A 3D Modelling and Visualisation System for Clay Pigeon Shooting

Final report

Asa Denton
June 2003

Supervisor: Tony Field
Second Marker: William Knottenbelt
Acknowledgements

I would like to thank Tony Field for being a good supervisor and always finding time for me. Also I would like to thank Iain Buchanan, Russell Wood, Simon Coulson and my father for endless discussions on splines. Finally I would like to thank Kathy Langford for being a helpful and understanding girlfriend.
Abstract

This report details the design and implementation of a 3D clay pigeon shooting environment. The motivation for this project is as a diagnostic visualisation system to aid learning and improving shotgun shooting. Major features of this simulation are accurate ballistics modelling, a ‘replay’ mode, a 3D world and aiming aids.

A thorough analysis of the current methods and relevant background are presented and implementation and evaluation of the chosen methods are discussed. The ballistics are taken from a previously published model and an innovative method using splines for modelling pellet motion and collision detection is built upon this. The ‘ROAM’ level of detail algorithm is used to produce a realistic simplified terrain. Other rendering algorithms and simulation structure are also detailed.

The report concludes by summarising the major achievements and suggesting possible further work to this project.
Contents

1 Introduction 1
  1.1 Objectives 2
  1.2 Report Structure 2

2 Background 4
  2.1 Clay Pigeon Shooting 4
  2.2 Virtual Reality 6
  2.3 Existing VR Clay Pigeon Simulators 6
  2.4 Ballistics model 8
  2.5 Windowing 8
  2.6 Terrain 9
    2.6.1 Modelling 9
    2.6.2 Terrain Level Of Detail 11
    2.6.3 Texturing 13
  2.7 Features 13
    2.7.1 Bill Boarding 14
    2.7.2 Coarse Polygon Approximation 14
    2.7.3 Procedural Trees 14
    2.7.4 Vertex Shaders 15
4.1.4 Training Mode ................................................. 43
4.1.5 Tracking Mode .............................................. 44
4.2 Clock ............................................................... 45
4.3 Recording and Replay ........................................ 45
4.4 Sound .............................................................. 46
4.5 Describing a Range ............................................. 46
  4.5.1 Trap Description ............................................. 48
  4.5.2 Terrain Description ...................................... 48
  4.5.3 Features Description .................................... 49
  4.5.4 Sky Description .......................................... 51
  4.5.5 Constructing a Range ................................... 51

5 Rendering Algorithms ........................................... 53
  5.1 Terrain ......................................................... 53
    5.1.1 ROAM ................................................... 54
    5.1.2 Textures ............................................... 57
  5.2 Features ....................................................... 57
  5.3 Sky ............................................................. 60
  5.4 Clay ............................................................ 60
  5.5 Pellets ......................................................... 61

6 Summary and Evaluation ......................................... 62
  6.1 Successes ..................................................... 63
  6.2 Further Work ............................................... 64
Chapter 1

Introduction

Clay pigeon shooting is a fast-growing and entertaining sport, attracting many new participants each year. The idea is simple: a fast moving clay disc (the "clay") is projected from a "trap" and the shooter attempts to hit the clay in flight using pellets fired from a shotgun. There are many clay shooting 'formats' which vary in the layout of the shooting range, but the basic principles are the same for each.

Because the target is in motion when the gun is fired the shooter must aim in front of the clay to allow for the time taken for the pellets to reach it. Gauging the lead of the shotgun and maintaining a smooth "swing" of the gun barrel can be difficult, especially for those inexperienced in the sport. Even for an experienced instructor standing behind the shooter, it is often hard to determine how a target has been missed as it is not generally possible to see the pellets in motion - the pellet 'cloud' can sometimes be spotted as a faint shadow in favourable lighting conditions, but this does not give an impression of the cloud’s three-dimensional characteristics. It is also difficult to paint a mental picture to a beginner of where they should be aiming when they fire the gun, as this varies with the distance to the clay and the clay’s speed and trajectory at a given time.

The aim of this project is to produce an interactive virtual clay pigeon shooting system to help a user improve their performance through interactive computer models of the pellet cloud and target trajectories. A key component of this work is therefore a powerful visualisation system which incorporates extensive diagnostic ‘playback’ facilities and, additionally, the ability to view required target lead either in real-time or retrospectively.

This project is part of a larger project whose goal is to design a complete virtual reality clay pigeon shooting simulation. The idea is for the user to fire at computer generated targets with a real shotgun, which will be tracked using a camera based visual tracking system. The project described here concerns itself with the rendering of a 3D world, the modelling of the pellet ballistics and
the design and implementation of the diagnostics feedback system. This system can be operated independently of the tracking system with input being from a mouse and keyboard. The ‘sister’ project [17] is concerned with the construction of the gun tracking system.

1.1 Objectives

The original objectives of the project can be summarised as follows:

- To implement an efficient and accurate pellet cloud ballistics model based on the work described in [22] and prototyped in [16]
- To build a realistic model of the trajectory of a clay pigeon.
- To provide detailed feedback to the user on both pellet cloud and target trajectories to help diagnose misses and/or evaluate the quality of a hit.
- To provide a visualisation in real-time of the lead required to hit the target (“mental picture”).
- To render an artificial 3D landscape constituting the virtual shooting range, including terrain, sky and other features.
- To provide a stereographic visualisation of the virtual range to allow the user to interact with the system in three dimensions.

The first five objectives have been met in full and are discussed in detail in the succeeding chapters; the final objective was not realised because of resource and time constraints. This is revisited in the section on future work.

1.2 Report Structure

The report is structured around the major sections of the project. The background introduces the relevant topics to this project. It details considered methods for implementing each part of the code and their advantages and disadvantages.

The ballistics model section discusses the existing implementation of the pellet modelling software and the problems surrounding it. As a result of this the model was re-implemented. The methods used to do this and also the collision detection are discussed as part of this. Several problems resulted from this first implementation and as a result a second innovative solution was found. Together with this are methods to predict clay motion, calculate the closest passing point of the pellet cloud and new collision detection methods.
The simulation framework discusses the distinguishing features of the program. This includes the user interface and how the important aspects are implemented. The rendering algorithms section describes the algorithms used to render the scene displayed in the simulation and displays their results.

Finally conclusions are drawn as to whether the project was a success. The lessons learned from implementing this project are given, possible improvements and future work.
Chapter 2

Background

2.1 Clay Pigeon Shooting

“Clay pigeon shooting is the art of shooting at special flying targets, known as clay pigeons or clay targets, with a shotgun.”[2]

Clay pigeon shooting was developed as a substitute for live game shooting. Since then it has developed its own identity as a sport. It is one of the fastest growing sports in the UK with membership increasing by 180% in the last 10 years [1]. As a hobby it appeals to a wide age range of people of both sexes.

Shotguns, as defined by law, are smooth barrelled guns with barrel of length 24 inches or greater and a diameter not exceeding 2 inches. Clay pigeon shooting is more restrictive and barrels must be 12 bore or less (equivalent to 0.729 inches) diameter and between 26 and 32 inches in length [1].

A shotgun fires a cartridge containing propellent, a wad and pellets. When the cartridge is fired the propellent expands forcing the wad and hence pellets out of the barrel. The pellets leave the barrel at approximately 400 ms$^{-1}$ and spread as their distance from the gun increases, forming a cloud. Figure 2.1 shows a 2D plate pattern representing the locations the pellets hit after 40 yards. This gives the shooter more leeway in their aim at a moving target. The cartridge can vary greatly; shot size and shot load weight are just two varying factors.

Clay pigeons or clays are usually in the shape of an inverted saucer and made from a mixture of pitch and chalk. They are designed to withstand being thrown from traps at very high speeds but also to be easily broken by just a few lead pellets shot from a gun. The shooter must fire in front of the clay to allow for the flight time of the pellet cloud to the clay, this is referred to as “lead”.

Clays are usually black but can be many other colours so they can be seen
against varying backgrounds. They are made to very exact standards depending on their type. The most common is ‘standard’ (figure 2.2) and must weigh 105g, be 110mm diameter and 25-26mm in height.

Traps launch the clays at distances of up to 100 metres. These vary from the simple hand load and release spring mechanisms to the fully automatic activated by voice or remote control. An automated trap is illustrated in figure 2.2. Target launch speeds and trajectories can be easily modified and varied to suit shooting requirements.

Clay pigeon shooting is a collective term. The sport itself can be divided into numerous categories, each with their own layout, rules and equipment. There are three major sections within the sport comprising of smaller sub sections\(^1\). These vary in the trajectory of the target and the position of the shooter relative to the clays.

- Trap : Down the Line, Double Rise, Automatic Ball Trap, Five Trap, Double Trap, Olympic Trap and ZZ
- Sporting : English Sporting, F.I.T.A.S.C. Sporting, Compak or Sportrap and specially designed for television - White Gold Sporting

\(^1\)For a detailed description of the sporting categories see [3]
• Skeet: English Skeet, Olympic Skeet and Skeet doubles

The most popular of these is English Sporting. Trap and Skeet are highly regulated using only standard clays on set trajectories. English Sporting is a much broader discipline as it aims to simulate live quarry shooting. A variety of clays are used and fired in a range of trajectories.

2.2 Virtual Reality

“Computer simulations that use 3D graphics and devices such as the data glove to allow the user to interact with the simulation” [7]

Virtual reality involves projecting a world in such a way that it appears in 3D to the user. Communication with this environment should be similar to that expected in the real world. The input to this software will be in the form of the orientation of a shotgun and a trigger pull event. How this is retrieved is part of the sister project [17]. To make this project stand alone, the user input will be in the form of a keyboard and mouse.

2.3 Existing VR Clay Pigeon Simulators

There are several virtual reality (VR) clay pigeon simulators mainly for use in arcades and shooting ranges. As there are so few simulators they do not attract reviews so there is only manufacturer’s information available for research. Three such systems are commercially available.

• Winchester Total Recoil (WTR) [13] (figure 2.3) is a virtual reality system that uses a Winchester 101 shotgun. There are many software variations that can be run on the simulator on this but “Trap Master” is the main program to simulate clay pigeon shooting. Documentation on this product is very limited and appears mainly to be advertising. Its technical specifications are low relative to today’s capabilities, as it was released in 1995, using dual 100Mhz Motorola chips with some dedicated graphics hardware.

• DryFire [5] (figure 2.4) is a home clay pigeon simulation product. It can be simply plugged into a Windows PC and then used in the home or office. The user’s own gun can be employed to fire at a dot projected on a wall. The ballistics model provided with this package is apparently very realistic allowing for variable shot size, number of pellets and barrel choke size.

• ShotPro 2000 [11] (figure 2.5) employs lasers to detect the angle at which the gun is fired. It uses several projectors to display a shooting range scene in 2D to enhance the shooting experience. Again, with this simulator, the user may use their own gun loaded with a special laser cartridge.
Figure 2.3: Winchester Total Recoil [13]

Figure 2.4: DryFire [5]

Figure 2.5: ShotPro 2000 [11]
2.4 Ballistics model

Ballistics of pellet flight has previously been measured to assist in the development and evaluation of non-toxic shot [21] [22]. As part of this study an extremely accurate statistical model of pellet distribution and flight has been developed for several different types of material, size and propellent. This is based on a representation of each pellet as a sphere travelling at either transonic or subsonic speeds.

To model the pellet cloud, parameters for every pellet within this cloud are generated. The pellet direction is determined by two angles $\theta$ and $\phi$ (figure 3.1). $\theta$ is the angle between the direction of the gun barrel and the direction of the pellet flight and is normally distributed around zero with a constant standard deviation for a specific gun. $\phi$ is the angle of rotation from the vertical of the gun to the direction of the offset from the gun aim. This second angle is uniformly distributed.

Each pellet has a diameter generated from a normal distribution based on the cartridge type used. This diameter determines the deceleration constant of the specific pellet. The deceleration constant of the leading (largest) and trailing (smallest) pellets are specified by the model and values between these two are linearly interpolated using the diameter.

The time taken to travel a given distance is given by differential equations for modelling sphere motion. Equation 2.1 models the transonic (speeds between mach 0.5 and mach 1) and equation 2.2 the subsonic (speeds below mach 0.5) flight [21] [22].

\[
\frac{dv}{dt} = -kv^3 \quad (2.1)
\]

\[
\frac{dv}{dt} = -kv^2 \quad (2.2)
\]

where $v$ is velocity, $t$ is time and $k$ is the deceleration constant.

This model was implemented as a previous project [16].

2.5 Windowing

A key aspect of this project will be efficiency, whilst trying to achieve realistic graphics. OpenGL or DirectX will be used to render the 3D scene taking full advantage of the speed of graphics hardware. As there is the potential to use a cross-platform graphics library such as OpenGL, a cross platform windowing library could also be used. This type of windowing system will provide much greater freedom in the future.

The sister project is likely to be fixed to a Windows platform but due to computation requirements the software for each part may run on separate computers and communicate across a network. Also due to the general nature of
these libraries they tend to have more basic interfaces than if communicating directly with the operating system. Two common free cross platform libraries are wxWindows [14] and SDL [12].

wxWindows is a much larger library and contains functionality to draw widgets such as buttons, scroll bars and other common components. This project will not require these as the input from the user will be in a very non-standard format (from a shotgun). Should they be desired but prove difficult to implement, then there are widget libraries for drawing components in OpenGL or DirectX, such as “The Fast Light Toolkit” [6].

SDL is aimed directly at games software. It has very limited functionality but is small and simple to use. SDL is also free for all platforms and has previously been used in the writing of several widely distributed games making it a reliable option. SDL is very lightweight and fast, thus leaving as much processing power for games as possible. It also provides functionality for other devices used in gaming such as sound cards and joysticks. To use this with DirectX would be largely redundant but would complement the use of OpenGL.

2.6 Terrain

A simulation of clay pigeon shooting will require a terrain constructed of polygons so it renders quickly. The most basic terrain would be flat and represented as a single polygon. It would require no designing and would render extremely quickly. With a flat polygon, there is no need to calculate heights of given points on which to draw trees for example, and the person would always be at the same height. However, there would be an issue of how to deal with the edges of this polygon. Programming this would be very straightforward but it would be boring for the user.

2.6.1 Modelling

To add flexibility and interest to this project, it is preferable allow for more interesting landscapes than a simple plane. These could be hard coded but that would limit the expansion of the software. A published interface could be distributed but any designer would require programming skills. A map designed using this method would require significant development and testing time.

Another option would be to load the terrain details from a file. This would enable terrains to be loaded at run time and designed outside of the code itself. This allows for much more expansion capability and flexibility. Using a standard file format would make it easier for individuals with experience in design programs that can save to this. There are many options and languages for this standardisation.
3D Modelling Languages

One possibility is to save the terrain in 3D modelling languages such as the Virtual Reality Modelling Language (VRML) or Visual Studio 3D Max’s file format. There are many converters between languages freely available on the Internet and if the file format chosen is standard other file formats could be transferred. Two of the most common and well documented are those given above.

VRML was mainly designed as a 3D equivalent to HTML. It uses similar constructs except they describe models. These constructs are largely based around drawing common objects such as spheres and cubes. These shapes do not occur in landscapes so the efficiency and simplicity of this language is lost in this case.

Visual Studio 3D file formats have not been officially released but many attempts have been made to reverse engineer them. For this reason the documentation is incomplete and parsers are buggy. To write a parser, example files would be required and with no available means of constructing these this could prove extremely difficult. A further disadvantage is the prohibitive cost of a licence for Visual Studio 3D Max.

Writing directly without the aid of a tool in either of these formats tends to be difficult. The languages are bulky and not representative of the scene the design will form. The text based format also means that the file will take a lot of space for storage and parsing time will be high. There are tools for designing 3D environments and for saving the result into one of these languages but these are expensive and difficult to learn.

Another option would be to use splines to model the landscape. Again these would make it relatively difficult for a designer to manually specify the landscape. This would limit those who could use them and require specification of a custom file format. A tool for designing landscapes could be written for any of these formats but it would prove time consuming.

Height Maps

Height maps are another common way of specifying landscapes. The colour intensity of a bitmap is used to provide the height at regular points. For example white (255) might represent a high point and black (0) a low point in the terrain. This gives an evenly spaced mesh of height values, which is a little redundant since only terrain close to the viewpoint requires this high level of sampling. In the future, should the ability to move around the scene be added, then this level of sampling would be beneficial.

Height maps are still much more efficient storage mechanisms than text file formats such as VRML. It is also very easy to design new maps with this method, requiring only a simple image editor of which there are several free...
versions available. There are several options for the file format choice as the requirement is simply a bitmap and once chosen, converting to this another can be done using a standard paint package.

**Generation Methods**

The final method is to generate the terrain every time it is run. Using noise generation algorithms such as Perlin [9], this is simple. The noise algorithms usually generate something similar to that of a height map, these methods could easily be combined.

With this method great care must be taken to generate a realistic landscape. Only limited control could be provided over the form of the landscape and a user could not reuse a particularly liked landscape. This could provide problems if a user wishes to imitate a specific discipline and range, and the generated landscape is not suitable.

**2.6.2 Terrain Level Of Detail**

In all formats it is very likely that there will be a degree of redundancy. A large flat terrain may still be drawn with a large number of polygons and slow the process rendering unnecessarily. Also distant terrain requires less polygons to look effective, as described above.

This leads to the need for a polygon reduction algorithm of which there are many variations [19]. The majority of these are based on calculating the error created in the terrain by removing certain points. If the error is small then the point can be safely removed or merged with another point.

These polygon reduction or Level Of Detail (LOD) algorithms are complex and will require much consideration. There are many spurious effects that can occur if great care is not taken. It may not be necessary to use an algorithm for some of the described formats if the developers of new maps are careful in their designs. Height maps with their regular sampling will definitely need a LOD algorithm. The regular sampling of the bitmaps does however, make them well suited to such algorithms and several have been developed (figure 2.6).

The simplest LOD algorithm would be a sub sampling method although this may result in key features being missed. Should the ability to move around be added at a later stage then some irregular effects may be seen such as large features suddenly appearing.

Two of the most capable and well documented LOD algorithms are Real-time Optimally Adapting Meshes (ROAM) [18] and View-Dependent Progressive Meshes (VDPM) [23] [24]. Both algorithms are based on traversing a tree of triangles constructed from the grid of heights. They both perform tests on the vertices and remove or merge those that have little visual impact. Both
Both techniques store a series of transformations, each describing how to mutate to the next level of detail. These must be saved for both directions, increasing and decreasing levels. For ROAM these transformations are splitting and joining triangles in a specified format. VPDM splits and joins vertices, constructing and decimating triangles as necessary.

VDPM produces a more optimum triangle mesh but ROAM is faster so is generally preferred for real-time uses. A comparison has been given in ‘ROAM vs. VDPM’ [10] where ROAM is considered the better of the two algorithms. The mesh produced by VDPM may also contain slivers. These are very narrow triangles that can cause holes in the rendering because they are discarded by the graphics card as too small to display. ROAM does not suffer from this problem.

Modern graphics cards have many hardware accelerations such as caching to optimise drawing. The papers on both ROAM and VDPM also present extensions on how to adapt the algorithms to take advantages of these.

Some consideration should also be given to the fastest method for drawing triangles. There are several means of drawing these and reducing the number of calculations required, the data sent and the data stored. Usually the fastest method uses triangle strips [8] in figure 2.7.

Creating a run time LOD algorithm to incorporate triangle stripping is very difficult. A utility called nvTriStrip [26] is available from NVIDIA that performs
stripping on any set of triangles. Some testing will be required to establish what
the speed increase resulting from utility is although it definitely will not be a
run-time tool.

2.6.3 Texturing

The simplest approach to rendering the landscape would be to colour the entire
landscape green however, this would look quite dull. Lighting would add a little
to the landscape to give it some depth.

An alternative to plain colouring is to texture the landscape. A texture is a
1, 2 or 3D image containing a vector or scalar value at each of its points. In the
case of DirectX and OpenGL each vertex is given a texture co-ordinate. The
value at this point in the texture is assigned to that vertex. The areas between
vertices are interpolated from the texture map.

A landscape is obviously very large and ideally only small textures are used,
interpolated and repeated to reduce the memory usage. Close to the viewpoint
a detailed texture gives more appealing terrain. High detail textures can be
repeated across the terrain but this always reveals patterns in the small texture
so the landscape appears to have stripes. An alternative is to have a low detail
texture stretched across the entire landscape, hence removing the detail close
to the user.

An extension to OpenGL and DirectX allows several textures to be combined
and mapped to the same area. This, combined with the possibilities above, gives
high detail close to the user and no visible repetition across the landscape. The
texture map stretching across the whole terrain hides the patterns in the detail
texture.

2.7 Features

Features are objects such as tufts of grass, trees and rocks. These will be
placed on the terrain to add to the character of the scene. The potential to
develop these is limitless. The clay object could also use similar techniques to
the features with some extra functionality added to model motion and collision
detection.

Techniques to draw features range from very simple and fast to much more
complex and slower. Simple methods do not look as effective close to the view-
point but it is advantageous to use them at a distance because of their speed.
This is even more significant in a 3D projection as the close features will be
viewed from distinct angles. Initially all features will be drawn using the sim-
plest techniques then if time permits the closer ones will use more advanced
models.
2.7.1 Bill Boarding

Bill boarding [15] is one of the most simple methods and gives good results at a distance. Each object is drawn as a single polygon that always faces the viewpoint with a partially transparent texture of the object applied to it. This method is satisfactory for distant objects as it is fast to render and it will be very difficult to tell they are flat.

2.7.2 Coarse Polygon Approximation

A more complex method draws objects as a number of different polygons in the general shape of the object2 (figure 2.8. Textures can then be added to give a more realistic appearance. One of the modelling languages file formats discussed in section 2.6.1 can be used to specify object shapes and textures.

In 2D this method gives a good representation of objects that can be closely approximated by a few polygons such as rocks. Unfortunately close objects that are not planar such as trees and grass do not look realistic. This method would still be preferable to the bill board method because the trees look different from different angles giving a better 3D appearance. This is the usual method for drawing near trees in current games as it is very fast.

2.7.3 Procedural Trees

Using more polygons, trees can be drawn after procedural generation (figure 2.9. These are much more realistic but require many parameters and a complex algorithm for generation. On the NVIDIA web site [25] there is example source code to deal with the generation of these trees.

With this method only the textures and parameters need saving so it is a very efficient way to store many tree types. Random elements can also be added

2This will be called coarse polygon approximation
to vary the identical (same species) trees slightly as in nature. Due to the large number of polygons used to render these trees, they are typically only used in the foreground.

### 2.7.4 Vertex Shaders

Another highly effective technique is vertex shaders. A procedure is written to control the vertices that construct an object. This is uploaded to the graphics card and can only be done on the most modern cards, such as the NVIDIA GeForce 4. The code is run every time the image is redrawn and will perform some calculation on the vertices. This gives very simple animation to vertices enabling very realistic movement in the scene such as grass and trees blowing in the wind. It is all performed in hardware so is very fast.

### 2.8 Sky

Rendering a sky further increases the realism of the scene. Two options for the sky are a dome and a curved plane to ensure that the sky is visible in every direction. Both of these options have been previously implemented in software [28]. The dome can be seen in figure 2.10 and the plane in figure 2.11. The dome has the problem of slithers occurring at the very top, which the graphics card may render incorrectly.

### 2.9 Lighting

Lighting is very important to the construction of a realistic scene. Modern graphics cards handle this requiring only the light location, the normals of the polygons and the material reflective qualities. Lighting can then be enabled and the graphics card will perform all the necessary calculations.
Testing will need to be performed to find the optimum material properties but this will require testing and viewing. Code will also need to be written to generate the normals.

2.10 Shotgun

The shotgun will not require modelling graphically as the user will be able to see it through the polarising goggles. They will use the gun to shoot at the virtual clay and the sister project will send the co-ordinates and the moment at which the trigger is pulled. The software running these calculations will be quite intensive, as they will occur every frame the camera records. It will therefore probably run on a separate machine and the co-ordinates passed across a network socket either using request-reply or, more likely, UDP.

The location of the shooters will have to be passed from the hardware as often as they are available; they are required to calculate the viewpoint. This simulation will have to take into account the most up-to-date co-ordinates every time it redraws the screen so that accurate projection is achieved.

There are two possibilities for the information sent regarding gun co-ordinates. The first is to have the current co-ordinates of the gun passed across the link. This project will then have to decide what to do with each of those points and maintain the current gun location and possible velocity. The second option is for the sister project to maintain these details. It is likely that the sister project will be carrying out velocity calculations anyway to predict the location of the
next point. This code would then be informed of all the relevant information at the time when the trigger was pulled and could then calculate a hit or a miss.

2.11 Sound

Sound enhances the realism of the virtual world. As this simulation will be mainly based around watching and shooting at a visual target, sound will not be necessary but it would be a nice touch to add at the end. Adding some simple effects would not be too difficult using some freely available libraries and downloaded sound files such as birdsong or a shotgun firing. Both SDL and DirectX provide support for sound card interaction.
Chapter 3

Ballistics Model

As a clay pigeon simulation and training tool it is very important that the ballistics are accurately modelled. Both the clay and the pellet flight must be statistically correct and the collision detection between the two exact. The pellet flight will be calculated using the model given by [21] and [22].

The clay flight model has not previously been specified. The complexity of modelling the clay could vary greatly but it should achieve the smooth motion that any experienced clay shooter would expect. As this project does not cover the physical modelling of the motion of a clay, although it should be simple to change the modelling code to this in the future.

The number of pellets in a cloud can be anything up to 1100 but is typically around 300 and calculating the location and intersections of the pellets will be required every frame. Evaluating this for high numbers of pellets with one or more targets is computationally expensive so efficiency will be paramount to ensure real-time calculations.

As part of the specification of this project, the existing implementation [16] will be considered for modelling the pellet cloud. Due to concerns over efficiency it was first rewritten with the structure remaining largely the same. Experiments with this improved implementation highlighted serious problems and as a result of this a new method of modelling the pellet flight was found.

3.1 First Attempt

The existing implementation of the model described in [21] and [22] is detailed in [16]. This exploits the analytical solution to equations 2.2 and 2.1, which give the flight time of a pellet as a function of distance travelled, equations 3.1

---

1 This will be referred to as ‘the model’.
2 This will be referred to as the ‘existing implementation’.
and 3.2.

\[ t = \frac{1}{k v_0} (e^{k R} - 1) \] (3.1)

\[ t = \frac{1}{v_0} R + \frac{k}{2} R^2 \] (3.2)

where \( k \) is the declaration constant, \( v_0 \) is the initial velocity, \( R \) is the distance travelled by the pellet and \( t \) is the time taken.

To implement the model equations 3.1 and 3.2 had been rearranged to give equations 3.3 and 3.4. These give the distance travelled by the pellet from the subsonic and transonic flight time respectively.

\[ R = \frac{\ln (t v_0 k + 1)}{k} \] (3.3)

\[ R = \frac{-\frac{1}{v_0} + \sqrt{\frac{1}{v_0^2} + 2 k t}}{k} \] (3.4)

where \( R \) is the distance travelled by the pellet, \( t \) is time, \( v_0 \) is the initial velocity and \( k \) is the declaration constant.

To determine which equation to use the point at which the pellet passes from transonic to subsonic velocity is needed. This is also derived from differential equations modelling sphere motion [16].

\[ v = \frac{v_0}{1 + k v_0 R} \] (3.5)

where \( v_0 \) is the initial velocity from the gun barrel, \( k \) is the declaration constant, \( R \) is the distance travelled by the pellet and \( v \) is the velocity.

By rearranging equation 3.5 and substituting \( v \) for mach 0.5, the range at which the transition between transonic and subsonic flight occurs can be calculated. If a range calculated for the given time using equation 3.4 is greater than the transition range then it is the sum of the transonic range and that calculated for the remaining time using equation 3.3.

Equations 3.3 and 3.4 give the distance travelled along the direction specified by the two angles \( \theta \) and \( \phi \) (figure 3.1) generated from a normal and uniform distribution respectively. This gives the location of the pellet at that time.

The existing implementation also modelled the clay flight and performed the intersection calculations between this and the pellets. The clay was approximated by a sphere following the equations of motion with constant acceleration.

\[ v = u + at \] (3.6)

\[ s = ut + \frac{1}{2} at^2 \] (3.7)

where \( v \) is the velocity of the clay, \( u \) is the initial velocity, \( a \) is the acceleration (gravity in this case), \( t \) is time and \( s \) is distance travelled.
Calculations of the intersection between the clay and the pellets were performed by assuming the clay motion during a frame was negligible. If the pellet distance to the clay is greater than 5 metres no intersection calculations were performed. Within this distance each pellet is checked for collision with the plane passing vertically through the centre of the clay sphere in that frame. If this occurs then the intersection point is checked to see whether it is within a circle on that plane.

3.1.1 Evaluation of the Existing Implementation

The implementation [16] stores each individual pellet parameter as the two angles (θ and φ). To produce the vector direction to scale the cosine and sine of both angles are taken. Sine and cosine are slow commands and their implementation and speed is hardware dependent. This is particularly important because four of these commands were run for each pellet for every frame in the animation.

A large proportion of the main program for the simulation including lengthy initialisation code was written into one header file. This made the header file bloated and unsuitable for importing into the program, which only required a small proportion of the code. Also small functions integral to every pellet calculation were not inlined.

The clay modelling was very basic using a sphere and following a very simple flight model. The collision detection algorithm could therefore also be very simple. The assumption made that the clay is stationary is valid for high frame rates but may cause inaccuracies at lower ones. As a result the clay model and collision detection code were deemed unsuitable for use in this project.
3.1.2 Modified Pellet Cloud Implementation

The main change to the structure of the code is to change the pellet parameter storage to contain a direction vector ($\vec{v}$ in figure 3.1) rather than the two angles. This results in more memory overhead for the three coordinates but means that the sine and cosine are evaluated only once, on initialisation of the pellet cloud.

Some of the methods for generating distribution samples have also been changed to attempt to improve the efficiency of the initialisation of the pellet. The new algorithms have been taken from [29]. Also many of the smaller functions called only in one place have been inlined.

3.1.3 Validation of the Modified Implementation

The old implementation had been validated against the mathematical model to ensure correctness. All of the changes were thoroughly checked against this to ensure there were no errors. The new code was checked by first removing the sampling part of the code and several randomly selected values hard coded into both implementations. The results in all cases were the same.

The distributions were compared against each other to check that the samples generated were distributed identically. A large number of samples were divided into small buckets and the number of values in each one were compared, the results were similar. This was also done for the location of pellets in a cloud at several different times.

As a final check the generated pellet clouds were visually compared to ensure there were no obvious errors. Examples can be seen in figures 3.2 and 3.3 where the results after one second of the old pellet model are shown in blue, and green for the new model. One second is considerably greater than the usual flight time of a pellet cloud but in this case it demonstrates that the models are very similar.

3.1.4 Clay Modelling

The flight of the clay is modelled after discussion with a Microsoft game programmer who shot clay pigeons. The model uses a drag factor that reduces the forward velocity over time as shown in figure 3.4. Gravity acts downwards on the clay and a lift factor reduces this as the clay begins to descend. This gives the effect of the clay hanging in the air as it reaches the top of its flight path. These factors can easily be changed for different effects.
Figure 3.2: The pellet spread in the Z plane (following the pellet cloud).

Figure 3.3: The pellet spread in the X plane (side on to the pellet cloud).

Figure 3.4: An example clay flight path.
3.1.5 Collision Detection

Calculations to determine whether the pellet cloud intersected with the clays are required to determine a ‘hit’. The code in the existing implementation assumed basic calculations described in 3.1.1, thus rendering them unsuitable for slowed replay.

Collision detection calculations are required for every pellet with every target. To reduce the execution time required for this function, it is possible to use bounding objects. A simple bounding object is constructed to contain each object that may collide. These objects are chosen to match the shape of the object and allow simple collision calculations. A fast equation is evaluated for the bounding objects and if they overlap then more accurate collision detection is performed.

Bounding The Clay

A sphere is used to bound the clay for the time of one frame, see figure 3.5. This is simple to generate given the centre of the clay at the start and end of the time period. The distance of the furthest point on a clay from its centre is used to extend the line of motion in both directions. This gives two points in space, which are averaged to get the centre of the bounding sphere and the radius is the length of one point minus the centre.

Bounding The Pellet Cloud

A bounding sphere is generated to contain all the points in the pellet cloud. To ensure that all the pellets are enclosed it is generated a little oversize for the end time of the represented period. The pellet with the widest angle from the gun direction is used with the smallest and largest pellets to get the maximum
Figure 3.6: Pellet Cloud bounded by a sphere.

and minimum possible pellet locations at the widest angle. A sphere is then generated to include those pellets as shown in 3.6.

For the pellet cloud motion a capsule \([19]\) is used to bound the pellets. This is a cylinder with two hemispheres at each end. This has been chosen because it encapsulates the motion of a sphere along a line. The line through the centre of the cylinder part of the capsule is determined by finding the location of a pellet with average parameters at the beginning and the end of the time period. This line can be represented parametrically using equation 3.8

\[
P = P_0 + \mu(P_1 - P_0) \tag{3.8}
\]

where \(P_0\) and \(P_1\) are the end points of the line and \(\mu\) clamped between 0 and 1.

Bounding Object Intersection

The point representing the centre of the clay bounding sphere is compared to the line through the centre of the pellet capsule. The point on the line at which the sphere centre is closest is calculated. A point is closest to a line when the angle of the vector between the line and the point is a right angle \([19]\), see figure 3.7.

In 3D the perpendicular to a line must lie in a plane that has a normal equal to the gradient of the line. Consider a plane determined by the equation 3.9.

\[
Ax + By + Cz + D = 0 \tag{3.9}
\]

where \(A, B, C\) and \(D\) are constants and \(x, y\) and \(z\) are the \(x, y\) and \(z\) coordinates of any point on the plane. The normal to that plane is the vector \((A, B, C)\), it is the gradient of the cylinder centre line. The plane equation can be resolved by finding \(D\) with the point in the centre of the clay bounding sphere on the

24
plane, $\mu$ can then be resolved by substituting the $x$, $y$ and $z$ equations from 3.8 into the equation 3.9. The resulting value of $\mu$ corresponds to the point on the cylinder line which is closest to the clay centre as demonstrated in 3.8. If the point is not within the limits of the pellet cloud motion for this time period ($\mu$ between 0 and 1) then it is clamped to the closest end of the line.

The vector corresponding to the minimum path between the bounding objects is determined by subtracting the clay sphere centre and the clamped point on the capsule line. The length of this vector is the closest distance that these two bounding objects pass each other during flight. If this less than the sum of bounding spheres of the clay and the pellet cloud radii then further comparison is required. If this is not the case then the objects do not pass close enough during this time period to warrant more accurate calculation.

**Pellet-Clay Intersection**

Once it has been determined that the clay and pellet cloud pass close enough to possibly collide, further tests are performed on each individual pellet. These
are implemented as line intersections with the cylinder that contains the clay. The clay shape is similar to a cylinder and intersection calculations for this are much faster than with each polygon used to render the clay.

To simplify the calculation of two moving intersecting objects, the motion is considered relative to the clay. The location of each pellet at the end of the time period is adjusted by the motion of the clay. The clay is then considered to be stationary.

First the minimum distance between the line through the centre of the clay and the adjusted pellet motion vector is calculated. The shortest vector between the two lines is uniquely perpendicular to both lines [20]. If the pellet motion line has the direction vector \( u \), the clay cylinder centre has the direction vector \( v \) and the shortest vector between the two lines is \( w \) then:

\[
 u \cdot w = 0 \tag{3.10}
\]

and

\[
 v \cdot w = 0 \tag{3.11}
\]

By substituting in the equation of the vector between the two lines in the form of equation 3.8 (using one point on each line) and solving these two equations, the two end points of the vector can be found. Each end point of the vector is on one line and as the lines are parametric the result for each point is a value of \( \mu \) for each line equation (3.8).

The intersection of the line of the pellet motion with the top and bottom plane of the clay cylinder is calculated in a similar manner to the line-plane intersection used in section 3.1.5. Two values of \( \mu \) for the equation of the pellet motion line (in the form of equation 3.8) result from this. These give the line segment contained between the two planes.

If the \( \mu \) for the closest distance of the pellet to the centre line of the clay is outside the line segment then it is clamped to be contained within. Figure 3.9 shows the closest point the pellet passes to the centre line of the clay. It is above the top plane of the clay cylinder so is clamped to the value of \( \mu \) that represents this point.

The distance of the resulting (possibly clamped) point from the centre line is calculated. If this is less than the radius of the clay then a collision has occurred.

### 3.1.6 Evaluation

Average times for the evaluation of one million generations of the pellet cloud parameters and the time taken to compute the pellet locations of the same number of cartridges after 0.5 s are shown below for the original and the modified implementation.
Figure 3.9: Pellet motion to clay centre.

<table>
<thead>
<tr>
<th></th>
<th>Cloud generations (s)</th>
<th>Location after 0.5 s (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existing implementation</td>
<td>648.4</td>
<td>442.2</td>
</tr>
<tr>
<td>Modified implementation</td>
<td>649.4</td>
<td>124.3</td>
</tr>
</tbody>
</table>

This shows that the new implementation of this model is approximately 3.5 times faster for calculating the pellet location. Finding pellet locations is evaluated up to four thousand four hundred times per frame (2 pellet clouds of up to 1100 pellets and a start and finish point for their motion) while the pellet clouds are still in flight.

The time steps used to perform the intersection calculations are not fixed: they are determined by the time of the previous frame rendering. The length of the time period varies depending on the objects rendered and when the pellet cloud was fired. Unfortunately this difference in frame time causes the pellet cloud to have effects such as regularly missing the clay in the game mode but hitting it in replay mode.

3.1.7 Considered Solutions

Initially it was thought that the problem with varying results in replay was caused by the frame time dependence in the clay flight path. The future velocity for the clay was calculated based on the previous frame time. There are two solutions to this problem: to sample the clay flight only at specific times and interpolate from those, or to find a more reliable method which gave the same flight path every time.

The latter solution was chosen for reasons of speed and accuracy. With this method, there is no need to run two end point calculations and then interpolate the results. In a rapidly changing flight path the interpolation could also become visible. After several attempts to rewrite the current flight path method to fit this idea, it was decided to use a parametric spline curve fitted to the current
clay model described in 3.1.4. This ensures a smooth, fast and reliable method of calculating the clay location.

After experimenting with this new clay model, the same problem with varying results in replay occurred. It was determined that this was due to the collision detection using interpolation and being sampled at irregular intervals. Both the clay and the pellet movement through the time to render one frame were being modelled as linear for calculating collisions. Although the pellet flight is in one constant direction, a linear interpolation of it is later shown to give a significant error.

Again there are two solutions similar to the previous ones. First to run all the calculations at a given rate. This presents the same problems as before and could prove hard to implement. The other solution is a numerical solution. The pellet flight path could be equated to the clay and an exact result achieved. The advantage of this method is that this calculation need only be run once rather than every frame.

3.2 Second Attempt

A numerical solution was considered superior because it would give exact results and remove the need for interpolation. This would result in the program being faster as the calculation need only be run once when the gun is fired.

Equating the clay spline curve (see section 3.1.7) to each of the sphere modelling equations (equations 3.4 and 3.3) was found to be unsuitable. Equation 3.4 for pellet motion required solving a quadric and the time required to do this is large. Equation 3.3 creates an equation of the form of equation 3.12.

\[ at^2 + bt + c - \ln (tv_0k + 1) = 0 \] (3.12)

where \( t \) is unknown and all other variables are constants.

As far as can be established, there is no analytical solution to this equation. An iterative method such as Newton-Raphson could be used but that is not ideal as the number of iterations required may be high. Parametric splines were therefore considered to model the pellet flight path in addition to the clay. The advantage of this is that the both flight paths are of the same form and can be simply equated. Parametric splines are of the form 3.13.

\[ P = a_2\mu^2 + a_1\mu + a_0 \] (3.13)

where \( \mu \) is varied to get a point at \( P \) and \( a_2, a_1 \) and \( a_0 \) are vectors of constants.

In 3D \( a_2, a_1 \) and \( a_0 \) are all vectors of order three, resulting in \( P \) being a 3D point in space. \( \mu \) is replaced by time and the \( a \)'s are calculated by substituting

\footnote{see 3.2 for a detailed explanation of splines}
three results from the existing pellet model into the equation and solving it. This results in a spline equation that returns the location of the pellet for time passed as a parameter.

### 3.2.1 Model Validation

Initially a single spline was used to model the pellet location. The three points required for the curve were taken as the starting location, the time at 0.5 s and the time of transition between transonic and subsonic flight (around 0.17 s). Pellet flight in the model is in a single direction so a comparison can be made using the distance traveled along that line.

One spline (figure 3.10) has a maximum error of around 3.5 m at 0.33 s which is unacceptable. To approximate the model better two splines (figure 3.11) can be used, one for the subsonic and another for the transonic time period. The third centre point used is the average of the two end times.

The spline modelling the subsonic time of travel has a maximum error of 0.00002 m, a fiftieth of a millimetre, against the model. The subsonic spline has a much greater error due to its curve. The largest error occurs at 0.03 s and is 0.85 m. This can be reduced by dividing the first spline into two. By using up to five spline curves the error is dominated by that in the first spline curve. Figure 3.12 shows repeated subdivision of the splines to reduce this error.

The values from the model are results taken using a high sampling rate (around 200 Hz). By sampling the model at a much lower rate and linearly interpolating between these points the accuracy of the current intersection cal-
Figure 3.11: Division using two splines modelling the subsonic and transonic time.

Figure 3.12: Subdivision for four curves.
Calculations can be estimated. It can be seen how close the four-curve approximation is to the model in figure 3.13. The linear interpolation, model results and this approximation are almost along the same line. In fact four-curves are more accurate than the linearly interpolated values at 50Hz. The errors are shown in table 3.2.1, where time and range correspond to when the error is greatest.

<table>
<thead>
<tr>
<th>Curve</th>
<th>Max Error (m)</th>
<th>Time (s)</th>
<th>Range (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Interpolation (50 Hz)</td>
<td>0.18</td>
<td>0.01</td>
<td>3.77</td>
</tr>
<tr>
<td>One Spline</td>
<td>3.5</td>
<td>0.34</td>
<td>67.64</td>
</tr>
<tr>
<td>Two Splines</td>
<td>0.85</td>
<td>0.03</td>
<td>10.31</td>
</tr>
<tr>
<td>Three Splines</td>
<td>0.23</td>
<td>0.015</td>
<td>5.52</td>
</tr>
<tr>
<td>Four Splines</td>
<td>0.05</td>
<td>0.01</td>
<td>3.77</td>
</tr>
</tbody>
</table>

This table shows that four spline curves is a considerably better approximation than the linear interpolation. It is not necessarily the case that the four curves need to be used to get the optimum results. A shot is not normally fired when the clay is within a certain distance. As the most significant error occurs within the first few hundredths of a second, a slightly less accurate model could be used to increase speed but still give a similar results within the simulation. The greater the number of curves used the higher the computational cost for collision detection so using more than four curves is generally deemed unnecessary.
3.2.2 Spline Implementation

The flight for the clay was implemented as a single spline that modelled the centre of the clay during motion. Each pellet was modelled as a number of splines as described above. The code uses a single compile-time constant to determine how many times the pellet path is divided in the form described above.

As the conversions to spline models for the clay and the pellet cloud were implemented at different times, they were originally coded separately. They also have different requirements as the pellet is modelled as a particle but the clay has dimensions. Originally the intersection calculations were split across several classes making the design unclear and irregular. This also meant that adding new targets with flight paths that could not be modelled with a single spline would be difficult.

The program was refactored to move all the spline and intersection code to one place. This has the added advantage that the pellets are modelled in the same way as the clay. The new structure involves a ‘Spline’ class containing one modelling spline and object dimensions as an axis aligned cuboid. The ‘Spline’ class is used by the ‘MultiSpline’ class, which stores the required numbers of splines to model the path of the given object. The correct spline is chosen depending on the requested time for a point.

The implementation wraps up the modelling code for all of the objects into several self contained classes. This will simplify the inclusion of other objects (such as birds) whose flight can be readily modelled by a collection of splines. The intersection code is already implemented within these classes so only the framework to instruct comparison will need adding.

3.2.3 Collision Detection

Calculations, detailed below, are performed to determine exactly when the pellets and the clays intersect. As the method is numerical, this only needs performing once and the result stored. It must however, be performed for each pellet and each spline. The program starts with the first spline modelling and then loops through the others until an intersection result is achieved.

There are three axes for intersection to occur, one axis will be described (the others are similar) and then how they are combined will be demonstrated. Every object is represented as a spline modelling the motion of its centre with an interval in that axis to describe its size. This is the same for pellets except their dimension is zero. This generalised modelling means that the code can be used for detecting other collisions such as between two clays or pellet bounding objects. The code is written in such a way that this generalisation has very little impact on the efficiency but allows the maximum flexibility.

Using bounding objects similar to section 3.1.5 for fast collision detection is
simple. An average pellet flight path with the maximum dimensions of the pellet cloud is used to intersect with the clay. This calculation determines whether to test each of the pellets individually.

The average clay path represented by a dashed line in figure 3.14 shows the path the clay spline models. The minimum and the maximum lines show the interval of the object in this axis. The pellet flight path is shown as a line, with dimension zero. As can be seen from this figure, there is a period where the pellet overlaps with the clay in this axis.

These points can be found by equating the two spline curves:

\[ a_0 t^2 + a_1 t + a_0 \pm \text{dimension}_a = b_0 t^2 + b_1 t + b_0 \pm \text{dimension}_b \]  

(3.14)

This is a quadratic and can be solved as such, resulting in two solutions for each minimum/maximum pair of times. As can be seen from the equation 3.14 the dimension is part of the constant, this is considered in the implementation to increase the speed of the calculation of the four results. The results are checked to see whether they lie within the time period for which this spline is responsible. Cases are also considered for one and zero intersections.

The results from the three axes are compared to see if there is a time period where a result from each axis overlaps. If this exists then the pellet is inside the clay for this period. The lower bound of this time period is considered the collision time.

### 3.2.4 Plane Calculation

When viewing the replay of the pellets from the firing point it is very difficult to tell when they are at the same range as the clay. Knowing this will allow the
user to determine by how much they have missed.

The initial idea was to record the closest point the pellets passed to the clay. This had to be calculated while the pellets were in motion, recording their distance each frame. This is a frame rate dependent method and not ideal. When viewing the results in slow motion groups of pellets changed at the same time. The problem was that in real time the sampling rate was not great enough and many pellets passed at their closest at one sampling time.

The chosen solution calculates when the pellets passed through the plane of the clay, as shown in figure 3.15. With this method the question arises as to which clay to use - several clays may be released simultaneously. The calculation is evaluated for all clays and the one the pellets pass closest to is the one that is used. If the pellets are greater than a constant distance to any clay at all times then it is ignored.

This method involves first calculating a plane for each clay. Unfortunately this plane must be relative to the clay’s location at the time the pellets pass through it. To simplify the problem the clay was considered to have a constant direction at the time the pellet cloud is fired until the pellets pass through this plane. This is only around 0.15 s so this is a reasonable assumption for most targets, as users must be capable of tracking them.

To calculate the plane the current (at the time of firing the shotgun) direction of the clay is calculated. Also the vector corresponding to the direction from the user to the clay is calculated. The cross product of the direction from the clay to the user and the direction of motion is taken to get another vector (figure 3.15). This and the clay motion vector are considered as the two vectors needed to define a plane. The point on the plane was that of the current clay location.

This gives a plane containing the clay’s immediate motion with a normal towards the user. This plane is equated to the pellet spline curve’s equation 3.15 to achieve the time of intersection. If the time achieved is modelled by the
current spline curve then this is represents the time the pellet passes through the plane of the clay’s motion.

\[ a_2 t^2 + a_1 t + a_0 = P + r\vec{m} + s\vec{n} \] (3.15)

where \(a_2\), \(a_1\) and \(a_0\) are the vectors from the parametric spline, \(t\) is the time of intersection, \(P\) is the clay’s current location, \(\vec{m}\) and \(\vec{n}\) are the two vectors defining the plane and \(r\) and \(s\) are the parameters of the plane. Figure 3.16 shows the result of this calculation. The yellow pellets have not passed through the clays plane but the red ones have.

The calculation for when the pellets pass through the clays motion gives a better result than simply using the point at which the pellets pass closest to the clay as shown in figure 3.17. The figure shows the difference in the two results. If a shooter was given the minimum distance the pellets passed to the clay and they then adjusted their aim to compensate, then it is likely that they would still miss the next clay from the same trap. The plane of motion calculation gives a much more accurate result of the distance in front or behind that the shooter should be aiming at.

### 3.2.5 Motion Prediction

An extremely valuable feature of a training system is the ability to show the user where they need to aim in order to hit a moving clay. This is termed ‘clay prediction’. A similar feature (‘pellet prediction’) shows where the clay would have to be along its current flight path in order for a shot made at the current time to hit the clay.
Clay Prediction

This involves predicting where the clay will be in the time it takes for the pellet cloud to reach it. Drawing a fake target at this point gives a target to aim at which if used correctly will result in a hit on the real target.

This ‘clay prediction’ facility is implemented by calculating the current clay location and its position in 0.15 s. This figure is chosen as an average pellet flight time so the linearly interpolated is as close to the real result as possible. The planes described in section 3.2.4 are calculated for each point. A simulated gun is aimed at both clays and the time taken for an average pellet to intersect the clay plane is calculated.

The assumption is then made that the clay distance from the user and the times of intersection of the clay with the pellets vary linearly. This gives two equations, 3.16 and 3.17

\[
\text{clayDist} = (t - \text{curTime}) \frac{\text{dist}1 - \text{dist}0}{0.15} + \text{dist}0 \quad (3.16)
\]

\[
\text{pelletDist} = (t - \text{intT}0) \frac{\text{dist}1 - \text{dist}0}{\text{intT}1 - \text{intT}0} + \text{dist}0 \quad (3.17)
\]

where \(\text{clayDist}\) is the distance from the clay to the user, \(\text{pelletDist}\) is the distance from the pellet cloud to the user, \(t\) is the predicted time of intersection between the clay and the pellet cloud if correctly aimed, \(\text{curTime}\) is the current time, \(\text{dist}0\) and \(\text{dist}1\) are the clay location at the current time and 0.15s in the future, \(\text{intT}0\) and \(\text{intT}1\) are the times the pellet intersect the planes when fired at the plane of the clay at the current time and at 0.15s in the future respectively.

The time at which this clay and the pellet cloud are at the same distance
Figure 3.18: The faint clay on the right is the predicted location of the clay on the left, taking into account the location of the shooter and target.

Figure 3.19: Calculating the predicted clay position and showing the error.

from the user is found by equating those equations. The predicted clay is drawn at this time on the clay flight path as shown in figure 3.18. This can be aimed at to hit the real clay.

The clay’s true location is not exactly at the linearly interpolated position because the clay flight is not straight. The maximum error in the resulting time is that taken for the pellets to travel between these two locations. The error in the predicted location is the distance the clay travels in this time, see figure 3.19.

This error is relatively small because the clay motion can be considered reasonably smooth over the time taken for the pellets to reach it. Also the pellets travel very quickly so the clays motion during this period is very small.
Figure 3.20: Calculating the predicted clay position relative to the current aim.

**Pellet Prediction**

The pellet prediction demonstrates where the clay should be if the gun were fired at the current time. The real and the predicted clay must be aligned to hit the real clay.\(^4\)

This method is an extension of the methods used in section 3.2.5. The time and location of predicted intersection is again calculated as before with the simulated gun pointing at the clay. The location of the predicted clay is subtracted from that of the clay at the current time. This results in a vector of the motion of the clay during the time of travel for the pellets to the clay. A predicted clay is then drawn (as in figure 3.21) at the location of the pellets at the intersection time with the current aim subtracting the clay motion vector, see figure 3.20.

### 3.3 Comparison of the Two Attempts

The most important consideration is accuracy. As discussed in section 3.2.1 the splines are only an approximation to the model. To use the model directly samples must be taken at regular intervals and interpolated. Compared to this approximation, the spline model gives more accurate results.

The spline model is slower on initialisation. The same generations as the model implementation must occur but a spline approximation to this and intersection calculations must also be evaluated. After testing with 10 clays and the maximum number of pellets (neither of which are common), the time delay is

\(^4\)This was affectionately named ‘Claybuster’ mode after its similarity with the ‘Dambuster’ story
only just noticeable and certainly not significant.

The results for 1,000,000 repetitions can be seen in table 3.3. The collision detection is run for hitting one clay. Comparison of the efficiency of the two methods is difficult as they greatly differ in structure. The first attempt appears faster until it is considered that the collision detection for this must be run every frame while the bounding objects intersect. The calculation is variably faster when these do not overlap.

<table>
<thead>
<tr>
<th></th>
<th>1st Attempt (s)</th>
<th>2nd Attempt (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Generation Time</td>
<td>649</td>
<td>954</td>
</tr>
<tr>
<td>Cloud Location Time</td>
<td>124</td>
<td>103</td>
</tr>
<tr>
<td>Cloud Collision Detection</td>
<td>976</td>
<td>1574</td>
</tr>
</tbody>
</table>

The significant delay using splines occurs immediately after the gun is fired, as it is recoiling. It is unlikely that anyone will be firing the gun and will not be taking aim at another target during this period. Due to the form of these equations it would be simple to speed up the calculations by running them on several, possibly distributed, processors concurrently.

Locating the pellet every frame is faster when using the spline curve model. There is also no need to run intersection calculations every frame. In the existing implementation [16] the calculations can be seen to visibly impact the frame rate (and also the apparent velocity of the clay!).

The simplicity and speed using splines becomes more significant for other calculations that must be performed. The numerical aspect to the splines allows for simple and fast solutions to plane intersection and motion prediction, which must be run in real time. These have not been implemented for the first attempt but initial ideas on methods are much more complex and slower. The spline approximation also separates the model from the collision detection allowing simple updating of the model or addition of different targets.
Chapter 4

Simulation Framework

This program consists of a main loop that processes events and instructs the rendering\(^1\) to occur. Event handling consists of responding to computer and human input events. The painting and event handling will change depending on which state the program is in.

4.1 States

An interface for the state is specified. The program loop sends the events, keyboards status and paint calls to a pointer of the interface type. This pointer is initialised with the program to an instance of the main menu, see section 4.1.1.

The different states are:

1. Main Menu
2. Simulation Selection
3. Simulation
4. Training Mode
5. Tracking Mode

A pointer to the class containing the main loop is maintained in a global namespace. The different state are then responsible for loading the correct next one into the pointer used in the main loop. This is implemented as a function call to the main loop class, which deletes the current state and replaces it with the new one. At this point the next simulation state should be loaded and ready to receive events.

\(^1\)Detailed rendering algorithms are given in chapter 5
4.1.1 Main Menu

The main menu screen is initially loaded, figure 4.1. This is a simple screen with basic menus such as help, about, quit and start. A background picture is loaded from a file.

The about box is simply an overlaid frame containing text, a key press or click will exit from this frame. Help is similar to the about box. Start will change the state to the simulation selection menu, section 4.1.2.

The buttons throughout the program are all instances of a standard class. This class can load the button as either an image or a centred text string. All text is drawn through one globally accessible renderer to simplify generation and use of strings.

4.1.2 Simulation Selection

Before the simulation can be run several parameters must be set. These include varying the type of cartridge and shotgun. The scene must also be chosen before it can be loaded. These choices are made through the simulation selection window, as shown in figure 4.2.

Centre left in figure 4.2 is an image of the currently selected scene. This can be a screen shot or another image, all that is required is that it has a standard name in the scene directory (see section 4.5). Similarly for the scene description, shown bottom left of figure 4.2, this is a text file with a specific name in the relevant directory.

Rather than load all the images and descriptions at run time, which would be time consuming, they are loaded as required. The first scene in the list is loaded initially. Any other scenes will have their data loaded when they are
selected. The data is then cached in case the item in the list is clicked on again. All this data is discarded when the simulation is run to free resources.

4.1.3 Simulation

Once the scene and shotgun details have been selected the simulation is loaded, see figure 4.3. This renders the chosen landscape and implements the rules set for launching clays. Help is available throughout the simulation and training modes by pressing ‘F1’.

The clays can either be set to release on a timer or by pressing the space bar. There can be several traps set to release, either in order or randomly. Once fired the clay can be shot at using the mouse to track and the left mouse button to fire.

The simulation is set so that only two cartridges are available in the shotgun. These are represented in the lower right hand corner of figure 4.3. They are greyed out when they have been fired and must be reloaded by pressing ‘R’. The current number of hits of the pellet cloud fired from each cartridge is shown next to it. In slow motion the counter changes as the pellets collide with a target. The current time and scalar for simulation time (see section 4.2) is also shown here.
The ‘Claybuster’ and pellet tracking mode in section 3.2.5 are toggled on and off by pressing the ‘o’ and ‘p’ buttons respectively. To enter the training or replay (section 4.1.4) mode ‘t’ is pressed.

4.1.4 Training Mode

The training mode defaults to start at the time of the last reload. Five controls are shown in the centre bottom of figure 4.4 to control the replay. The buttons are large so when this project is integrated with the shotgun tracking mechanism [17], they are easy to select using the shotgun as a pointing device. The buttons control the replay direction, pausing and speed.

Beneath these is the time line. This has a marker to show the current time relative to the recorded history. Also shown are markers representing the times of the gun reload and firing. This bar can be clicked with the mouse to move to the time corresponding to that position in real-time.
Right clicking the mouse switches it between controlling the camera view direction and a pointer for use with the buttons. To return to the simulation ‘t’ can be pressed again; the simulation resumes at the time it was suspended.

4.1.5 Tracking Mode

To enter tracking mode the user must first be in training mode and must then press ‘tab’. This will make four more buttons on the bottom left of figure 4.5 available. A large 3D arrow also appears pointing to one of the objects flashing and the simulation immediately pauses.

Either a clay or a pellet cloud can be pointed to and pressing ‘tab’ again will move to the next object. This represents the currently selected object. If a pellet cloud is selected the cartridge it was fired from will also flash. Once the desired object has been selected one of the four buttons in the bottom left of the screen can be pressed to track that object.

The four buttons represent different ways to track an object:

1. Follow - This tracks above and behind the object. The camera looks directly at the object.
2. Above - The camera is positioned directly above the object looking down on it. Up in the screen is the direction of motion projected into the plane of the screen.
3. Ride - The camera is positioned with the object. It flies as the object with the view direction being the direction of motion.
4. Free - This is the most flexible view. When the mouse pointer is not visible (right-click to toggle) the mouse controls the position of the camera around the object. The camera always looks at the object. To lock the view the mouse pointer can be set to visible.
In all of these modes the distance from the object can be adjusted using the mouse wheel.

The speed controls can all be used while in tracking mode. To select another object to track or another mode of tracking the tab button can be pressed again. If the replay is resumed while no object is being tracked the simulation returns to training mode, section 4.1.4.

4.2 Clock

A clock is required for the program and is implemented as a class. Anything that requires time must request it from the single instance of this class. The instance is instructed to update its time on each execution of the program loop. This synchronises the entire program at each frame.

The program is run off two different types of time, both controlled by this class. Either a frame rate or a clock reading can be requested for each of these types. The first type is real time and is used for any animations that must be relative to user time. The second type is a scaled offset time, which is used to perform simulation animation. The simulation can be slowed by scaling the frame time. During replay this clock is reset to the user requested time and the animation performed from there. To continue the simulation the clock is automatically set to the time replay mode was entered.

4.3 Recording and Replay

Recording of all actions is required to replay the simulation at a later date. This is implemented as a single globally accessible object. Every generated object is responsible for registering with the recording class. Objects that require registration are those such as clays and pellet clouds.

Enough data must be passed to the recording class to regenerate the object in the future. For example, a clay requires the launch time and the trap launched from. These parameters are saved in a structure in a vector.

During replay the recording code is responsible for recreating the objects at the correct time. This is simple due to the clock (see section 4.2) time being reset, objects can be constructed at the original time. It is implemented by iterating through the vectors and checking the construction time of objects.
4.4 Sound

The SDL [12] interface is used to play audio during the simulation. The audio clip is passed to a function which stores the clip and initiates playing it by passing it to the library. A callback procedure has been written to load the next parts of the clip on request. The sounds that have currently been implemented are a shotgun firing and the gun reloading. Adding further sounds would not be difficult, requiring only the file name to be passed to a function to load it and an instruction to play that sound at the correct time.

4.5 Describing a Range

To allow the shooting range to be varied, a means to describe them using images and simple text files has been devised. These files are loaded and verified at run time to give maximum flexibility in range design. The structure has been developed so that it is simple and intuitive to add a new ranges.

The files describing a range are stored in a single directory, which must be in the ‘./Scenes’ directory. Also in this location there must exist a file named ‘SceneNames.dat’. This file contains the names of the directories of the scenes separated by a new line. These are also the names used to describe the scenes for simulation selection (section 4.1.2). Using this method instead of automatically retrieving the directory names is simpler and does not require an extra library to make it cross platform.

Inside each directory there should be a ‘Desc.txt’ file and an ‘Img.tga’ file which describe and display the range when they are clicked on during selection in section 4.1.2. The file must also contain a ‘scene.dat’ file. This file contains an XML style description of the range details. The scene description file is divided into several parts describing different aspects of the range.

An example scene description file is given below. Each block is enclosed in brackets.

```
CLAYS {
  NUMSYNCS 1
  SYNC {
    NUM 1
    CLAY {
      LOCATION x: 370 z: 340
      VELOCITY x: -18 y: 15 z: 8
      LIFTFACTOR 1
      FRICTION 1
      TIME 0
    }
  }
}
```
TIME 0
}

TERRAIN_DATA {
    DIM x: 2000 y: 500 z: 2000
    HEIGHTMAP heightmap.tga
    TEXTUREMAP texmap.tga
    TEXTURES {
        NUM 1
        TEXTURE {
            DIM x: 500 z: 500
            FILE grassTex.tga
            TEXTUREMAPVAL 0
        }
    }
}

FEATURES {
    FEATUREMAP featmap.tga
    ASEOBJECTS {
        NUMASEOBJECTS 3
        ASEOBJECT 1 {
            FILE Stone.ase
        }
        ASEOBJECT 2 {
            FILE Tree.ase
        }
        ASEOBJECT 3 {
            FILE Bush.ase
        }
        NUMASEFEATUREVALS 2
        ASEFEATUREVAL {
            ASEOBJECTNUM 2
            FEATUREMAPVAL 255
            MINSPACING 2
            DENSITY 4
        }
        ASEFEATUREVAL {
            ASEOBJECTNUM 1
            FEATUREMAPVAL 85
            MINSPACING 1
            DENSITY 30
        }
    }
}

SKY {
    DIVISIONS 32
    PLANETRADIUS 4000
    ATMOSPHERE_RADIUS 5500
    TEXTURE {
        DIM x: 1 z: 1
4.5.1 Trap Description

The trap block is enclosed in a ‘CLAYS’ keyword. Beneath this is the number of synchronised blocks. A synchronised block describes a group of clays that will all be released as a group. These clays will all release for one ‘pull’ command.

Each synchronised block begins with the number of clays it clusters together. A clay is described by a launch location, an initial velocity in vector form, a lift factor, friction and a time delay. The time delay is the number of seconds between this clay’s launch and the next in this block.

During flight the clays are subject to a gravitational force. When the clay is descending the lift factor is subtracted from gravity to give the downward force acting on it. Friction is the deceleration force acting in the horizontal direction during the clay’s flight. By varying these parameter many different clay flight paths can be described.

At the end of a synchronised block is a ‘TIME’ keyword and if this is set to 0 then the clays release on a ‘pull’ command. If time is a non negative float then this represents the delay between the trap firing its last clay and the next firing.

4.5.2 Terrain Description

The terrain is saved using a height map. This is a simple bitmap image with the colour of each pixel representing the height of the terrain at this point. The size of the terrain is given in the ‘TERRAIN_DATA’ block in metres. These values scale the bitmap coordinates to get terrain of the required dimensions. The name of the height map file is also given here.

Applying textures to the terrain makes it more realistic and allows more aspects to be added such as roads, lakes, etc. These are also described by this block. The texture map is similar to the height map except its pixel values correspond to the texture at that location.

The different textures are described in the ‘TEXTURE’ sub block. Each texture within this has a scale for mapping it to the terrain, the file in which it is located and the pixel value to which it corresponds. For more details on how the textures are mapped see section 5.1.2
4.5.3 Features Description

The feature map is also similar to the height map. The pixel values correspond to the feature rendered at that point. Every object to be rendered in the scene is described by a Visual Studio Max ‘ase’ file. Files of this type contain geometry, topology and textures. A list of the files describing the objects is given.

This is an example of a small file that describes a rock, the resulting rock can be seen in figure 4.6:

```
*3DSMAX_ASCIIEXPORT 200
*COMMENT "AsciiExport Version 2.00 - Thu Jun 02 21:41:42 2003"
*SCENE {
  *SCENE_AMBIENT_STATIC 0.3000 0.3000 0.3000
}
*MATERIAL_LIST {
  *MATERIAL_COUNT 1
  *MATERIAL 0 {
    *MATERIAL_AMBIENT 0.1791 0.0654 0.0654
    *MATERIAL_DIFFUSE 0.5373 0.1961 0.1961
    *MATERIAL_SPECULAR 0.9000 0.9000 0.9000
    *MAP_DIFFUSE {
      *BITMAP "marble.tga"
    }
  }
}
*GEOMOBJECT {
  *MESH {
    *MESH_NUMVERTEX 6
    *MESH_NUMFACES 8
    *MESH_VERTEX_LIST {
      *MESH_VERTEX 0 0.1 0 0.1
      *MESH_VERTEX 1 -0.1 0 0.1
    }
  }
```

Figure 4.6: Simple example ‘ase’ rock.
A description of what each pixel value corresponds to is given. First is the ‘ase’ object number so the file can be retrieved. Each of the pixel values have a minimum spacing for the objects. This is a guarantee that within the area of this colour the objects will never be closer than the distance specified. There is also a density, describing the average spacing of the feature. The greater the distance between the minimum spacing and the density, the more random the pattern of the feature location.
4.5.4 Sky Description

The sky is drawn as a curved plane centred over the users location. As the plane is curved it must be approximated with polygons and the number of divisions must be specified. The planet radius and the atmosphere radius are features of the size of the plane and of the curvature. The texture that is to be mapped to the sky plane and the scale of the texture to the size of the plane are also be specified.

4.5.5 Constructing a Range

To construct a range the required files must be given along with any files they refer to. The maps are a particularly simple way to specify a terrain. Using a good graphics package, the three required images can be overlaid so the structure of the landscape can be visualised.

Creating 3D objects with any design package is difficult unless the user is familiar with it. Visual Studio Max is a very powerful tool and converters from most formats to this are available from the Internet. Predesigned objects are also available to download from the Internet.

The height map, feature map and the large scale texture given in figure 4.7 construct the terrain shown in figure 4.8. The texture map for this terrain is a single colour and has not been given as the high detail texture is not visible from this range. The white box represents the shooter location with the tree features close to it. The other features (rocks) are not visible from this distance.
Figure 4.8: Resulting terrain from description given in 4.7
Chapter 5

Rendering Algorithms

Graphics add an element of entertainment and interest to the program. They also add perspective and more realism to a 3D world to improve the training function of the program. Perspective is very important to help with gauging the distance of the clay.

OpenGL was chosen as the graphics rendering library. It is a very common library and easier to use than Microsoft’s DirectX. It is also multi platform, which may prove useful.

The graphics implementation has been divided as much as possible. Each part of the landscape is implemented as a different group of classes. On initialisation there is a call to the main class for each section passing it the relevant configuration file and requesting the construction of a display list.

Display lists are a group of graphics rendering primitives. If several primitives remain unchanged then they can be collated into this list. Display lists are compiled by the graphics card to run as fast as possible and then saved in the graphics card’s memory. This makes them very quick to use requiring only one function call and very little communication between the processor and graphics card. They are highly suited to unchanging parts of the landscape such as the terrain or clay shape.

Once initialised, before every frame a call to the animate and paint function for each section of the landscape is made. The paint functions are inlined often to increase the speed of the rendering.

5.1 Terrain

The terrain of the landscape is saved as a height map. This is a bitmap image in which each pixel corresponds to a height in the terrain so it is simple to add
new maps but they provide a high degree of redundancy. If every point in the landscape is drawn the graphics card will struggle to render it. The solution is to reduce the number of triangles.

To do this a limited implementation of the ROAM [18] algorithm is used. This algorithm effectively reduces the number of triangles without introducing slithers (very narrow triangles the graphics card may fail to render correctly). In the future this implementation could be adapted to perform continuous level of detail (CLOD) as the user moves around the terrain.

5.1.1 ROAM

The ROAM algorithm divides the terrain into a number of square patches with each of these patches divided into two equilateral triangles. Similar to dividing a square into a quadtree, a triangle can be divided into a bintree [27], see figure 5.2. This is the structure used to subdivide these equilateral triangles.

For each equilateral triangle the error metrics are first calculated but the method for this can vary. The standard method is to calculate the vertical error in the plane if the node is not drawn (figure 5.3). This implementation uses this method but also considers whether there is a change in mapped texture across this triangle to form the error metric.

The error can also be dependent on the viewpoint. Distant details of the landscape are not so obvious as close ones therefore they can be removed. The error from above is adjusted by the straight line distance between the centre of the triangle and the user. Calculating the error for a single triangle has been separated into a function so it can easily be changed. The total error for any triangle is the greater of its own and that of its sub triangles error.
Figure 5.2: First 4 levels of a triangle bintree

Figure 5.3: Error of for ignoring a node

Figure 5.4: Dividing to the level below
Where the error is greater than a specified amount the triangles are subdivided until it is satisfactory. Triangles may only be divided if the neighbouring triangle on their base is attached by it’s base otherwise this can lead to holes in the terrain. The triangles can then be divided as shown in figure 5.4.

At the edges of adjoining patches the same problem with holes occurs if they are not split together. For this reason the patches must be ‘stitched’ on neighbouring sides. To split triangles that are not neighbours on their base side the forced splits must be carried out (see figure 5.5). Forced splits divide the triangles until this condition is met, possibly recursing through different patches across the stitching.

A display list was created from these triangles as the user remains within a small area. The structure of the triangles is saved so the height of the terrain at any point can be found. If these heights were read directly from the height map then features may appear to float or disappear, due to the triangulation only being an approximation. The other option would be to recreate the structure every time but this would prove too computationally expensive.

The terrain resulting from this implementation can be seen in figure 5.6. The triangles that would have to be used to render this terrain with any degree of accuracy without a level of detail algorithm is the smallest triangles visible. On the height map shown in figure 4.7, the required number of triangles to render the terrain is reduced from 2,093,958 to 14,567.

An attempt was made to use the triangle stripping tool provided by NVIDIA [26]. This proved very successful and the entire terrain was converted to one single strip. Unfortunately this had no effect on the frame rate. This is probably due to the fact that the terrain was compiled into a display list and therefore the same faster rendering primitives were automatically used.
5.1.2 Textures

Two textures are applied to the terrain at any point. The first is a very large texture that stretches across the entire map. This is a very low detail image that contains the very basic features of the terrain, satellite or aerial photographs look good. Blended with that is a high detail texture that gives the detail much closer to the user, for example be a grassy texture.

The advantage of this method is that the detail texture adds realism and perspective close to the user. This texture can be small and repeated. The global texture map gives the distant views and hides any patterns in the detail texture. This method offers good flexibility and acceptable results for very little storage space.

The results of these two can be seen in figures 5.7 and 5.8. In the high detail image the repetitive nature of this texture is hidden as the general colour changes across the image as a result of the low detail texture.

To use this dual texture technique an extension to OpenGL is used to improve efficiency. These extensions have been approved by the OpenGL review board and have now become standard on almost all graphics cards. As they are not part of the current version of OpenGL they must be loaded at run time from a library. This may cause problems on older graphics cards.

5.2 Features

Features are imported from files in the Visual Studio ‘ase’ format. Files of this type minimally contain the geometry, topology, normals, texture co-ordinates
Figure 5.7: Low detail in the distance.

Figure 5.8: High detail close up.
and texture map file information. No good free parsers are available on the Internet so one was written.

There are many possible options that could feature in an ‘ase’ file but most of these were considered irrelevant for this project. This simplified the parser greatly but does mean that not all designs can be displayed. The file format allows for animation but there was not time to implement this in the parser.

Once the ‘ase’ file has been parsed a display list of that object is constructed. The objects are placed based on the parameters given in the scene description file (see section 4.5.3). This algorithm attempts to make the placement of the features random but still adhere to the parameters.

One the positions of each of the objects had been decided upon a display list was constructed. This involved pushing a translation matrix onto the graphics rendering stack for each position and rendering the object display list at this point. The matrix at the top of the stack multiplies the rendered points, in this case it is used to translate them. This saved redrawing the object at every location. This is the display list that is rendered every time the paint function is called.

As can be seen from the tree placement screen shot (figure 5.9) this was reasonably successful. The pattern is semi random and no trees are laid on top of each other. Although it looks quite good, there are still patterns of a vague grid created by this algorithm. Ideally this will be replaced with a better method of placing features. Once a more suitable algorithm is found, it is very simple to replace this one by rewriting one function.
5.3 Sky

The sky for the simulation is implemented as a curved plane. This removed the problem occurring with the slivers in the dome. The curved plane also reduced the effects of stripping across the sky when the cloud texture is mapped to it. Mapping a small texture to the entire sky gives a good wispy effect as shown in figure 5.10.

The sky plane is rotated around its centre to give the effect of clouds moving. This is not apparent unless the user looks directly up for a long time. Another technique would be to change the texture coordinates on the plane. This would remove the centre problem but be slower to render as a display list could not be used.

5.4 Clay

As the clay is a lathed\(^1\) object only half of its cross section is required to construct it. These were taken by measuring a standard clay although this could be changed at a later stage. The normals for the cross section are calculated. The points and normals are rotated a given number of times around the centre to generate a 3D clay as shown in figure 5.11. The quadrilaterals produced are converted into triangles and the clay a display list of these points is constructed.

This clay is constructed by a special class, which is simple to alter or replace. The manager is responsible for releasing a new clay at the correct time and from the right trap. The clay initial conditions are converted to a modelling spline

---

\(^1\)A lathed object is one that rotation around one axis will result in the object appearing unchanged.
and the clay is then responsible for rendering itself using the standard display
list at the correct location for the given time.

If a clay is hit at a certain time then a constant number of particles are
generated. These are given random initial velocities and directions. The velocity
of the clay at that time is added to the generated velocities. These particles
are animated according to their initial velocity and gravity. This results the
clay appearing to turn to dust when it is hit, which rains down around where it
exploded. The particles are rendered as simple points in OpenGL (figure 5.12),
that fade over a given time.

5.5 Pellets

The pellets are very simple to rendered simply as points. Their position is
calculated for the current time from the spline model. They are rendered in
three different colours depicting those that have not reached the clay, those
that have passed it and those that have hit it, fading as they get further away.
The pellet cloud is not rendered during the simulation, only on replay.
Chapter 6

Summary and Evaluation

As a result of this project, a 3D clay pigeon shooting environment has been designed and developed. Five of the six objectives were met and as such it has been considered a success. The last objective was to provide a stereographic visualisation of the range. This should be relatively simple to implement as to do so requires only projection from two viewpoints but, due to time constraints this was not addressed.

At the heart of the simulation is a model of the pellet ballistics, which was modelled in a straightforward manner in an earlier project. The design of this implementation and associated intersection calculations were evaluated and considerably improved. Further to this the model was then approximated in an interesting manor to improve performance.

The first attempt simplified the representation of the pellets to improve speed. The intersection calculations were discarded due to limited accuracy. Bounding boxes and a more detailed model of a clay were used to simulate the collision detection quickly. The resulting performance of this model was still unsatisfactory collision detection varied when replayed.

The second attempt approximated both the pellet and the clay trajectories with spline curves. Several were used for each trajectory to reach satisfactory accuracy. The details of this are given in section 3.2.1. This renders the model sufficiently accurate for most situations however, in unusual circumstances greater accuracy can be achieved by using more splines. Accuracy in this project is given up to four splines but further work is required to determine the effect of using more splines and also to calculate bounds for the errors.

Limited assumptions are made about motion of the clay being linear over a negligible time period. These are made in the prediction and plane calculations so do not impact the collision detection. Linear interpolation leads to inaccuracies as discussed in section 3.1.6 and more complex programming however, the need for it has almost been eradicated, and its only use is in clay location
prediction as described in section 3.2.5.

The most significant advantage of the splines is the simplicity of adding further different targets. The motion modelling and intersection code is already wrapped up in several classes. The only information required is several points on the path and the dimensions of the new object. Better collision detection will be object dependent, but the existing calculations should be accurate enough for most applications. The other facilities such as motion prediction and the range resolution will also be available without any need for any extra programming.

The simplicity of the spline modelling has left the majority of the computation on the graphics card. With the exception of clay launch the processor is under utilised leaving scope for much more functionality to be added.

The simulation provides feedback about shots via an animated replay providing many different camera angles and accessories to aid with evaluation of a shot. A mechanism for giving a user a guide as to where they should be firing was also implemented in two ways, pellet and clay prediction.

The graphics rendering is flexible and powerful whilst still being efficient. Landscapes can be simply designed and added to the program. There is still large potential for development in this area but the range that can currently be produced is realistic.

6.1 Successes

The modelling, visualisation and user feedback components of this project have been particularly pleasing. The major successes in these areas can be summarised as follows:

- A visually appealing and realistic landscape was produced.
- The implementation of the original ballistics model and pellet/target modelling code has been improved substantially using spline approximations.
- Smooth and highly realistic clay flight was has been achieved using a simple model of drag and lift.
- A motion prediction system which visualises the required target lead has been developed; this feature is unique as far as can be determined and constitutes an important contribution of the project from the users’ perspective.
- The ability to replay fired shots from multiple camera viewpoints has been implemented; this is an important diagnostic tool which appears to be substantially more sophisticated than those offered by existing commercial simulators.
• The system has been delivered as a fully integrated and extremely robust stand-alone game, but can be easily extended to a full virtual reality simulation.

6.2 Further Work

Unfortunately there was no point in trying to integrate this software to the shotgun location from the sister project. Although very accurate, the other project unfortunately is unable to track the shotgun with a frequency of greater than around 9Hz. After some experimentation this was determined to be insufficient. Should the shotgun tracking become suitable it should be relatively easy to integrate with this project.

Due to time constraints the projection was not implemented in 3D. This should be simple to implement by projecting the range from two viewpoints. The interesting part will be evaluating how users interact with the 3D range. Some consideration should be given to the difficulties of projection. A large arc of view is required but also the ability to see the shotgun. This means that the user can not be totally immersed in the 3D environment but a window of projection may not be sufficient. The ideal solution would be a hemispherical screen.

The clay trajectory model is very simple. This could be improved using existing ‘frisbee’ flight models. Wind could be factored in to affect the flight of the clay. Also more energy is required to break a clay ‘side on’. This could be accounted for when the pellets hit the clay.

Barrel motion and length could be added to the ballistics model. This may require testing with several shotguns to see how spread is affected. Many misses are due to ‘loss of swing’, it may prove useful to record the barrel motion so this could be noted as part of the replay.

The graphics may be further developed to increase realism of the projected scene and take advantage of new hardware. This may range from simply designing better landscapes or implementing more complex grass and tree rendering techniques. This could also extend to animating features (with the wind for example).

A tool could be designed to convert photographs of a range into the format read by this software. Multiple photographs would be required and the geometry constructed from these. The foreground would require development to make it appear realistic. Techniques for this could be very complex, but is an interesting and challenging problem.

The closest a clay is shot at is around 5m. The splines become less accurate as a model around this distance. Further work is needed to evaluated fully the accuracy of the spline approximation for shorter distances and with higher resolution in terms of splines.
Bibliography

   http://www.cpsa.co.uk.

   http://www.claypigeonshooting.net.

[3] Clayshooting.co.uk.
   http://www.clayshooting.co.uk.

   http://www.delphigl.de/tuts/opengl/alphamask.html.

   http://www.dryfire.fsnet.co.uk.

   http://www.fltk.org/.

   http://www.dictionary.co.uk.

[8] Opencl graphics library web site.
   www.opengli.org.

   http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[10] Roam vs. vdpm.


    www.libsdl.org.


   http://www.lighthouse3d.com/opengl/billboarding/.


[26] NVIDIA. Tri-stripping library.  

