3D Games as Motivation in Fitts' Law Experiments

Julian C. A. Looser
Department of Computer Science
University of Canterbury
Christchurch, New Zealand
jcl55@cosc.canterbury.ac.nz

Supervisor: Andy Cockburn

November 8, 2002

Abstract

Traditional Fitts' Law evaluations in the field of human-computer interaction (HCI) often use artificial applications to collect data from participants. This report presents an investigation into the viability of using computer games as a basis for Fitts' Law experiments. Such a possibility is intriguing because it helps to overcome a fundamental problem in HCI research: evaluation participants are a scarce and precious resource. The entertainment provided by games may be sufficient incentive for people to be willing participants in evaluations.

We propose and implement a 3D game based mouse selection test, and report on an experiment which shows that the test can be reliably modelled by Fitts' Law. Subjective responses indicate that the test is both fun and motivating. We then suggest possible applications of the new test in the area of bimanual input research, and in various 3D navigation tasks.

Contents

Chapter 1

Introduction

This report presents the development of a Fitts' Law evaluation technique built upon a 3D computer game. The goal of this research is to determine whether 3D game environments can provide a platform for a variety of target acquisition experiments, including HCI evaluations.

Fitts' Law is a universally accepted law that relates human movement time to the distance and size of a target. The law has been applied extensively in HCI, where it is used to evaluate the usability of software interfaces and guide the development of hardware input devices. Fitts' Law is explained further in Section 2.1.

Computer game playing is perhaps the most popular use of computers today. Modern computer hardware is optimised to produce immersive interactive environments. In Section 2.2, we discuss the features of modern computer games that make them appealing foundations on which to build other applications. Techniques by which games can be modified for other purposes are explained, and several applications of this type are described.

Bimanual input research has uncovered potential benefits that can be gained by utilising our non-dominant hands in computer input. We intend to contribute to this body of research, outlined in Section 2.3, by producing an accurate learning curve which illustrates how performance of the non-dominant hand improves over time. This data will be of use to researchers wishing to know how much practice to give a novice bimanual input user before evaluating performance.

The motivation for this research stems from an interest in the three areas introduced above, and a long standing problem in HCI research: experimental participants are difficult to attract and retain. Often subjects must be enticed with rewards for their participation, such as money or course related credit. We believe that more innovative and interesting evaluation techniques could be employed to attract participants, and ensure a high level of motivation during the evaluation. We propose the use of a 3D game and report on an experiment to determine whether the game can be reliably used in Fitts' Law type evaluations.

We then intend to apply this technique in the area of bimanual input, where we hope to gain insight into the acquisition of skill with the non-dominant hand.

Chapter 2

Related Work

2.1 Fitts' Law

2.1.1 Fitts' Law Fundamentals

Fitts' Law (Fitts 1954) is a universally accepted law that relates human movement time to the size of and distance to a target. The general form of the Fitts' Law equation is:

$$MT = a + b \times IoD \tag{2.1}$$

where MT is the movement time in seconds, a and b are experimentally derived constants, and IoD is the index of difficulty in bits. The index of difficulty is a measure of how difficult a target is to select, formulated with respect to the target's size and distance. The index of difficulty is calculated as:

$$IoD = \log_2\left(\frac{A}{W} + 1\right) \tag{2.2}$$

where A is the amplitude, or distance to the target, and W is the width of the target. By experimenting with varying values for the IoD, it is possible to derive values for the constants a and b from Equation 2.1. Fitts' Law describes a linear relationship, where a gives a base value, and b gives the slope. In the task of selecting a target, a represents the cognitive and motor preparation time and b is a measure of hand-eye coordination.

Using b, the index of performance can be calculated. This value is also known as *bandwidth*, and is measured in bits per second.

$$IoP = \frac{1}{b} \tag{2.3}$$

The bandwidth in Fitts' Law is a measure of how much information can flow through a particular channel. This is analogous to bandwidth in the context of information theory. In this case the channel is the user's limb and pointing device.

2.1.2 Fitts' Law in HCI

Fitts' Law has been employed extensively in HCI. Its use has increased significantly over the past decade due to the rise of the personal computer and the desktop metaphor. Interfaces in which users directly manipulate on-screen objects are ubiquitous, so there is a need for a reliable model to predict human movement time (MacKenzie 1991). Within the field of human-computer interaction, Fitts' Law has been applied in several contexts. An overview of these is provided here.

Modeling Basic Tasks

Fitts' Law has been shown to be a good model of the task of dragging as well as pointing, although bandwidth is lower when dragging than when pointing (MacKenzie, Sellen & Buxton 1991).

Hinckley, Cutrell, Bathiche & Muss (2002) discovered that Fitts' Law is also an accurate model of scrolling performance. They compared several mice equipped with scroll-wheels with varying levels of acceleration and found reliable correlations between movement time and index of difficulty.

Input Device Comparison

Fitts' Law is commonly used to compare different input devices. Kabbash, MacKenzie & Buxton (1993) used Fitts' Law in an investigation into the performance of various input devices in the preferred and non-preferred hands. They evaluated a mouse, trackball and stylus and found that the stylus was superior for pointing, but inferior for dragging.

More recently, the International Standards Association (ISO) proposed a standard to guide the development of computer input devices. This standard is known as *ISO 9241 Ergonomic Requirements for Office Work with Visual Display Terminals*. Of particular interest is part 9 of the standard, which deals with non-keyboard input device requirements. Fitts' Law provides the base for much of the analysis of pointing performance in this part of the standard (Douglas, Kirkpatrick & MacKenzie 1999).

Mobile Phone Text Entry

Fitts' Law has been used to model the movement time of users entering text on a soft keyboard using a stylus (Soukoreff & MacKenzie 1995). Such keyboards are common on handheld devices such as PDAs and mobile phones.

Further attempts were made to examine whether the law could be applied to conventional text entry methods such as multi-press and T9. Again, it was found that Fitts' law was an accurate predictor of movement time, and equations were derived that provide expected performance values for these methods. (Silfverberg, MacKenzie & Korhonen 2000).

2.2 Computer Games

2.2.1 Background

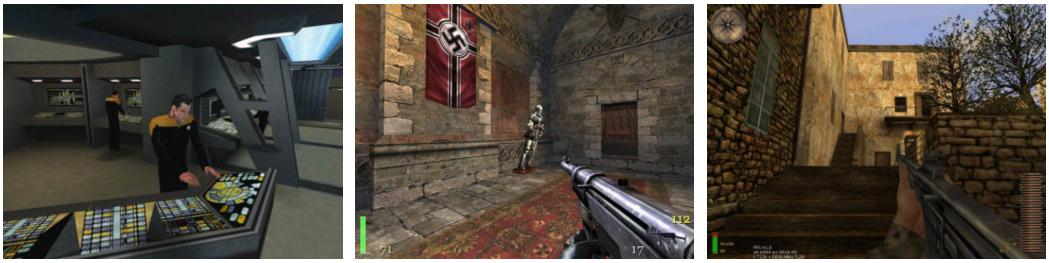
One of the most popular uses for modern computer systems is gaming. Sixty percent of all Americans play video games; with a reasonably even spread over gender and age groups (Interactive Digital Software Association 2001). In the same report it was stated that the major reasons people play games are because they are both fun and challenging.

One game genre of particular interest is that known as the "first-person shooter". This type of game immerses the player in a virtual-reality environment viewed from the first-person perspective. The first-person shooter originally appeared in the 1970s with a simple game called Spasim[1], but gained consumer attention in the 1990s, with the rise of powerful home computers and games such as Wolfenstein, Doom and Quake[2]. Such games are typically designed with a generic core known as the *engine*. The engine can be re-used again in different games by replacing the superficial features that define the look and feel of a particular game, such as the characters, textures, and sounds. The engine's responsibilities generally include aspects such as graphics rendering, the modeling physics and lighting, and networking. Figure 2.1 shows screenshots from three games, each with slightly different gameplay and each developed by a different company, but all based on the same engine.

As game engine complexity, in terms of visual quality and realism, has increased, so has the need for more powerful hardware to support it. This need has given rise to the 3D accelerator: a hardware device designed for the sole purpose of rendering three-dimensional graphics quickly. The first 3D accelerators were add-in PCI cards that worked in conjunction with existing 2D video cards. This combination provided

[1] Spasim was written by Jim Bowery. `http://www.geocities.com/jim_bowery/spasim.html`.

[2] Wolfenstein, the Doom series and the Quake series of games are trademarks of Id Software, Inc.



(a) Star Trek: Elite Force (b) Return to Castle Wolfenstein (c) Medal of Honor

Figure 2.1: Three games based on the Quake3 engine from id Software.

additional performance to those users - mostly game players - willing to pay for it. The initial Voodoo 3D accelerators from 3DFX[3] were based on this design, although later models from 3DFX were integrated solutions as discussed next.

Today, 3D acceleration is a standard feature on most video cards, so dedicated 3D accelerator cards are not required. The inclusion of high-performance 3D graphics capabilities in even the most budget consumer systems is a clear sign that computer gaming is a widespread phenomenon. Market leaders in computer graphics solutions include Nvidia[4] and ATI[5], who now produce graphics cards powered by sophisticated chips known as GPUs (Graphics Processing Units). These chips are analogous to the CPU within a computer but are tuned to generate graphics. They process in hardware operations traditionally handled by software, such as texturing and lighting (Lewis 2002). These consumer-priced graphics solutions are now being used in preference to traditional super-computers for some applications. For example, the U.S. Department of Energy uses standard PCs with consumer-level graphics cards for scientific visualisations (Rhyne 2002), and the same chips are used in the cockpit displays of F-22 fighter aircraft (Lewis 2002).

2.2.2 Features of Game Engines

Modern game engines are complicated pieces of software. Engine development often follows the same software engineering processes as other large projects. Modularity and extensibility are important considerations because an engine's success depends upon the ease with which third parties can use the engine to create their own games.

Such extensible game engines, and the abundance of high-performance 3D accelerators to support them, provide an environment waiting to be exploited. Many engine features designed to enhance the player's experience can be manipulated to serve other purposes. Any application, game or otherwise, based on a game engine would be able to make use of these features, some of which are outlined here.

Graphics Rendering

Modern game engines are rigourously optimised for the efficient rendering of three-dimensional virtual environments. Such optimisations include the culling of polygons not directly visible to the player and the automatic lowering of the level of detail of distant objects. Because both the engine and the graphics hardware are designed to work with standard APIs such as DirectX and OpenGL, the engine is able to make full use of the advanced features of the hardware.

Fast graphics rendering capabilities make game engines potentially useful tools in the field of scientific visualisation. A first-person shooter type game engine is particularly suited to visualisation as it promotes

[3] 3DFX Interactive has since been bought by Nvidia
[4] `http://www.nvidia.com`
[5] `http://www.ati.com`

exploration by immersing the player within an environment in which they can navigate freely.

Spatial Audio

Spatial audio has recently become a common feature in game engines. Subsequently, the sound-card market has become saturated with hardware to support this feature, such as the Sound-Blaster Live range from Creative[6]. APIs such as Aureal's A3D and Creative's EAX are used to create realistic environmental effects.

One area where spatial audio is useful is collaboration. Baldis (2001) investigated the effects of spatial audio in desktop conferencing and and found it significantly improved users' ability to remember the other participants in the conference, to focus on the active speaker and to generally comprehend what was happening. Game engines provide spatial audio support as well networking capabilities, making them ideal test-beds for studies such as Baldis', or in fact, foundations on which complete collaboration systems can be built.

Networking

One reason for the popularity of computer games is the ability to play with others over both public and private networks. This feature could be used to add collaboration options to a modified game engine. Engine developers often pay the networking component particular attention. Players in the game expect to play in real-time so the engine attempts to minimise latency. Some engines are optimised to such a degree that clients are synchronised to a higher precision than they can display to the user (Lewis & Jacobson 2002). Prediction schemes are often employed to compensate for poor network conditions. Although the latency itself cannot be reduced, its impact can be lessened by anticipating user input and extrapolating player paths (Pantel & Wolf 2002).

Support

Due to the popularity of games, it is common for numerous support tools to be written to facilitate the editing of a game. This includes tools to design new levels, characters and graphics. Most games acquire a following of players interested in the modification of the game. These tools and support groups are a valuable resource when attempting to modify a game.

2.2.3 Projects Based on Game Engines

There are numerous examples of systems that use a game as some sort of starting point. Some maintain the original game metaphor, while others only utilise the features of the underlying game engine for convenience.

Process Management

PSDoom uses the Doom engine to visualise processes within the Unix operating system (Chao 2001). The modification was possible after the Doom source code was released under the GPL[7] in 1997. In the classic Doom game, the player faced monsters in various maze-like levels. In PSDoom the monsters represent running processes on the local machine. The player is equipped with various weapons with which to fight the monsters. Wounding a monster lowers the priority of the process the monster represents. Killing the monster terminates the process. This behaviour is shown in Figure 2.2. Chao (2001) believed these activities were a plausible metaphor for process management under Unix. Whether this is a useful metaphor is debatable but it nevertheless demonstrates the range of modification possibilities that exist.

[6] `http://www.creative.com`
[7] The GNU General Public License, `http://www.gnu.org/copyleft/gpl.html`

Figure 2.2: Process management using PSDoom. Shooting a monster affects the priority of a running process.

Interactive Tours

Shiratuddin, Yaakub & Arif (2000) proposed the use of a 3D engine for walkthroughs and virtual tours. They compared a game-based approach to two other virtual reality technologies: VRML (Virtual Reality Modeling Language) and QuickTime VR. VRML is a simple, text-based format for describing virtual worlds. The authors stated that VRML viewers (typically web-browser plug-ins) suffer from poor graphics performance and unintuitive navigation modes. QuickTime VR uses a series of neighbouring photographs to create a panoramic view of a scene. This can result in high quality views, but the user cannot explore the scene in the same way as in a true 3D representation.

A real-world manifestation of the ideas proposed by Shiratuddin et al. (2000) is the VRND (Virtual Reality Notre Dame) project (VRND Project Team 1999). This project utilises the Unreal Tournament game engine to create a realistic virtual tour of the Notre Dame cathedral in Paris, France. A screenshot from the VRND project is shown in Figure 2.3.

Christoffel & Schmitt (2002) took a similar approach in their implementation of a virtual library using a modified version of the Quake 2 engine. The flexibility of that game engine was made obvious when the authors integrated the it with the library catalogue system. This link allowed the virtual library shelves to contain the correct books in the correct locations, so that the library could be browsed as shown in Figure 2.4.

CaveUT

CaveUT is a modification of Unreal Tournament that allows the game to be displayed in panoramic theaters (Jacobson & Hwang 2002). It provides flexible configuration options allowing it to be used in large auditorium settings as well as much smaller personal theaters, or *caves*. CaveUT has applications in education and virtual-reality research. The code for CaveUT is freely available online along with instructions for constructing a physical cave environment[8]. CaveUT modifies the game engine at a low level. Higher level modifications, such as the VRND project described above, could theoretically run simultaneously with CaveUT.

Software Visualisation

Knight & Munro (1998) proposed the use of a game engine for software visualisation. They generated maps representing call-graphs, which could be loaded into the game and explored. The multi-player capabilities of the engine allowed multiple users to explore the same visualisation simultaneously.

[8] Source code and instructions are available at `http://www.planetjeff.net/ut/CaveUT.html`

Figure 2.3: A screenshot from the virtual Notre Dame project. The cathedral is recreated almost entirely accurately within the Unreal engine.

2.2.4 Game Modification Techniques

Extending a game engine can be approached from two directions. Firstly, if full (or partial) access to the original source code is available then modification is a matter of redesign. Secondly, certain engines support scripting languages that can be used to add new functionality to the engine, or to override existing functionality.

Source Code Access

Access to the source code of a game engine is possible if the game is an open-source project, if the engine has been licensed from its authors, or if the authors have freely released the code.

Generally, licensing a game engine is prohibitively expensive. The popular Quake3 engine from id Software costs at least $250,000US. There are exceptions though, where the engine developers are keen to see their engine used more widely. For example, GarageGames, a support site for independent game developers, offer the Torque game engine, used in the popular game Tribes 2[9] for $100US.

Alternatively, there are many open-source game engines available. One of the most popular is Genesis3D[10] which is free for both non-commercial and commercial use (as per the conditions in the Genesis3D license).

The source code of some commercial game engines has also been released. This typically happens only when older games become no longer commercially valuable to their developers. Examples of such engines are Doom, Quake 1 and Quake 2.

Scripting Languages

An example of a game with a particularly elegant modification architecture is Unreal Tournament (UT) (Epic MegaGames 1998). UT provides an integrated scripting language called UnrealScript, which is reminiscent of Java in both operation and syntax. It runs safely in a virtual machine on the client, is object-oriented and supports garbage collection. Although most of the low-level engine code of UT is written in C++, the logic controlling the gameplay is written primarily in UnrealScript. A small part of this code is shown in Figure 2.5.

[9] Tribes 2 is a registered trademark of Sierra Entertainment, Inc.
[10] `http://www.genesis3d.com`

Figure 2.4: Browsing the books within the Karlsruhe virtual library. The visualisation communicates with the catalogue system so correct book information is shown to the user.

Code written in UnrealScript is compiled using the `ucc` compiler tool included with the game. Currently, the compiler and other support tools are provided for the Windows platform only. However, the compiled modifications are platform independent so they can also be used with the Macintosh and Linux versions of UT.

There are two main ways to modify UT using UnrealScript. The first is to create a *mutator*, the second is to create a new *game-type* (Reinhart 1999).

Mutators are small modifications that typically alter a single aspect of the game. For example, the low-gravity mutator reduces the effects of gravity, allowing players to jump higher and further. Multiple mutators can be applied in a single game session, so there is the possibility for unlimited variation. Mutator selection is shown in Figure 2.6(a). Writing mutators is simple because they only need to override the behaviour the developer wishes to manipulate.

Complicated modifications require the creation of an entirely new game-type. Game-types define the rules by which the game is played. For example, one game-type is "Capture The Flag". In a game of Capture The Flag, each of two teams attempt to steal a flag from the opposing team's base. Game-types cannot be combined in the same way as mutators, but mutators can be used within different game-types: Capture The Flag can be played with low-gravity for example. Writing a game-type gives the developer access to a lot more of the game's functionality.

Other game resources such as maps, textures and sounds can be created using the support tools included with UT, or with third-party products. The `unrealed` tool is provided with the game and serves as a map creator, code editor and resource browser. `unrealed` was used to develop most of the content in UT. Figure 2.6(b) shows the map creation mode of `unrealed`.

2.2.5 Ethical Considerations: Violence in Computer Games

There are ethical considerations that need to be addressed when modifying a game into a different application. Most first-person games are extremely violent. The character controlled by the player is frequently armed, with the goal of killing various opponents. This type of game has attracted a lot of attention in the media, and is often held accountable for violent acts by young people. However, it is important to distinguish between the violence visible in the game, and the technology within the game engine used to display it. In their report on the usefulness of game engines in scientific research, Lewis & Jacobson (2002) claim that despite their often violent nature, first-person games have historically provided the first and sometimes only platform on which new and advanced graphics techniques can be demonstrated. Such games are often the unrecognised drivers behind improvements throughout the entire computer graphics industry. It is

```
function bool SwitchToBestWeapon()
{
        local float rating;
        local int usealt, favalt;
        local inventory MyFav;

        if ( Inventory == None )
                return false;

        PendingWeapon = Inventory.RecommendWeapon(rating, usealt);                    10
        if ( PendingWeapon == None )
                return false;

        if ( (FavoriteWeapon != None) && (PendingWeapon.class != FavoriteWeapon) )
        {
                MyFav = FindInventoryType(FavoriteWeapon);
                if ( (MyFav != None) && (Weapon(MyFav).RateSelf(favalt) + 0.22 > PendingWeapon.RateSelf(usealt)) )
                {
                        usealt = favalt;
                        PendingWeapon = Weapon(MyFav);                                20
                }
        }
        if ( Weapon == None )
                ChangedWeapon();
        else if ( Weapon != PendingWeapon )
                Weapon.PutDown();

        return (usealt > 0);
}
```

Figure 2.5: A portion of the Unreal Tournament game code, written in UnrealScript.

usually possible to remove the violent content from a game, leaving the underlying engine and the useful features it provides intact.

2.3 Bimanual Input

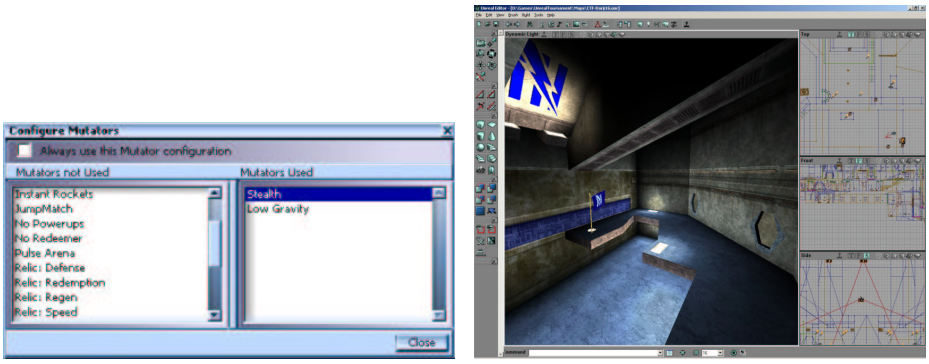
Bimanual input involves distributing the user's workload more evenly between the two hands. In most current point-and-click interfaces, it is typically only the dominant hand that is used to manipulate on-screen objects. The dominant hand is overworked, while the non-dominant hand lies dormant, only being engaged while the user is typing.

2.3.1 Bimanual Input Fundamentals

Modeling the Two Hands

In order to successfully implement a bimanual interface, knowledge of how our hands operate at a fundamental level is required. The most popular model for describing the relationship between the two hands is the kinematic chain (Guiard 1987). In this model, the non-dominant hand forms the base of the chain while the dominant hand forms the tip. The non-dominant hand provides the rough frame in which the dominant hand can operate with precision. Guiard used handwriting as an example: although the dominant hand is responsible for actually marking the paper, the non-dominant hand is used to move and orient the paper into a workable position. Figure 2.7 illustrates Guiard's example. Handwriting is significantly more difficult when the support provided by the non-dominant hand is unavailable.

Using the kinematic chain as a model, we expect different computer input characteristics from each hand: the dominant hand being swift and precise and the non-dominant hand being slower and less accurate. This was shown to be the case in a comparison of various input devices across both hands (Kabbash



(a) Selecting a mutator configuration in Unreal Tournament.

(b) Creating a map in the Unrealed map editor.

Figure 2.6: Modifying Unreal Tournament

et al. 1993). It was found that for rough pointing the hands perform comparably, but that the non-dominant hand is less capable for precise pointing. For example, tool selection from a palette requires precise pointing due to the small target sizes, making it a task suitable for the dominant hand. Conversely, scrolling is suited to the non-dominant hand because it does not require complete accuracy and is an act of setting the frame of reference (MacKenzie & Guiard 2001).

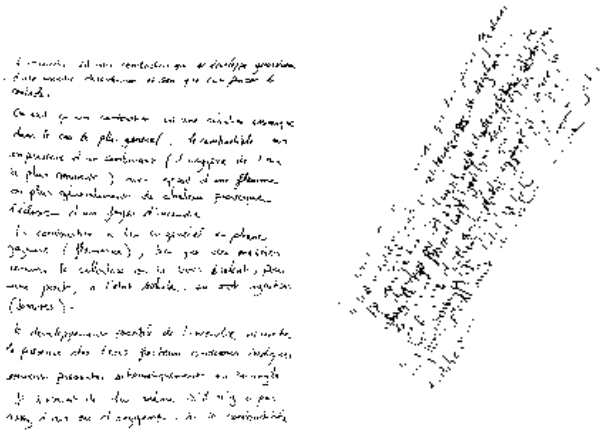
Task Suitability

It is clear that there are tasks that the non-dominant hand could carry out. The greatest benefits are gained, however, when both hands work in parallel. Parallelism is possible when the user is presented with tasks that are *integrated*. An integrated task is one that involves multiple distinct actions but is seen conceptually as a single task. The act of hitting a nail with a hammer is an integrated task, because although it is composed of both the striking action with the hammer and the steadying of the nail, it is seen as a single, fluid motion. Tasks that are not integrated risk dividing the user's attention. If the user is constantly switching focus between two distinct tasks than the parallelism will be lost. It is extremely difficult to apply both hands to two unrelated tasks.

Attempts to construct bimanual systems that conform to the above observations, particularly regarding Guiard's kinematic chain, have shown greater success than those that ignore or oppose them (Leganchuk, Zhai & Buxton 1998).

Hardware Support

Bimanual input research requires hardware and software that can handle two input devices simultaneously. In 1998, Leganchuk et al. (1998) claimed the cost of hardware to facilitate bimanual input was a restricting factor. However, a simple solution consisting of two mice on a standard desktop workstation is now inexpensive and easy to implement. This is largely due to the abundance of USB (Universal Serial Bus) devices and the widespread support they have received. Using the DirectX API within the Windows 98 operating system it is possible to simultaneously sense input from any number of mice (or other USB input devices). This capability appears to be relatively unknown, but was proven to be both available and usable by means of a simple prototype application we developed. It also appears to be the foundation on which the multiple input device support in the collaboration software developed by Hourcade & Bederson (1999) is built.



(a) The handwritten document

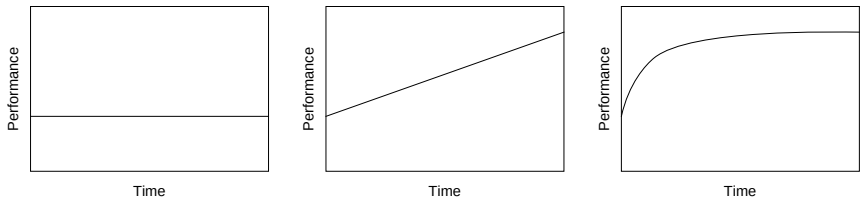
(b) Marks made on the desk, captured using carbon paper

Figure 2.7: The dominant hand operates the pen within a small area, while the non-dominant hand moves the paper, and orients it into a comfortable writing position.

2.3.2 Recognised Problem in Bimanual Input Research

One aspect of bimanual-input research that has received little attention in the literature is the rate at which non-dominant hand performance improves. Most computer users have only ever used their non-dominant hand when typing. Typing is considered a discrete task; most users have little or no experience using their non-dominant hand for continuous computer input, such as selection and dragging tasks with the mouse. A question that needs to be answered is: what does the learning curve associated with the non-dominant hand look like? Such information would be useful to researchers when evaluating bimanual interfaces. Knowing when users have progressed beyond an initial learning stage would allow researchers to investigate the users' typical performance. Beginning to evaluate performance too early would result in the collection of data representing the learning phase, which is unlikely to accurately reflect average user performance.

We anticipate one of three general learning curve shapes, as shown in Figure 2.8.



(a) No learning effects

(b) Increasing performance, no plateau

(c) Increasing performance, plateau

Figure 2.8: Possible learning curve shapes.

Firstly, there may be no learning effects. The users' performance may remain constant regardless of the amount of practice received. This case is highly unexpected, as it assumes not only that the non-dominant

hand suffers from a lack of practice, but also that it is generally incapable of improvement through practice. Secondly, the performance may increase linearly with practice. In this case, there may be a long initial learning phase before some sort of plateau is reached. Thirdly, performance may increase quickly initially, and the improvement may slow and reach a plateau. In this case, the initial learning phase may be short. This is the expected case.

Chapter 3

Experiment

3.1 Aim

The aim of this experiment is to determine whether a 3D game can be used as a substitute for a traditional Fitts' Law test. It is hoped that the entertaining nature of a game will serve to attract people to participate in an experiment, and perhaps also not be so conscious of the fact that they are in fact being tested at the time.

The acquisition of experiment participants is a long-standing problem for HCI researchers, but a game-based Fitts' Law test must be proved scientifically valid to be of any use. The experiment aims to show that there is no significant difference between the mean bandwidths in the game-based test and the traditional test.

3.2 Method

Two interfaces were implemented, one to evaluate performance in a traditional desktop environment, and another to evaluate performance within a 3D computer game.

The independent variable in this experiment was IoD, the Index of Difficulty. This value depends on the distance to a target and the size of the target, and is calculated using Equation 2.2. The dependent variable was Movement Time, the time taken for the participant to select the next target following a successful selection.

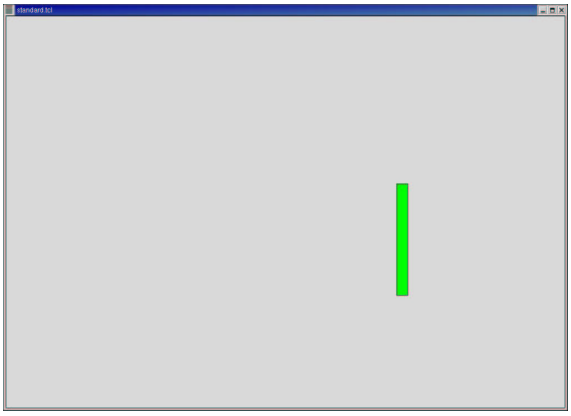
3.2.1 Apparatus

Both interfaces were evaluated using the same monitor and mouse. The monitor was a 19" Compaq S920 and the mouse was a standard 3-button, PS2 model (Logitech M-S35). No keyboard input was required.

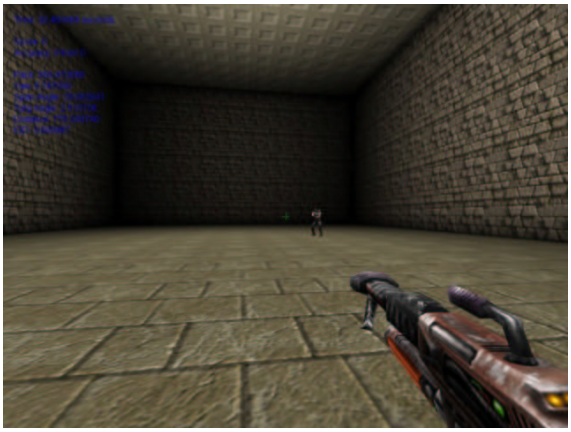
The traditional interface was run under Redhat Linux at a desktop resolution of 1600x1200 pixels. The game-based interface was run under Microsoft Windows 2000 at a resolution of 1024x768 pixels. The traditional interface ran as a windowed application, while the game-based interface ran in full-screen mode. The game was run under Windows because the Linux version is less stable and random crashes would have been unacceptable under experiment conditions.

3.2.2 Participants

Participants for this experiment were recruited from the Computer Science department of the University of Canterbury. Eleven subjects (9 males and 2 females) participated in the experiment. All had a high level of experience with computers and claimed to be familiar with computer games. Every participant was right handed except for one, and all used their dominant hand for mouse control.



(a) The traditional Fitts' Law test. The target reappears at random horizontal locations when selected.



(b) The Unreal Tournament Fitts' Law test. Selection involves shooting the enemy targets.

Figure 3.1: The two implemented Fitts' Law tests.

3.2.3 Interface Designs

Traditional Interface

The traditional test was implemented in Tcl/Tk. It consisted of a large blank area within which a green vertical bar appeared. The window measured 1000x700 pixels, and the green bar measured 20x200 pixels. The user's task was to click on the green bar, which would then disappear and reappear in a new location within the window. The bar always appeared at the same vertical position — only the horizontal position was changed. The horizontal positions were selected randomly from a set of seven positions distributed at intervals across the workspace. The standard interface is shown in Figure 3.1(a). The user was required to make 100 selections. Performance data was recorded in log files generated automatically by the program.

Unreal Tournament Based Interface

The game-based interface was implemented using UnrealScript, the scripting-language integrated into the game Unreal Tournament. Several different game-types exist by default in UT; each defines a set of rules

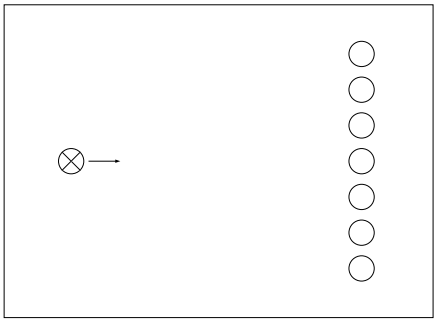


Figure 3.2: The positions of the player (indicated by the cross) and the target start points, viewed from above.

for a different twist on the basic game. A new game-type was created to define the rules for the experiment. The source code for the modification is included in Section 6. The targets in the game-based interface were computer-controlled players, or *bots*, and the selection technique involved shooting the bots with a futuristic blaster weapon. This behaviour is shown in Figure 3.1(b).

Only one bot was in play at any one time and was analogous to the green bar in the traditional interface. The bot would appear at a predefined position and would not move. The goal of the player was to shoot the bot as quickly as possible following its appearance. Once this was done, the bot would reappear at a new location, selected at random from a predefined set. The player would reacquire the target, and the process would repeat.

The targets had a cylindrical collision volume, meaning that although they appeared to be human-shaped, any shot striking the volume would result in a hit. The cylindrical nature of the volume also meant that the bots presented a rectangular target from all directions within the same horizontal plane.

One hundred selections were made, as in the traditional interface. The set of starting positions were defined in a map specifically created in `unrealed` for this experiment. The layout of the map is shown in Figure 3.2.

The player could not move from their original location, but could look around the environment freely using the mouse. Other aspects of the game were kept as close to their original nature as possible. For example, the sound in the game was left enabled because it was one of the most entertaining aspects, and was important in creating an immersive environment.

3.2.4 Design Considerations

One-Dimensional Movement

Although it would have been possible to present targets on a 2D plane or even within a 3D space, both interfaces presented targets distributed along a single dimension. A one-dimensional selection task was chosen because of its simplicity. This research is a first step towards using a game for evaluation purposes, and it was considered wise to keep things simple to begin with.

Amplitude and Width Measurements

Amplitude (distance to target) and width of target measurements are required for Fitts' Law calculations. Pixel measurements were used in the traditional test, however, the 3D nature of the game-based test made pixel measurements difficult. Specifically, the distance in pixels between two locations in the 3D world becomes meaningless. The solution to this problem was to use angular measurements instead. The amplitude was measured as the angle through which the user must rotate before they are pointing at the target. The width was measured as the portion of the user's field of view that the target occupied. Bearing in mind that targets are presented along a single dimension, the player's rotation refers only to the player's *yaw*. A field of view (FOV) of 90° was used throughout the experiment. This is the default FOV in UT.

As an example of angular measurements, consider the layout shown in Figure 3.3. The span angle, or angle through which the player must rotate in order to be facing the new target, is found by taking the dot product of the vectors between the player's location and the locations of the two targets (the target just selected and the target just appeared).

$$\begin{aligned} \mathbf{AB} \cdot \mathbf{AC} &= |\mathbf{AB}|\ |\mathbf{AC}|\ \cos\ \theta && (3.1) \\ \theta &= \cos^{-1}\ \frac{\mathbf{AB} \cdot \mathbf{AC}}{|\mathbf{AB}|\ |\mathbf{AC}|} && (3.2) \\ \theta &= \cos^{-1}\ \frac{(200 \times 200) + (-75 \times 50)}{213 \times 206} \\ \theta &= \cos^{-1}\ 0.826 \\ \theta &= 34.294^{\circ} \end{aligned}$$

The target angle, that is, the angle in the player's field of view that the target occupies, was simpler to calculate. The distance between the player and the target was known, and the radius of the target was a constant within the game (17 Unreal Tournament units). The target angle can be calculated using simple trigonometry, as shown here, beginning with Equation 3.4.

$$\begin{aligned} \tan\ \phi &= \frac{17}{206} && (3.3) \\ \phi &= \tan^{-1}\ \frac{17}{206} \\ \phi &= 4.718^{\circ} \\ targetangle &= 4.718 \times 2 \\ targetangle &= 9.436^{\circ} \end{aligned}$$

The span angle and target angle can now be substituted into the Fitts' Law equation to derive an index of difficulty.

$$\begin{aligned} IoD &= \log_2 \left(\frac{A}{W} + 1 \right) && (3.4) \\ IoD &= \log_2 \left(\frac{34.294}{9.436} + 1 \right) \\ IoD &= 2.862 bits \end{aligned}$$

So a selection requiring a sweeping movement of 34.294° in the UT test represents a task with an index of difficulty of 2.862 bits.

The discussed behaviour was not difficult to implement because the vector based operations such as normalisation, dot-product and vector magnitude are directly supported by UnrealScript.

Control-Display Gain

Control-display gain is a measure of how much movement the control device must make to produce one unit of movement on the display device. There is typically some sort of linear relationship between these two values, but this need not be the case (MacKenzie 1991). It has been suggested that Fitts' Law experiments are unaffected by different control-display ratios, but in order to remove a possible confounding factor, a 1:1 control-display ratio was used in both interfaces. This was achieved by measuring and aligning physical mouse movement and on-screen movement. Within the Linux environment a 1:1 mapping was established using the `xset` command. In UT, a value of 0.2 was used for the in-game sensitivity option, and the options for mouse input smoothing and DirectInput were disabled.

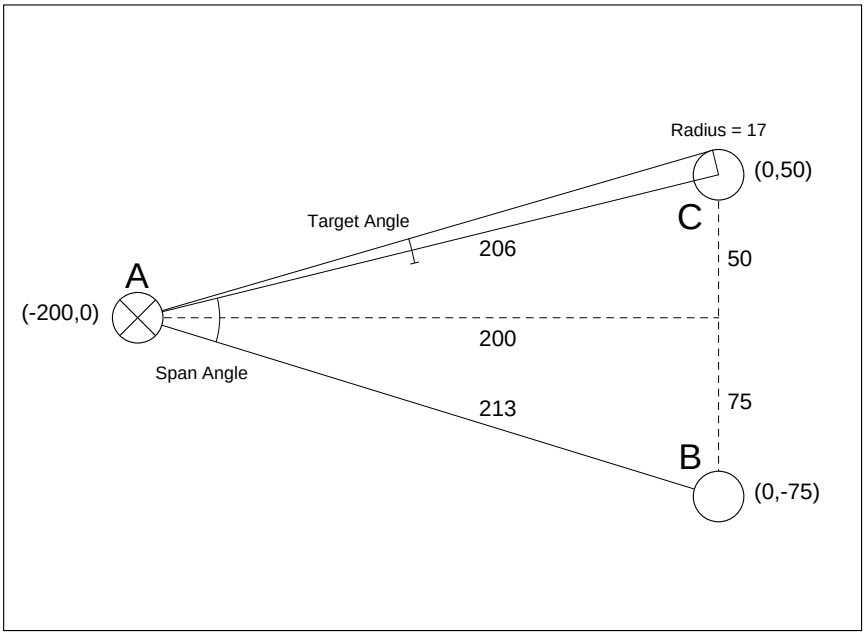


Figure 3.3: Example of angular measurement calculations. The player is located at position A. The target at position B has just been eliminated, and the new target has just appeared at position C.

Dealing with Misses

A clear difference between the two tests was recognised early. The reload time on the weapon used in the game to shoot the targets was approximately 850 milliseconds, while the time required to re-click the mouse button was only 300 milliseconds. This difference meant that the game-based test introduced half a second of dead time after each shot, in which time the player was able to press the mouse button without a shot being fired (see Figure 3.4). Normally, this was not a problem, because the delay time would elapse before the player tracked to the next target. The one situation where this dead time could potentially cause problems was when the player missed a target and needed to select again without tracking. In such cases, the player would attempt to fire, but could not, resulting in an incorrect measure of performance, as well as user frustration. Two solutions to this problem were considered. The weapon could either be modified to remove the reload delay, or instances where the player missed the target could be ignored. The second approach was chosen because it allowed the game to remain closer to its original nature.

Mouse Inversion

A common option when navigating within 3D virtual worlds is to invert the vertical behaviour of the input device. The uninverted setting leaves navigation in the standard state where *up is up* and *down is down*. Most users seem to find this setting the most intuitive. The inverted state makes navigation more like the task of controlling an airplane, where *pulling back on the stick* results in an upward movement, even though the mouse is drawn downwards. Subjects were permitted to choose whether or not the mouse was inverted in the UT interface. All but one subject opted for the non-inverted setting. The option was given to the subjects because initial testing showed that using a non-preferred setting resulted in extremely poor performance.

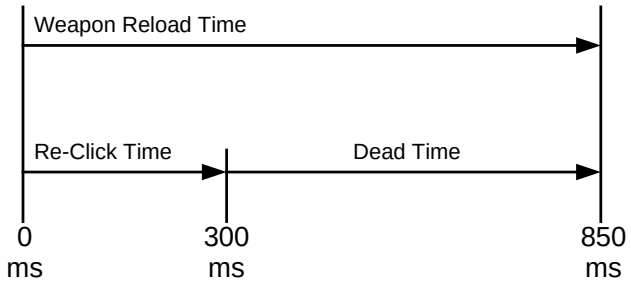


Figure 3.4: Dead time introduced by the reload time on the weapon.

3.3 Procedure

The experiment followed a within-subjects design, meaning that all participants used both interfaces. Counterbalancing was used so that half the participants used the traditional interface first, and the other half started with the UT test.

Each interface was explained to the participants and they were given the opportunity to practice before being tested. All subjects signed consent forms indicating their willingness to participate and their acknowledgement that the experiment involved graphic violence. Participants were free to leave at any time, for any reason whatsoever.

Each subject made 100 target selections in each interface, for a total of 200 target selections per subject. The interface evaluations took at most five minutes per subject. Afterwards, participants filled in a short questionnaire in which subjective information was collected.

3.4 Results

Data Collection

Results were collected through log files generated by the two interfaces. The movement time and index of difficulty values for each participant, in each interface, were recorded.

Adjustment of Data

The bandwidth result for Subject 10 in the UT test was originally 19.936 bits. This was significantly higher than values for the other subjects. Subject 10 repeated the experiment and produced a bandwidth of 6.0827 bits, which is closer to the expected value. The original value has been omitted and is considered the result of a logging error.

Quantitative Analysis

Regression was used to generate the Fitts' Law coefficients a and b for each interface. An extremely strong correlation was found to exist between movement time and index of difficulty in both interfaces. The R^2 values were 0.93 and 0.94 in the traditional and UT interfaces respectively. Any correlation higher than 0.8 is considered strong and reliable. Therefore, these findings indicate that Fitts' Law is a highly accurate predictor of movement time in both interfaces.

The regression coefficients a and b were recorded from the regression results. a is the motor and cognitive preparation time, and b is a measure of hand-eye coordination. The reciprocal of b gives the index of performance, or bandwidth. The bandwidth is the rate at which information flows from the user. A high bandwidth means that the user can complete the task quickly.

A significant difference between the mean motor and cognitive preparation time in the two interfaces was uncovered. (Traditional mean = 0.22 seconds, σ = 0.12, UT mean = 0.51, σ = 0.26, $p < 0.05$).

The hand-eye coordination values for the two interfaces were extremely close. In the traditional test *b* was 0.18, giving a bandwidth of 5.51, and in the UT test, *b* was 0.19, giving a bandwidth of 5.36 bps. These values are consistent with other Fitts' Law experiments such as that done by MacKenzie (1991), which produced a bandwidth of 5.6 bps for the mouse.

The data collected for each interface is shown in Tables 3.1 and 3.2. The significant correlations and similar bandwidths within the data are visualised in the graph in Figure 3.5.

IoD (bits)	Movement Time (seconds)
1.81	0.58
2.58	0.68
3.09	0.77
3.46	0.80
3.75	0.79
4.00	0.96
4.21	0.91
4.39	0.94
4.55	1.07
4.70	1.02
5.07	1.15
5.36	1.24

Table 3.1: Movement times for varying levels of task difficulty in the Traditional interface.

IoD (bits)	Movement Time (seconds)
2.37	0.95
3.22	1.03
3.69	1.11
4.10	1.21
4.44	1.28
4.73	1.41

Table 3.2: Movement times for varying levels of task difficulty in the UT interface.

Qualitative Analysis

Interface Preference

Subjects were asked which interface they preferred using. All subjects said they enjoyed using the UT interface the most. Subjects rated their enjoyment level for each interface on a Likert scale ranging from 1 for "not enjoyable at all" to 5 for "very enjoyable". There was a significant difference between the responses: UT mean = 4.34, σ = 0.54, traditional mean = 1.91, σ = 0.51 (Wilcoxon Signed Ranks Test, $p < 0.05$).

All subjects stated that they would be enthusiastic to participate in further evaluations involving the UT interface. Four of the eleven said they would be eager to participate in evaluations involving either interface. No participants would rather use the traditional interface in favour of the UT interface.

Violent Content

Subjects were asked for their opinion of the violent content visible in UT. Their responses appear in Table 3.3. Subjects were permitted to select all options that they considered applicable. This accounts for the

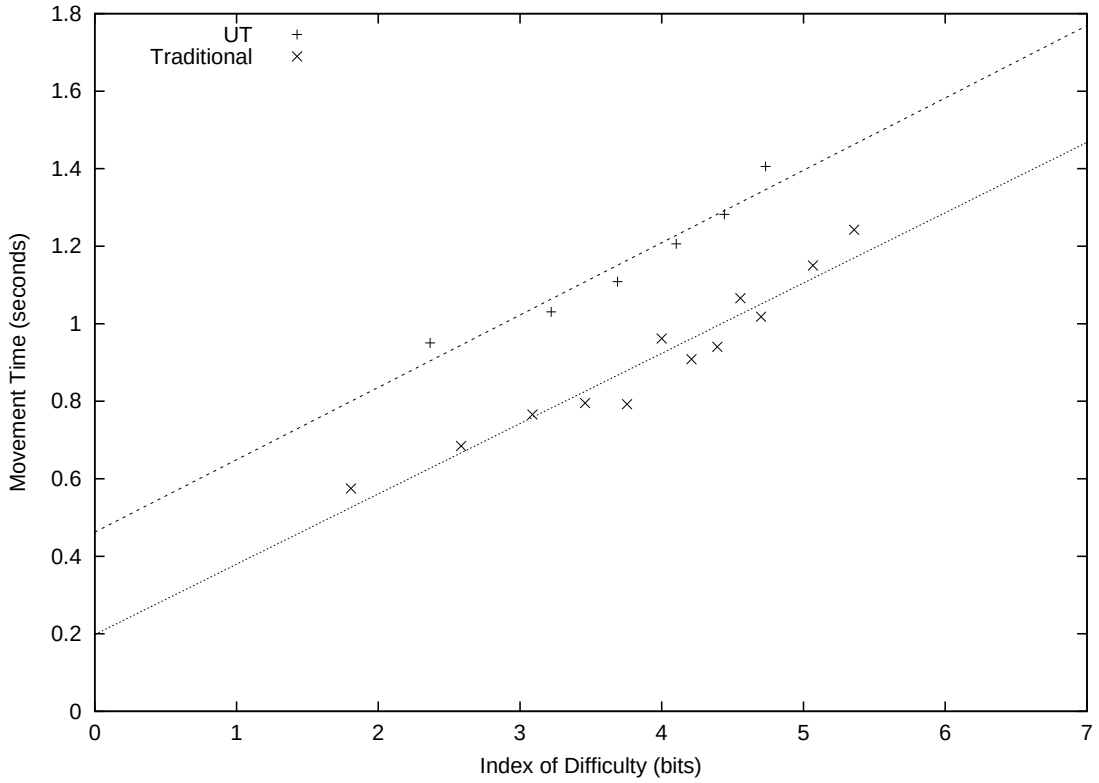


Figure 3.5: Graph of movement time against index of difficulty across the two interfaces.

Violent content was...	Agreement
Disturbing	18%
Irrelevant	55%
Entertaining	72%
Motivating	46%

Table 3.3: subject opinions concerning the violent nature of the game-based interface.

percentages summing higher than 100%.

Interestingly, the subjects that claimed the test was disturbing also said that it was entertaining.

3.5 Discussion

Interface Comparison

The results show that both interfaces are modelled extremely accurately by Fitts' Law. Furthermore, the bandwidth values for both interfaces are very close. This means that user hand-eye coordination was the same between interfaces.

There was a significant difference of around 250ms in the motor and cognitive preparation time between the two interfaces, with the traditional test requiring less time than the UT test (198ms compared to 462ms). One possible explanation for this difference is that the UT test produced some short graphical effects after each target was eliminated. This brief display may cause a momentary distraction resulting in a delay before the subject begins the task of acquiring the next target. The different interface type may also have played a

role. The traditional interface involves the movement of a cursor within an unmoving environment, while the UT interface involves a stationary crosshair within a moving environment. It is plausible that movement in the more immersive environment demands more thought and planning.

Overall, the results show that the experiment was a success. It is clear that Fitts' Law accurately predicts movement time in the UT interface. The UT interface can confidently be used in the evaluation of target acquisition experiments.

Subject Comments

Subjects were given the opportunity to comment on the experiment, and some interesting points were made.

One subject commented that the lighting and colours in the 3D environment made the target difficult to see. This problem would be relatively simple to resolve by editting the map.

Several subjects were frustrated by the 1:1 control/display gain on the mouse, mainly because they were used to playing the game with a much higher mouse sensitivity level.

Overall, subject responses were positive. The game-based interface was described by several subjects as "cool" and enhancements such as selections in 2D and different targets (all the targets currently look the same) were suggested. One subject specifically commented that the game-based interface was motivating, and that at moments they forgot they were being evaluated, and simply played the game.

Chapter 4

Further Work

This report has shown that a pointing task built within a computer game can be reliably modelled by Fitts' Law. The current implementation is quite simple, but demonstrates that such evaluation techniques are possible. These findings open up numerous further research opportunities. For example, the game-based interface could be extended to support the evaluation of more advanced tasks such as 2D movement, dragging and tracking.

The current implementation constrains the player's freedom within the virtual world to a high degree. It would be interesting to research whether more freedom can be given to the player, while at the same time collecting useful, and reliable, performance information. Ideally, data could be collected from an implementation in which the user was unaware they were under evaluation. For example, if the data collection process was integrated into a game such as Unreal Tournament, then the recruitment of experiment participants would become irrelevant. The worldwide community of game-players would become those participants. Such an idea may not be as unrealistic as it sounds — the infrastructure to support such a system is already in place. The NGStats[1] logging system is a component built into Unreal Tournament that collects game-based performance data so that players be ranked on a global ladder. Presently the data recorded is only related to the game, such as weapons used and scores achieved.

4.1 Application of Game-Based Interface

4.1.1 Bimanual Input

The game-based interface evaluated in this report can be used in the investigation of the nature of the non-dominant hand. There is a general lack of detail in the literature surrounding the learning rate of the non-dominant hand. We propose an experiment in which the game-based interface described in this report is used to collect performance data throughout a longitudinal study. Using this data it would be possible to model a learning curve. If the learning curve for the non-dominant hand use is consistent with learning-curves for other tasks, then we would expect there to be an initial phase in which skill was rapidly gained, followed by a phase of gradually less and less improvement. Of importance is the initial phase. Evaluating a bimanual interface using subjects who are still rapidly acquiring skill will lead to unreliable results due to strong learning effects. We intend to discover how much practice is required before the initial phase is passed.

4.1.2 3D Navigation Techniques

The emergence of virtual reality as a popular visualisation technique has prompted a large amount of research into navigation methods in 3D worlds. There is current research into the structure of navigation systems, such as optimal field-of-view angles (Czerwinski, Tan & Robertson 2002), and also into interfaces to facilitate efficient and effective navigation, such as speed-coupled flying (Tan, Robertson &

[1] `http://ut.ngworldstats.com`

Czerwinski 2001). Such interfaces are evaluated using target acquisition tasks, based on Fitts' Law. It is therefore important to be confident that Fitts' Law can be successfully applied in three dimensions. The experiment described in this report involved a simple one-dimensional selection task, within a complete 3D environment. This configuration was shown conclusively to be modelled by Fitts' Law. Further work on the UT interface is required to determine whether the same strong correlation between movement time and index of difficulty exists when targets are presented in two and three dimensions.

Chapter 5

Conclusion

This report has succeeded in its aim of showing that pointing tasks in the 3D game environment are accurately modelled by Fitts' Law. The game-based interface can be confidently used to perform Fitts' Law evaluations in preference to more traditional tests. The game provides a more attractive and motivating environment, making the recruitment of subjects easier.

In the course of this research, the nature of modern 3D game engines was examined. It was found that the features found in these engines are some of the most highly optimised examples of graphics rendering, spatial audio and client-server networking available. Furthermore, modification of these engines is straightforward due to integrated scripting languages which are well-supported by the game-playing community. This research has illustrated that 3D game engines can be modified into useful tools.

Acknowledgments

I would like to thank my supervisor, Andy Cockburn, for his support and enthusiasm. Thanks also to the Computer Science honours students of 2002 for their participation in experiments, as well as their friendliness and constant encouragement.

Bibliography

Baldis, J. J. (2001), Effects of spatial audio on memory, comprehension, and preference during desktop conferences, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 166–173.

Chao, D. (2001), Doom as an interface for process management, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 152–157.

Christoffel, M. & Schmitt, B. (2002), Accessing libraries as easy as a game, Technical report, Institute for Program Structures and Data Organization.

Czerwinski, M., Tan, D. S. & Robertson, G. G. (2002), Women take a wider view, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 195–202.

Douglas, S. A., Kirkpatrick, A. E. & MacKenzie, I. S. (1999), Testing pointing device performance and user assessment with the iso 9241, part 9 standard, *in* 'Proceeding of the CHI 99 conference on Human factors in computing systems : the CHI is the limit', ACM Press, pp. 215–222.

Epic MegaGames (1998), 'Unreal Tournament'. `www.unrealtournament.com`.

Fitts, P. (1954), 'The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement.', **47**, 381–391.

Guiard, Y. (1987), 'Asymmetric Division of Labor in Human Skilled Bimanual Action: The Kinematic Chain as a Model', *Journal of Motor Behavior* **19**(4), 486–517.

Hinckley, K., Cutrell, E., Bathiche, S. & Muss, T. (2002), Quantitative analysis of scrolling techniques, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 65–72.

Hourcade, J. P. & Bederson, B. B. (1999), Architecture and Implementation of a Java Package for Multiple Input Devices (MID), Technical report, Human-Computer Interaction Lab, Computer Science Department, University of Maryland.

Interactive Digital Software Association (2001), 'State of the Industry Report, 2000-2001'. `http://www.idsa.com/releases/SOTI2001.pdf`.

Jacobson, J. & Hwang, Z. (2002), 'Unreal tournament for immersive interactive theater', *Communications of the ACM* **45**(1), 39–42.

Kabbash, P., MacKenzie, I. S. & Buxton, W. (1993), Human Performance Using Computer Input Devices in the Preferred and Non-Preferred Hands, *in* 'Proceedings of INTERCHI'93 Conference on Human Factors in Computing Systems Amsterdam, April 24–29', pp. 474–481.

Knight, C. & Munro, M. (1998), Using an existing game engine to facilitate multi-user software visualisation, Technical report, Visualisation Research Group, Department of Computer Science, University of Durham.

Leganchuk, A., Zhai, S. & Buxton, W. (1998), 'Manual and Cognitive Benefits of Two-Handed Input: An Experimantal Study', *ACM Transactions on Computer-Human Interaction* **5**(4), 326–359.

Lewis, M. (2002), 'The new cards', *Communications of the ACM* **45**(1), 30–31.

Lewis, M. & Jacobson, J. (2002), 'Games Engines in Scientific Research', *Communications of the ACM* **45**(1), 27–31.

MacKenzie, I. S. (1991), Fitts' Law as a Performance Model in Human-Computer Interaction, PhD thesis, University of Toronto, Toronto, Ontario, Canada.

MacKenzie, I. S. & Guiard, Y. (2001), The two-handed desktop interface: Are we there yet?, *in* 'Proceedings of CHI'2001 Conference on Human Factors in Computing Systems Seattle, Washington, March 31–April 6', pp. 351–352.

MacKenzie, I. S., Sellen, A. & Buxton, W. (1991), A Comparison of Input Devices in Elemental Pointing and Dragging Tasks, *in* 'Proceedings of CHI'91 Conference on Human Factors in Computing Systems New Orleans, May', pp. 161–166.

Pantel, L. & Wolf, L. C. (2002), On the suitability of dead reckoning schemes for games, *in* 'Proceedings of the first workshop on Network and system support for games', ACM Press, pp. 79–84.

Reinhart, B. (1999), 'Mod Authoring for Unreal Tournament'. `http://unreal.epicgames.com/UTMods.html`.

Rhyne, T.-M. (2002), 'Computer games and scientific visualization', *Communications of the ACM* **45**(7), 40–44.

Shiratuddin, M., Yaakub, A. & Arif, A. (2000), Games engine in real world virtual reality application, Technical report, Virtual Reality Innovation Centre, Northern University of Malaysia.

Silfverberg, M., MacKenzie, I. S. & Korhonen, P. (2000), Predicting text entry speed on mobile phones, *in* 'Proceedings of the CHI 2000 conference on Human factors in computing systems', ACM Press, pp. 9–16.

Soukoreff, R. & MacKenzie, I. (1995), 'Theoretical Upper and Lower Bounds on Typing Speeds Using a Stylus and Soft Keyboard', *Behaviour and Information Technology* **14**, 370–379.

Tan, D. S., Robertson, G. G. & Czerwinski, M. (2001), Exploring 3d navigation: combining speed-coupled flying with orbiting, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 418–425.

VRND Project Team (1999), 'VRND Project'. `http://www.vrndproject.com/`.

Chapter 6

Appendix A

Game Modification Source Code

The following code is the main modification made to the game Unreal Tournament in order to convert it into a Fitts' Law experiment test-bed. Three classes were written to achieve this goal. The first is the main controller in contains the bulk of the evaluation code. The second class overrides the default heads-up-display in order to remove distracting on-screen text and animations. The third class defines the weapon used in the game.

Controller

```
class Experiment expands DeathMatchPlus;

/* Experiment
 *
 * This class defines a new game type based on DeathMatch. The
 * new game type states the rules that will constrain this
 * experiment.
 */

const EXPERIMENT_LENGTH = 600.0; // Experiment length in seconds
const MOUSE_RECORD_INTERVAL = 0.02; // Seconds between mouse position records
const HIT_RADIUS = 17;
const MAX_KILLS = 100;

// Useful constants for logging
const SPACE = " ";
const DELIMITER = "|";
const LOG_TAG = "NONDOM";
const KILL_MESSAGE = "KILL";
const MOUSE_MESSAGE = "MOUSE";
const POS_MESSAGE = "POSITION";
const BOT_POS_MESSAGE = "BOT_POSITION";
const FIRE_MESSAGE = "FIRE";
const END_MESSAGE = "END_EXPERIMENT";

// Useful constants for unit conversions
const RadianToDegree = 57.2957795131;
const DegreeToRadian = 0.01745329252;
const RadianToURot = 10430.3783505;
const URotToRadian = 0.000095873799;

var float currentTime, lastMouseUpdateTime;
var bool experimentRunning;
var int kills, shotsFired, killsCounted;
var Bot targetBot;
var PlayerPawn player;
```

```
var PlayerStart start, botStartPoints[16];
var int spCount, lastStart;
var float IOD, spanAngle, targAngle, dist;
var float lastIODRecordTime, spawnTime;
var bool fired, justMissed;
var NavigationPoint lastBotLocation;

// Returns the number of successful kills recorded so far
function int getCurrentKills() {
        return kills;
}

// Returns the total number of shots fired (attempts)
function float getShotsFired() {
        return shotsFired;
}

// Returns the bot instance that is the current target
function Bot getTargetBot()     {
        return targetBot;
}

// Returns the PlayerPawn instance for the player
function PlayerPawn getPlayer() {
        return player;
}

/* Selects a random start location for the bot from the
 * set define on the map. The previous start location
 * will never be selected again.
 */
function PlayerStart getRandomBotStart()        {
        local int i;
        i = Rand(spCount);
        while (i == lastStart) i = Rand(spCount);
        lastStart = i;
        return botStartPoints[i];
}

// Returns the start location for the player
function PlayerStart getPlayerStart() {
        return start;
}

/* Returns the angle the player is looking in. A
 * Rotator includes Pitch, Yaw and Roll values.
 */
function Rotator getPlayerLookAngle() {
        local Rotator lookAngle;
        if (getPlayer() == None) return lookAngle;
        return getPlayer().Rotation;
}

// Returns the Pitch at which the player is oriented
function float getPitch() {
        return getPlayerLookAngle().Pitch * URotToRadian * RadianToDegree;
}

// Returns the Yaw at which the player is oriented
function float getYaw() {
        return getPlayerLookAngle().Yaw * URotToRadian * RadianToDegree;
}

// Returns the location of bot (target)
function vector getBotLocation() {
        local vector botLocation;
        if (getTargetBot() == None) return botLocation;
```

```
        return getTargetBot().Location;
}
// Returns the location of the player (subject)
function vector getPlayerLocation() {
        local vector playerLocation;
        if (getPlayer() == None) return playerLocation;
        return getPlayer().Location;
}

/* Helper function to calculate Log in any base.
 * Needed because UT only supports Log calculatations
 * in base e.
 */
static final function float Logarithm (float a, optional float Base){
        if (Base == 0) Base = 10;
        return Loge(a) / Loge(base);
}

/* Helper function to computer ACos. Needed because
 * UT only supports ATan, not ASin or ACos.
 */
static final function float ACos (float A) {
        if (A > 1 || A < -1) return 0;
        if (A == 0) return (Pi / 2.0);
        A = ATan(Sqrt(1.0 - Square(A)) / A);
        if (A < 0) A += Pi;
        return A;
}

/* Returns the Index of Difficulty for the current
 * target.
 */
function float getIOD() {
        return IOD;
}

/* Returns the angle of the player's view that the
 * player must move through in order to be pointing
 * at the target. This is used as the Amplitude in
 * the Fitts' Law equation.
 */
function float getSpanAngle() {
        return spanAngle;
}

/* Returns the angle of the player's view that the
 * target bot occupies. this is used as the Width in
 * the Fitts' Law equation.
 */
function float getTargAngle() {
        return targAngle;
}

/* Returns the distance (in UT units) between the player
 * and the target (bot).
 */
function float getDistanceToBot() {
        return dist;
}

// Helper function to convert from radians to degress.
function float degrees(float rads) {
        return rads * RadianToDegree;
}

/* Returns a start point for the given Pawn.
 * If the pawn is the player (subject) then the special
```

```
 * start location set in front of the others is returned.
 * If the pawn is a bot then one of the other start locations
 * is randomly selected. This function also calculates the
 * values required for Fitts' Law calculations.
 */
function NavigationPoint FindPlayerStart(
        Pawn Player,
        optional byte InTeam,
        optional string incomingName ) {
                                                                        180
        local PlayerStart ps;
        local vector pPos;
        local vector bPos;
        local vector obPos;
        local vector pbPos;

        if (Player.IsA('PlayerPawn')) {
                // Pawn is the subject, return the special start location.
                ps = getPlayerStart();
        } else {                                                        190
                // Pawn is a bot, find a random location.

                ps = getRandomBotStart();

                obPos = getBotLocation(); // Where the bot is now
                pPos = getPlayerLocation(); // The player location
                bPos = ps.location; // Where the bot will next appear

                /* The span angle is the angle between where the bot used to
                 * be (was just eliminated), and where the bot will next appear.  200
                 */
                spanAngle = Abs(ACos(Normal(bPos - pPos) dot Normal(obPos - pPos)));

                pbPos = bPos - pPos; // Vector from player to new bot location
                dist = VSize(pbPos); // Distance from player to new bot location

                /* The target angle is the angle of the player's view that
                 * the target occupies.
                 */
                targAngle = Abs(Tan((HIT_RADIUS) / dist)) * 2;          210

                // Fitts' Law formulation of the Index of Difficulty
                IOD = Logarithm((spanAngle / targAngle) + 1, 2);

                spawnTime = currentTime;
                lastBotLocation = ps;

        }

        return ps;                                                      220
}

/* Removes items from the bot's inventory. Do not want
 * the bot to possess a weapon with which they could
 * eliminate the player.
 */
function AddDefaultInventory( pawn PlayerPawn ) {
        if (PlayerPawn.IsA('Bot')) return;
        Super.AddDefaultInventory(PlayerPawn);
}                                                                       230

/* Modify the bot's behaviour to turn it into a
 * stationary target.
 */
function ModifyBehaviour(Bot NewBot) {
        NewBot.setPhysics(PHYS_None);
        NewBot.MaxDesiredSpeed = 0;
```

```
        NewBot.GroundSpeed = 0;
        NewBot.WaterSpeed = 0;
        NewBot.AccelRate = 0;                                                    240
        NewBot.JumpZ = 0;
        NewBot.MaxStepHeight = 0;
        NewBot.AirControl = 0;
}

function float SpawnWait(bot B) { return 0.0; }

/* Removes items from the game that are not appropriate for the
 * experiment.
 */                                                                              250
function bool IsRelevant(actor Other) {

        local PlayerStart pStart;

        if (Other.IsA('TournamentHealth') ||
                Other.IsA('UT_Shieldbelt') ||
                Other.IsA('Armor2') ||
                Other.IsA('ThighPads') ||
                Other.IsA('UT_Invisibility') ||
                Other.IsA('UDamage') ||                                          260
                Other.IsA('Ammo')) return false;

        if (Other.IsA('Weapon') && (!(Other.IsA('SuperShockRifle')))) return false;


        // Record a reference to the bot you will be targetting
        if (Other.IsA('Bot')) targetBot = Bot(Other);

        // Record a reference to the pawn that represents you
        if (Other.IsA('PlayerPawn')) player = PlayerPawn(Other);                 270

        if (Other.IsA('PlayerStart')) {

                pStart = PlayerStart(Other);

                /* A start location tagged with a teamnumber of 1
                 * is the player's special start location. Any other
                 * start location is used for the targets.
                 */
                                                                                 280
                if (pStart.TeamNumber == 1) {
                        start = pStart;
                } else {
                        botStartPoints[spCount] = pStart;
                        spCount++;
                }

        }

        return Super.IsRelevant(Other);                                          290

}


/* StartMatch is called when the game begins. This is the point at
 * which the player gains control.
 */
function StartMatch() {

        local Pawn P;                                                            300

        currentTime = 0.0; // Clock counter reset
        experimentRunning = true; // The experiment is now running
```

```
        kills = 0; // Reset game statistics
        shotsFired = 0;
        killsCounted = 0;

        fired = false;
        justMissed = false;

        Super.StartMatch();

        for(P = Level.PawnList; P != None; P = P.nextPawn) { P.setPhysics(PHYS_None); }

}


/* This event is raised every game tick. deltatime is the time
 * since the last tick.
 */
event tick (float deltatime) {

        currentTime += deltatime;
        lastMouseUpdateTime += deltaTime;

        /* If the experiment is currently running and the allocated
         * time has elapsed then end the experiment.
         */
        if (isExperimentRunning() && (currentTime > EXPERIMENT_LENGTH)) { endExperiment(); }

        // Record the mouse position every MOUSE_RECORD_INTERVAL seconds
        if (isExperimentRunning() && (lastMouseUpdateTime > MOUSE_RECORD_INTERVAL)) {
                lastMouseUpdateTime = 0;
                recordMousePosition();
        }

}

// endExperiment can be called to end the experiment
function endExperiment() {
        experimentRunning = false;
        experimentLog(END_MESSAGE, "Experiment completed");
        endGame("Experiment Over");
}

// Returns the total time elapsed since the game began
function float getTime() {
        return currentTime;
}

// Returns whether the experiment is currently running
function bool isExperimentRunning() {
        return experimentRunning;
}

// The Killed event is triggered whenever one player kills another
function Killed(pawn killer, pawn Other, name damageType) {
        ScoreKill(killer, Other);
}

/* Returns the number of kills that have been recorded. This
 * will not include kills made after a miss.
 */
function int getKillsCounted() {
        return killsCounted;
}

/* Called when one pawn kills another. This includes
 * the case of the suicide where Killer and Other will be the same
 * pawn.
```

```
 */
function ScoreKill(pawn Killer, pawn Other) {

        fired = false;
        if (justMissed == false) {
                recordIOD();
                killsCounted++;
                if (killsCounted == MAX_KILLS) endExperiment();
        }

        justMissed = false;

        Super.ScoreKill(Killer, Other);

        recordKill(Killer, Other);

        kills++;

}

/* Logs data specific to the experiment. Each line is prefixed by a
 * tag so that it can be easily recognised amongst the normal log
 * messages generated by UT.
 */
function experimentLog(string ev, string message) {
        log(LOG_TAG $ DELIMITER $ getTime() $ DELIMITER $ ev $ DELIMITER $ message);
}

// Records a kill within the experiment
function recordKill(pawn Killer, pawn Other) {
        experimentLog(KILL_MESSAGE, String(Int(Killer.PlayerReplicationInfo.Score)));
}

// Records the Index of Difficulty for the current target
function recordIOD() {
        local float delta;
        delta = currentTime - spawnTime;
        lastIODRecordTime = currentTime;
        log(LOG_TAG $ DELIMITER $ delta $ DELIMITER $ "IOD" $ DELIMITER $ String(IOD));
}

/* Records the current position of the mouse in terms of the
 * pitch and yaw within the 3D environment.
 */
function recordMousePosition() {
        local Rotator looking;

        if (getPlayer() == None) return;

        looking = getPlayerLookAngle();
        experimentLog(MOUSE_MESSAGE, Int(getPitch()) $ DELIMITER $ Int(getYaw()));
}

// Records the current position of the bot (target)
function recordBotPosition() {

        local vector botPosition;
        local vector yourPosition;

        if (getTargetBot() == None) return;
        if (getPlayer() == None) return;

        botPosition = getBotLocation();
        yourPosition = getPlayerLocation();

        experimentLog(
                BOT_POS_MESSAGE,
```

```
                Int(botPosition.X − yourPosition.X) $
                DELIMITER $
                Int(botPosition.Y − yourPosition.Y) $
                DELIMITER $
                Int(botPosition.Z − yourPosition.Z));
}

// Records that the player fired the weapon
function recordWeaponFire() {
        local Rotator looking;
        if (getPlayer() == None) return;
        looking = getPlayerLookAngle();
        experimentLog(FIRE_MESSAGE, Int(getPitch()) $ DELIMITER $ Int(getYaw()));
        shotsFired++;

        if (fired) justMissed = true;
        fired = true;

}

// Called when the game ends
function bool SetEndCams(string Reason) {

        local pawn P, Best;

        for (P = Level.PawnList; P != None; P = P.nextPawn) {
                if (P.bIsPlayer) {
                        P.ClientGameEnded();
                        P.GotoState('GameEnded');
                }
        }

        log("Game ended at " $ EndTime);
        CalcEndStats();
        return true;

}

/* Called when a player re-enters the game. In this experiment,
 * this only happens when the bot reappears.
 */
function bool RestartPlayer( pawn aPlayer )     {

        local bool retVal;
        retVal = Super.RestartPlayer(aPlayer);
        if (aPlayer.IsA('Bot')) recordBotPosition();
        return retVal;

}

/* These methods have been overridden with empty bodies to
 * remove distracting behaviour from the experiment.
 */
function NotifySpree(Pawn Other, int num) { }
function EndSpree(Pawn Killer, Pawn Other) { }

// Default properties allow options in the game to be configured.
defaultproperties {
        MapPrefix="ND"
        BeaconName="ND"
        MaxPlayers=2
        DefaultPlayerState=PlayerStanding
        bChangeLevels=False
        HUDType=Class'NonDom.ExperimentHUD'     // Custom HUD (heads up display)
        GameName="Non-Dominant Hand Experiment" // Name of the experiment
        DefaultWeapon=Class'NonDom.NonDomRifle' // Using the shockrifle to eliminate damage problems
}
```

HUD

```
class ExperimentHUD extends ChallengeHUD;

/* ExperimentHUD
 *
 * This class provides a custom HUD (Head Up Display). A custom
 * HUD will remove distracting information normally shown with
 * the default HUD.
 */

/* Returns a reference to the Experiment game class to allow
 * communication between the HUD and the game controller.
 */
function Experiment getExperiment() {
        return Experiment(Level.Game);
}

/* This method is overridden so that only the crosshair is
 * shown. No other distracting information is drawn.
 */
function PostRender(canvas C) {
        DrawCrossHair(C, 0,0 );
}

defaultproperties {}
```

Weapon

```
class NonDomRifle extends SuperShockRifle;

/* NonDomRifle
 *
 * This class provides a weapon for the player. The fire
 * event is overridden so that the game modification can
 * know when the weapon was fired.
 */

function Experiment getExperiment() {
        return Experiment(Level.Game);
}

function Fire( float Value ) {
        getExperiment().recordWeaponFire();
        Super.Fire(Value);
}

defaultproperties {
     ItemName="Non-Dominant Hand Rifle"
}
```