translate[current_primitive][2];
                                                                      t_q_a = 1.0;
                                                                      t_q_b =
2*((temp_ray[0]*temp_eye[0])+(temp_ray[1]*temp_eye[1])+(temp_ray[2]*temp_eye[2]));
                                                                      t_q_c =
((temp_eye[0]*temp_eye[0])+(temp_eye[1]*temp_eye[1])+(temp_eye[2]*temp_eye[2])) -
p_space_radius[current_primitive];
                                                                      t_quadratic =
(t_q_b*t_q_b)-(4*t_q_a*t_q_c);
                                                                      if (t_quadratic >
0.0)
                                                                      {
                                                                         ray_hits =
current_primitive;
                                                                         double v1 = (-
t_q_b+sqrt(t_quadratic))/(2*t_q_a);
                                                                         double v2 = (-
t_q_b-sqrt(t_quadratic))/(2*t_q_b);
                                                                         if (v1 < v2)
                                                                         {

scalar = v1;
                                                                         }
                                                                         else scalar =
v2;
                                                                      }

                                                                      break;

                                                                      case 2:
                                                                      //cylinder
                                                                      {
                                                                      X1 = (unit_ray[0]-
o_translate[a][0])-p_translate[current_primitive][0];
                                                                      Y1 = (unit_ray[1]-
o_translate[a][1])-p_translate[current_primitive][1];
                                                                      Z1 = (unit_ray[2]-
o_translate[a][2])-p_translate[current_primitive][2];
                                                                      X2 = X1;
                                                                      Y2 = (Y1*cos(360-
p_rotate[current_primitive][0]))-(Z1*sin(360-p_rotate[current_primitive][0]));
                                                                      Z2 = (Y1*sin(360-
p_rotate[current_primitive][0]))+(Z1*cos(360-p_rotate[current_primitive][0]));;
                                                                      X3 = (Z2*sin(360-
p_rotate[current_primitive][1]))+(X2*cos(360-p_rotate[current_primitive][1]));
                                                                      Y3 = Y2;
                                                                      Z3 = (Z2*cos(360-
p_rotate[current_primitive][1]))-(X2*sin(360-p_rotate[current_primitive][1]));
                                                                      X4 = (X3*cos(360-
p_rotate[current_primitive][2]))-(Y3*sin(360-p_rotate[current_primitive][2]));
                                                                      Y4 = (X3*sin(360-
p_rotate[current_primitive][2]))+(Y3*cos(360-p_rotate[current_primitive][1]));
                                                                      Z4 = Z3;
                                                                      temp_ray[0] = X4;
                                                                      temp_ray[1] = Y4;
                                                                      temp_ray[2] = Z4;
                                                                      temp_eye[0] =
(eye[0]-o_translate[a][0])-p_translate[current_primitive][0];
                                                                      temp_eye[1] =
(eye[1]-o_translate[a][1])-p_translate[current_primitive][1];
                                                                      temp_eye[2] =
(eye[2]-o_translate[a][2])-p_translate[current_primitive][2];
                                                                      t_q_a =
(temp_ray[0]*temp_ray[0])+(temp_ray[2]*temp_ray[2]);
                                                                      t_q_b = 0.0;
                                                                      t_q_c =
p_scale[current_primitive][0]*p_scale[current_primitive][0];
                                                                      t_quadratic = -
(4*t_q_a*t_q_c);
                                                                      if (t_quadratic >
0.0)
                                                                      {

        double v1 = (-t_q_b+sqrt(t_quadratic))/(2*t_q_a);

        double v2 = (-t_q_b-sqrt(t_quadratic))/(2*t_q_b);
                                                                                     if
(v1<v2 && (temp_ray[1]*v1)<(p_scale[current_primitive][1]/2) && (temp_ray[1]*v1)>(-
p_scale[current_primitive][1]/2))
                                                                                     {
```

```
                ray_hits = current_primitive;

                scalar = v1;
                                                                                            }
                                                                                            else
if ((temp_ray[1]*v2)<(p_scale[current_primitive][1]/2) && (temp_ray[1]*v2)>(-
p_scale[current_primitive][1]/2))
                                                                                            {

                ray_hits = current_primitive;

                scalar = v2;
                                                                                            }
                                                                            }
                                                                            }
                                                                            break;

                                                                            case 3:
                                                                            //cone - INCOMPLETE
                                                                            {
                                                                            X1 = (unit_ray[0]-
o_translate[a][0])-p_translate[current_primitive][0];
                                                                            Y1 = (unit_ray[1]-
o_translate[a][1])-p_translate[current_primitive][1];
                                                                            Z1 = (unit_ray[2]-
o_translate[a][2])-p_translate[current_primitive][2];
                                                                            X2 = X1;
                                                                            Y2 = (Y1*cos(360-
p_rotate[current_primitive][0]))-(Z1*sin(360-p_rotate[current_primitive][0]));
                                                                            Z2 = (Y1*sin(360-
p_rotate[current_primitive][0]))+(Z1*cos(360-p_rotate[current_primitive][0]));;
                                                                            X3 = (Z2*sin(360-
p_rotate[current_primitive][1]))+(X2*cos(360-p_rotate[current_primitive][1]));
                                                                            Y3 = Y2;
                                                                            Z3 = (Z2*cos(360-
p_rotate[current_primitive][1]))-(X2*sin(360-p_rotate[current_primitive][1]));
                                                                            X4 = (X3*cos(360-
p_rotate[current_primitive][2]))-(Y3*sin(360-p_rotate[current_primitive][2]));
                                                                            Y4 = (X3*sin(360-
p_rotate[current_primitive][2]))+(Y3*cos(360-p_rotate[current_primitive][1]));
                                                                            Z4 = Z3;
                                                                            temp_ray[0] = X4;
                                                                            temp_ray[1] = Y4;
                                                                            temp_ray[2] = Z4;
                                                                            temp_eye[0] =
(eye[0]-o_translate[a][0])-p_translate[current_primitive][0];
                                                                            temp_eye[1] =
(eye[1]-o_translate[a][1])-p_translate[current_primitive][1];
                                                                            temp_eye[2] =
(eye[2]-o_translate[a][2])-p_translate[current_primitive][2];
                                                                            t_q_a =
(temp_ray[0]*temp_ray[0])+(temp_ray[2]*temp_ray[2]);
                                                                            t_q_b = 0.0;
                                                                            t_q_c =
p_scale[current_primitive][0]*p_scale[current_primitive][0];
                                                                            t_quadratic = -
(4*t_q_a*t_q_c);
                                                                            if (t_quadratic >
0.0)
                                                                            {

        double v1 = (-t_q_b+sqrt(t_quadratic))/(2*t_q_a);

        double v2 = (-t_q_b-sqrt(t_quadratic))/(2*t_q_b);
                                                                                            if
(v1<v2 && (temp_ray[1]*v1)<(p_scale[current_primitive][1]/2) && (temp_ray[1]*v1)>(-
p_scale[current_primitive][1]/2))
                                                                                            {

                scalar = v1;
                                                                                            }
                                                                                            else
if ((temp_ray[1]*v2)<(p_scale[current_primitive][1]/2) && (temp_ray[1]*v2)>(-
p_scale[current_primitive][1]/2))
                                                                                            {

                scalar = v2;
                                                                                            }
                                                                            }
```

```
                                                                  double variable1 =
sqrt((temp_eye[0]*temp_eye[0])+(temp_eye[2]*temp_eye[2]));
                                                                  if
((temp_ray[1]*variable1) < (p_scale[current_primitive][1]/2) &&
(temp_ray[1]*variable1) > (-p_scale[current_primitive][1]/2))
                                                                  {

       double variable2 = (temp_ray[1]*variable1)/p_scale[current_primitive][1];
                                                                               if
(((temp_ray[0]*variable1) < (p_scale[current_primitive][0]*variable2)) &&
((temp_ray[0]*variable1) > (-p_scale[current_primitive][0]*variable2)))
                                                                               {

              ray_hits = current_primitive;
                                                                               }
                                                                               else
if(((temp_ray[2]*variable1) < (p_scale[current_primitive][2]*variable2)) &&
((temp_ray[2]*variable1) > (-p_scale[current_primitive][2]*variable2)))
                                                                               {

              ray_hits = current_primitive;
                                                                               }
                                                                  }
                                                                  }
                                                                  break;

                                                                  case 4:
                                                                  //cube(oid)
                                                                  {
                                                                  X1 = (unit_ray[0]-
o_translate[a][0])-p_translate[current_primitive][0];
                                                                  Y1 = (unit_ray[1]-
o_translate[a][1])-p_translate[current_primitive][1];
                                                                  Z1 = (unit_ray[2]-
o_translate[a][2])-p_translate[current_primitive][2];
                                                                  X2 = X1;
                                                                  Y2 = (Y1*cos(360-
p_rotate[current_primitive][0]))-(Z1*sin(360-p_rotate[current_primitive][0]));
                                                                  Z2 = (Y1*sin(360-
p_rotate[current_primitive][0]))+(Z1*cos(360-p_rotate[current_primitive][0]));;
                                                                  X3 = (Z2*sin(360-
p_rotate[current_primitive][1]))+(X2*cos(360-p_rotate[current_primitive][1]));
                                                                  Y3 = Y2;
                                                                  Z3 = (Z2*cos(360-
p_rotate[current_primitive][1]))-(X2*sin(360-p_rotate[current_primitive][1]));
                                                                  X4 = (X3*cos(360-
p_rotate[current_primitive][2]))-(Y3*sin(360-p_rotate[current_primitive][2]));
                                                                  Y4 = (X3*sin(360-
p_rotate[current_primitive][2]))+(Y3*cos(360-p_rotate[current_primitive][1]));
                                                                  Z4 = Z3;
                                                                  temp_ray[0] = X4;
                                                                  temp_ray[1] = Y4;
                                                                  temp_ray[2] = Z4;
                                                                  double v1 =
sqrt(((-p_scale[current_primitive][0]/2)-temp_ray[0])*((-
p_scale[current_primitive][0]/2)-temp_ray[0]));

                                                                  double v2 =
sqrt(((p_scale[current_primitive][0]/2)-
temp_ray[0])*((p_scale[current_primitive][0]/2)-temp_ray[0]));

                                                                  double v3 =
sqrt(((-p_scale[current_primitive][1]/2)-temp_ray[1])*((-
p_scale[current_primitive][1]/2)-temp_ray[1]));

                                                                  double v4 =
sqrt(((p_scale[current_primitive][1]/2)-
temp_ray[1])*((p_scale[current_primitive][1]/2)-temp_ray[1]));

                                                                  double v5 =
sqrt(((-p_scale[current_primitive][2]/2)-temp_ray[2])*((-
p_scale[current_primitive][2]/2)-temp_ray[2]));

                                                                  double v6 =
sqrt(((p_scale[current_primitive][2]/2)-
temp_ray[2])*((p_scale[current_primitive][2]/2)-temp_ray[2]));

                                                                  if (v1<v2 && v1<v3
&& v1<v4 && v1<v5 && v1<v6)
                                                                  {
                                                                         ray_hits =
current_primitive;
                                                                         scalar = (-
p_scale[current_primitive][0]/2)/temp_ray[0];
                                                                  }
                                                                  else if (v2<v1 &&
v2<v3 && v2<v4 && v2<v5 && v2<v6)
```

```
                                                                                    {
                                                                                           ray_hits =
current_primitive;
                                                                                           scalar =
(p_scale[current_primitive][0]/2)/temp_ray[0];
                                                                                    }
                                                                                    else if (v3<v1 &&
v3<v2 && v3<v4 && v3<v5 && v3<v6)
                                                                                    {
                                                                                           ray_hits =
current_primitive;
                                                                                           scalar = (-
p_scale[current_primitive][1]/2)/temp_ray[1];
                                                                                    }
                                                                                    else if (v4<v1 &&
v4<v2 && v4<v3 && v4<v5 && v4<v6)
                                                                                    {
                                                                                           ray_hits =
current_primitive;
                                                                                           scalar =
(p_scale[current_primitive][1]/2)/temp_ray[1];
                                                                                    }
                                                                                    else if (v5<v1 &&
v5<v2 && v5<v3 && v5<v4 && v5<v6)
                                                                                    {
                                                                                           ray_hits =
current_primitive;
                                                                                           scalar = (-
p_scale[current_primitive][2]/2)/temp_ray[2];
                                                                                    }
                                                                                    else if (v6<v1 &&
v6<v2 && v6<v3 && v6<v4 && v6<v5)
                                                                                    {
                                                                                           ray_hits =
current_primitive;
                                                                                           scalar =
(p_scale[current_primitive][2]/2)/temp_ray[2];
                                                                                    }
                                                                                    }

                                                                                    break;
                                                                             }
                                                                             if (ray_hits > -1)
// If this is not the first primitive the ray has hit
                                                                             {

// Calculate the distances from the ray source to each primitive and keep the closer
one
                                                                                  double t3 =
sqrt(((unit_ray[0]*unit_ray[0])+(unit_ray[2]*unit_ray[1])+(unit_ray[2]*unit_ray[2]))*(
scalar*scalar));
                                                                                     double t4 =
sqrt(((unit_ray[0]*unit_ray[0])+(unit_ray[2]*unit_ray[1])+(unit_ray[2]*unit_ray[2]))*(
temp_scalar*temp_scalar));
                                                                                     if (t4 < t3)
                                                                                     {

       scalar = temp_scalar;

       ray_hits = temp_ray_hits;
                                                                                     }
                                                                                     else
                                                                                     {

temp_scalar = scalar;

temp_ray_hits = ray_hits;
                                                                                     }
                                                                             }
                                                                      }

                                                               }
                                                        }
                                                 }
                                          }
                                          if (ray_hits > -1)
                                          {
// Temperature values are calculated based on the separation of the primitive from
each //heat source, how long the primitive has been getting warmed and what material
the //primitive is made of
                                             double t_change = 0.0;
```

```
                         for (int r=0;r<max_primitives;r++)
                         {
                                if (primitive_temperature[r]>room_temperature)
                                {
                                              double t5 =
sqrt(((p_space_centre[r][0]-p_space_centre[ray_hits][0])*(p_space_centre[r][0]-
p_space_centre[ray_hits][0]))+((p_space_centre[r][1]-
p_space_centre[ray_hits][1])*(p_space_centre[r][1]-
p_space_centre[ray_hits][1]))+((p_space_centre[r][2]-
p_space_centre[ray_hits][2])*(p_space_centre[r][2]-p_space_centre[ray_hits][2])));
                                              if (t5 > 1.0)
                                              {
                                                            t_change =
t_change+((material_temperature_property[p_material[r]]*(primitive_temperature[r]-
room_temperature))/(t5*t5));
                                              }
                                              else t_change =
t_change+(material_temperature_property[p_material[r]]*(primitive_temperature[r]-
room_temperature));
                                }
                         }
                         t_change =
(t_change*material_temperature_property[p_material[ray_hits]])*time;
                         image[w][h] = primitive_temperature[ray_hits]+t_change;
                      }
                      else image[w][h] = 0.0;
       }
       Y = Y - pixel_h;
}


// The is the mapping of the image data into a .tga file format.
// The name of the output should be specified at runtime by appending the filename and
//extension to program call
FILE *f = fopen(argv[1],"w");

putc(0,f);
putc(0,f);
putc(2,f);
putc(0,f); putc(0,f);
putc(0,f); putc(0,f);
putc(0,f);
putc(0,f); putc(0,f);
putc(0,f); putc(0,f);
putc((image_w & 0x00FF), f);
putc((image_w & 0xFF00) >> 8,f);
putc((image_h & 0x00FF), f);
putc((image_h & 0xFF00) >> 8,f);
putc(24,f);
putc(0,f);

// These loops take every item from the 'image' array and put them into the tga
for (int m=0;m<image_h;m++)
{
       for (int n=0;n<image_w;n++)
       {
              // Because the array only stores integers, and because there are 16
rays, it will only have
              //     values between -16 and 16.  Some simple maths allows this to
give a corresponding value
              //     between 0 and 255.  The image is stored as a black and white
picture, no colours
              double t = (image[n][m]);
              // We are going for an infrared camera look, so we are only going to
use shades of red, however the common multicoloured thermal image could be fairly
easily achieved too
              fputc((int)t, f);
              fputc(0, f);
              fputc(0, f);
       }
}
fclose(f);
}
```

Testing

A lack of time meant that exhaustive testing could not be carried out. Frequent syntax testing was applied however, and no errors were found. Mathematical equations within the program were checked, but the possibility of programmer error has not been eliminated, again due to lack of complete testing.

Test Plan

Are models constructed?

Does the primitive array(s) contain primitive data?

Yes. Primitive data can be stored properly in the relevant arrays.

Does the object array(s) contain object data?

Yes. Object data can also be stored properly.

Do the world variables contain world data?

Yes, we can assign values of approximately a realistic scale in these variables.

Are models correct?

Are primitives stored correctly?

Yes, the primitive data can be stored in the correct order.

Are objects stored correctly?

Yes, object data can also be stored correctly ordered.

Do objects and primitives correspond?

Yes, a list of references to the primitives in each object does exist.

Is world displayed?

Do primitives appear?

No. Not all primitives can be ray traced, only the cylinder, cube and sphere can be displayed.

Do objects appear?

Only objects that do not feature a cone primitive can be displayed.

Is display correct?

Are primitives oriented correctly?

Rotation, scaling and translation have all been applied to the primitives.

Are objects constructed and oriented correctly?

Objects can be translated. Orientation depends on the individual orientation of primitives.

Does temperature model work?

Not tested. No syntax or mathematical errors found.

Does temperature model work correctly?

Not tested.

Do material properties work?

Can primitives be assigned different materials?

Yes, a list of primitive material types is stored corresponding to defined materials.

Do materials have different properties?

Yes, any number of materials with different properties can be created.

Do different properties cause different behaviour?

Requires temperature testing to answer, however temperature model depends on material some properties

Conclusion

Evaluation

In the end the solution that was implemented was somewhere between a ray tracer and a method similar to Radiosity. Although what was intended at the initial stages was something more advanced, the difficulty of reaching the current stage was quite highly underestimated. Initially it was intended to include more primitive types in the form of 4 and 5 sided pyramids, and a multiple sided prism like shape. Given the severe difficulty of ray tracing just a cone, and considering that pyramids would likely function as similar, but more complex shapes in this respect, we can assume implementing these additional primitives would certainly not be trivial. While the program is unfinished, due to lack of time, it has reached a stage with several implemented features. The lack of exhaustive testing was a problem, however since the program compiled cleanly and without error, any difficulties encountered while running the program would be down to mathematical errors that can be quite easily fixed. Overall, while the program produced by the project was not of exceptionally high level, the research involved in trying to implement such a model was highly enlightening. The more time spent thinking on how to build up a fully accurate temperature model revealed just how vast and complex, but interesting such a model would inevitably be.

Problems Encountered

As mentioned before, the cone primitive proved to be exceptionally difficult to handle. Also, problems were encountered initially in trying to ray trace the cylinder primitive as well. The cylinder was fixed in time, the cone wasn't. While researching the temperature model on paper it became clear that to make it fully accurate would require calculations very similar to Radiosity, but with one major difference. The maximum energy level in the Radiosity method is fixed, however temperature would require a constantly changing maximum energy level, since objects would naturally cool down if not continually supplied with heat. So long as the heat sources function, the amount of thermal energy being given out is constant, but if a heat source is removed, the energy level will slowly change from that point. The Radiosity technique could conceivably work here, so long as we don’t consider time. The consideration of time leads onto the idea of changing the “state” of the world, e.g. removing or adding heat sources at certain times. These kinds of actions change the maximum energy value that Radiosity relies on to perform calculations.

Ultimately the temperature model was not completed, although it reached a functioning stage. One aspect that we neglected to include was capping the primitive temperature value at the temperature of the hottest source. This is not a large problem, it would simply mean primitives can heat up beyond the sources that are acting on them. Physically this is wrong, but from a programming point of view it is only a minor error.

Future Improvement

Cold source modelling
We have considered primitives that emit heat in some detail, but have not mentioned those that are colder than room temperature. We could simply mirror our heat source calculations to deal with "cold" sources in the same way, but our desire is to simplify this model as much as possible. At this stage we will simply say that a primitive below room temperature will gradually change to reach room temperature, but will not affect anything else in the world. This allows cold things to show up in our thermal vision, while trying to keep our model simple, however it also takes away some of the accuracy of our model. It is entirely possible to model temperature in such a way that all primitives/objects strive to reach and, stay at, the normal room temperature, regardless of their initial temperatures.

Improved temperature modelling
As mentioned above modelling cold as well as heat sources does improve our temperature model vastly, however it is really only performing almost the same calculations as currently, simply trying to raise a temperature value up to room temperature, rather than just go above it. It seems clear that the best method to accurately model temperature is to make use of Radiosity based methods. From knowledge of normal Radiosity, this will involve a large number of complicated calculations. We cannot rely entirely on normal Radiosity as a base, since light energy and thermal energy behave differently. The only way to be sure would to work out the mathematics involved beforehand. This would undoubtedly be a long and difficult process, but would most likely result in a very pleasing physical model.

Light
As it stands our program already does most of the work required to ray trace whatever solid world we supply it. If we were to define some light sources and calculate the normals at the points where the rays hit the surfaces of the primitives we could already create a light model. Since ray tracing is being used it would relatively simple to build up and include reflection, refraction and shadow models.