Western Graduate&PostdoctoralStudies

Western University

Scholarship@Western

Electronic Thesis and Dissertation Repository

9-3-2015 12:00 AM

Endoscopic Targeting Tasks Simulator: An Approach Using Game Engines

Shaun W. Carnegie
The University of Western Ontario

Supervisor
Dr. Roy Eagleson
The University of Western Ontario Joint Supervisor
Dr. Sandrine de Ribaupierre
The University of Western Ontario

Graduate Program in Electrical and Computer Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science
© Shaun W. Carnegie 2015

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Biomedical Commons, Biomedical Devices and Instrumentation Commons, Controls and Control Theory Commons, and the Systems and Communications Commons

Recommended Citation

Carnegie, Shaun W., "Endoscopic Targeting Tasks Simulator: An Approach Using Game Engines" (2015). *Electronic Thesis and Dissertation Repository*. 3248.
https://ir.lib.uwo.ca/etd/3248

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

ENDOSCOPIC TARGETING TASKS SIMULATOR: AN APPROACH USING GAME ENGINES

(Thesis format: Monograph)

by

Shaun Carnegie

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Engineering Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Shaun Carnegie 2015

Acknowlegements

I would like to begin by thanking all the participating members of my thesis examination committee: Dr. Luiz Capretz, Dr. Ali Khan, Dr. Serguei Primak, and Dr. Jagath Samarabandu for making time in your busy schedules to be a valuable part of my examination committee. Your experience and guidance has been an invaluable addition not only to my presented research but also my appreciation for research and academia.

I would also like to thank my supervisors: Dr. Roy Eagleson and Dr. Sandrine de Ribaupierre for a truly remarkable experience these last two years (and then some...). Roy your constant enthusiasm and passion for research has served as the catalyst that drove me to always strive for more and never stop pushing the bounds for what's next. Sandrine, your insight into not only my research but that of the research group as a whole has always served as a guiding light for the work I have explored. Your clinical knowledge has been an indispensable addition to my work and opened my eyes to areas of research I never would have thought possible. The way in which you both compliment one another and your capacity to seamlessly jump from one area of research to another never ceases to amaze me. Thank you for making me apart of what I consider to be a close family of researchers that have had the privilege of working under your guidance.

I consider myself very fortunate to have had the support and friendship of the members from both Dr. Eagleson's and Dr. de Ribaupierre's research group, namely: Matt Kramers, Jing Jin, Saeed Bakhshmand, Lauren Allen, Oleksiy Zakia, Dayna Noltie, Ngan Nguyen and Ryan Armstrong. I would like to give a special thanks to Justin Mackenzie; it was of great comfort knowing that I always had someone that was in the same boat as me. Justin we had the unique fortune for being able to experience this exciting endeavour together, right from the first day. You had a great influence on helping me find my way and providing me with inspiration and reassurance. I wish you all the best with your family and future academic endeavours.

I would like to thank my family for being the constant, reassuring, and positive force that has supported me in every way possible as I have pursued my dreams and interests. It has been a long and far from ordinary road to get to this point and the one thing that has remained constant has been your unshakeable support for me through the highs and the lows. To my brother Dave, I wan't to give a special thanks for helping to guide me through a process that I knew very little about and for always offering a helping hand.

Lastly, I would like to thank my beautiful and amazingly patient and tolerant wife, Adara. First let me just say WOW! What a ride these last 12 plus years have been. I can honestly say that I couldn't have done this without your constant love and support. While I would never have envisioned us having taken the long and often stressful path that we have, I can honestly say that I wouldn't change any of it. Not only as my wife, but also more recently as the mother of our beautiful twin girls Zoey and Lola, you have truly shined as the amazingly resilient, strong, and incredibly patient woman that has made all of this possible. Thank you for being the rock that has kept our home and family strong and allowed me to pursue my dreams!

Abstract

The pervasiveness of simulators used in professions requiring the skilled control of expensive machinery such as is the case in the aviation, mining, construction, and naval industries raises an intriguing question about the relatively poor adoption within the field of medicine. Certain surgical procedures such as neuro-endoscopic and laparoscopic lend themselves well to the application of virtual reality based simulators. This is due to the innate ability to decompose these complex macro level procedures into a hierarchy of subtasks that can be modelled in a software simulator to augment existing teaching and training techniques.

The research in this thesis is focused with the design and implementation of a targeting-based simulator having applications in the evaluation of clinically relevant procedures within the neuro-endoscopic and potentially laparoscopic domains. Existing commercially available surgical simulators within these domains are often associated with being expensive, narrowly focussed in the skills they train, and fail to show statistically significant results in the efficacy of improving user performance through repeated use.

Development of a targeting tasks simulator is used to evaluate what methods can be applied to provide a robust, objective measure of human performance as it relates to targeting tasks. In addition to performance evaluation, further research is conducted to help understand the impact of different input modalities; focusing primarily on input from a gamepad style device and as well a newer, more natural user interface provided by the Leap Motion Controller.

Keywords: Surgical Simulation, Performance, Human Computer Interface, Targeting Tasks, Software Architecture, Game Engine, User Input

Contents

Certificate of Examination ii

Ackndowlegements ii

Abstract iii

List of Figures vii

Listings ix

List of Appendices x

List of Abbreviations, Symbols, and Nomenclature xi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Objectives . . . 2
- 1.3 Structure of Thesis . . . 3

2 Surgical Simulator Design Background 4
- 2.1 ETV: Endoscopic Third Ventriculostomy . . . 4
- 2.2 VR Simulators . . . 5
- 2.3 Serious Games . . . 7
- 2.4 Game Engines . . . 8
- 2.5 Fitts' Law: Targeting Performance . . . 12

3 Requirements and Modes of Operation 14
- 3.1 Overview . . . 14
- 3.2 Scenario Designer . . . 14
  - 3.2.1 Scenario Creation . . . 14
  - 3.2.2 Trial Dataset Creator . . . 15
- 3.3 Targeting Simulation . . . 16
  - 3.3.1 User Input . . . 16
    - Leap Motion . . . 17
    - Game Pad Controller . . . 17
  - 3.3.2 Data Acquisition . . . 17
- 3.4 Analysis and Evaluation . . . 17

3.4.1 Scenario Playback and Teaching . . . 18
3.4.2 User Performance Evaluation . . . 18

4 Targeting Simulator: Render Engine 20
4.1 Targeting Tasks Simulator: Overview . . . 20
4.2 Development Environment . . . 20
4.3 Architecture . . . 21
4.3.1 Objective-C Specific Considerations . . . 21
4.3.2 Modular Design . . . 21
4.3.3 Front-End . . . 21
4.3.4 Back-End . . . 22
4.4 Render Pipeline and Flow . . . 22
4.5 Scenes and Scene Objects . . . 25
4.5.1 Structure . . . 26
Entity . . . 27
CameraObject . . . 28
LightSource . . . 28
4.5.2 Serialization . . . 29
4.5.3 Extraction and Importing . . . 30
4.5.4 Depth Sorting . . . 30
4.6 Shader and Materials Bank . . . 33
4.6.1 Shader and Material Overview . . . 33
4.6.2 Shader Structure and Management . . . 33
4.6.3 Material Structure and Management . . . 34
4.7 Collision Detection . . . 35
4.8 Trial Data . . . 37
4.8.1 Structure and Overview . . . 37
4.8.2 Dataset Generation . . . 37
4.8.3 Capturing Trial Session Data . . . 39
4.8.4 Session Playback and Analysis . . . 41
4.9 Ellipsoid Targets . . . 49
4.9.1 Ellipsoids and Eccentricity . . . 49
4.9.2 Ellipsoid Target Generation . . . 49
4.10 Mesh Input: OBJ Parser . . . 50
4.10.1 OBJ File Format . . . 50
4.10.2 Parsing and Importing . . . 52

5 Targeting Simulator: User Interaction 54
5.1 Gamepad Controller . . . 54
5.1.1 Gamepad Controller: Framework . . . 58
5.2 Leap Motion Controller . . . 61
5.3 Voice Control . . . 63
5.4 Multi Touch Trackpad Support . . . 64

6 Results 65

6.1 Targeting Tasks: User Input . . . 65
6.2 Playback and Analysis . . . 67
6.3 Trial Data Acquisition . . . 68
6.4 Application of Fitts' Law . . . 68

7 Closing Remarks 77
7.1 Discussion and Conclusion . . . 77
7.2 Future Work . . . 78

Bibliography 80

A Additional Listings 83

Curriculum Vitae 87

List of Figures

2.1 Illustration showing how fenestration of ventricle floor is carried out during ETV. Vogel [32] 5
2.2 Illustrations of the NeuroTouch ETV and LapMentor Simulators 6
2.3 Figures illustrating the apparent visual disparity between the serious games: Critical Transport and Central Hospital 7
2.4 Figure showing initial OR recreation using Evolution Engine 11
2.5 Figure showing Fitts' classic 1D pointing task 12

3.1 Scenario Designer Use Case Diagram. 15
3.2 Trial Dataset Generator Use Case Diagram. 16
3.3 Simulator Use Case Diagram. 18
3.4 Playback and Analysis Use Case Diagram. 19

4.1 Abstracted Flowchart of the Render Pipeline and Flow. 23
4.2 Figure depicting basic structure of scene graph using mutable array with indexes pointing to individual scene objects 27
4.3 Entity class diagram 27
4.4 CameraObject class diagram 28
4.5 LightSource class diagram 28
4.6 No application of object-based depth ordering. Notice how the planes that are intersecting the manikin head of the perspective view are not drawn within the bounds of the translucent mesh. 31
4.7 Same scene rendered using the discussed method for sorting the scene based on object depth. Notice how the planes that are intersecting the manikin head of the perspective view are now drawn within the bounds of the translucent mesh. 32
4.8 Figure depicting structure of shader bank. Objects along with the render enging hold shader resources which specify the slot number corresponding to the compiled program. 34
4.9 Class diagram of the Materials object 35
4.10 Figure depicting results of collision detection system. Purple marker represents the tool's projected contact point on the skull surface, while the green marker represents the ideal position to achieve zero error. 36
4.11 Trial Dataset Generator window - example of available configuration options. . 40
4.12 Trial Dataset Generator window - example of available eccentricity configuration options. 41
4.13 Playback and Analysis of ellipsoid targeting task. 44
4.14 Playback view of ellipsoid targeting task. 45

5.1 Gamepad Controller Mapping [18] 55
5.2 Gamepad input ramp functions for camera control 55
5.3 Gamepad input ramp functions for tool control 56
5.4 Class diagram of Gamepad object 58
5.5 Class diagram for ControllerComponent object 59
5.6 Figure showing Leap Motion Device and Coordinate System [21] 62

6.1 Analysis of ellipsoid targeting task using gamepad input device. 66
6.2 Analysis of ellipsoid targeting task using Leap Motion input device. 67
6.3 Histogram of target eccentricity values 69
6.4 Distribution of target eccentricities throughout the dataset 69
6.5 Comparison of *Total Time*, *Setup Time*, and *Movement Time* for Gamepad input device as a function of target eccentricity 70
6.6 Comparison of *Total Time*, *Setup Time*, and *Movement Time* for Leap Motion input device as a function of target eccentricity 70
6.7 Results of normalized Distance and Rotation error (D-ERR and R-ERR respectively) recorded over successive trials as a function of target eccentricity using the gamepad controller 71
6.8 Results of normalized Distance and Rotation error (D-ERR and R-ERR respectively) recorded over successive trials as a function of target eccentricity using the LMD 71
6.9 Plot showing the relation between the calculated distance and rotation indexes of difficulty and that of the target eccentricity using results from the gamepad trials 72
6.10 Plot showing the relation betweeh the calculated distance and rotation indexes of difficulty and that of the target eccentricity using results from the LMD trials 72
6.11 Plot showing the relation of normalized distance error to the calculated distance and rotation indexes of difficulties using results from the gamepad trials 73
6.12 Plot showing the relation of normalized distance error to the calculated distance and rotation indexes of difficulties using results from the LMD trials 73
6.13 Plot showing the relation between *MT* and *ID* for the calculation of *IP* from the gamepad trials 74
6.14 Plot showing the relation between *MT* and *ID* for the calculation of *IP* from the LMD trials 75
6.15 Plot showing the improved performance that results from running repetitive trials using the gamepad 75

Listings

4.1 Configuration of Perspective Viewport 24
4.2 Configuration of Orthographic Viewport 25
4.3 Preparation of object buffers for submission to GPU for rendering 26
4.4 JSON Serialization of Scene 29
4.5 Compressed sample of trial session data 38
4.6 CSV example of trial session data 39
4.7 Excerpt from trial dataset generation process 42
4.8 Function used to start capturing trial data 42
4.9 Function used to capture scene state 43
4.10 Excerpt of tool and target isolation in replay file 44
4.11 Excerpt of replay analysis preprocessing 46
4.12 Continuation of Listing 4.11 calculating normalized distance and rotation error 47
4.13 Calculation of Fitts' Law Parameters 48
4.14 Playback state behaviour of render engine 48
4.15 Transformation of eccentricity applied to spherical mesh to generate ellipsoid target 50
4.16 Excerpt from OBJ Mesh file for a textured cube 51
5.1 Applying Input Ramp Functions to Analog Sticks 56
5.2 World Rotation About a Bound Object 57
5.3 Local Rotation of Camera Object 58
5.4 Preprocessing of Analog Stick Input 60
5.5 Handling of New Device Available Event 60
5.6 Handling of Has Data To Be Read Event 61
5.7 Handling of Device Removed Event 61
5.8 LMC processing of position and rotation data 63
A.1 Serialized Scene File in JSON 83
A.2 Playback slider action function used to update seek position 84
A.3 Excerpt from Xbox 360 Controller Profile 85

List of Appendices

Appendix A Additional Listings 83

List of Abbreviations, Symbols, and Nomenclature

AE	Animation Engine
CSF	Cerebrospinal Fluid
CSV	Comma Separated Values
DOF	Degrees of Freedom
EE	Evolution Engine
ETTS	Endoscopic Targeting Tasks Simulator
ETV	Endoscopic Third Ventriculostomy
FPS	Frames Per Second
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HID	Human Interface Device
ID	Index of Difficulty
IK	Inverse Kinematics
IP	Index of Performance
JSON	JavaScript Object Notation
LMD	Leap Motion Device
MT	Movement Time
MVC	Model View Controller
TDSG	Trial Dataset Generator
TP	Throughput
UID	Unique Identifier
VBO	Vertex Buffer Object
VR	Virtual Reality

Chapter 1

Introduction

1.1 Motivation

The continuos evolution of the video game entertainment industry has provided a driving force for the development of ever more powerful video game engines that are capable of powering immersive environments with a multitude of different genres of entertainment. One thing that becomes apparent when looking at the games built on top of these ever evolving engines is that they strive to create an interactive experience for players to overcome challenges and complete tasks. A fortunate byproduct of the video game industry is the above mentioned game engines. These engines allow users to create games of their own design.

The motivation for the design and implementation of a virtual reality (VR) based surgical simulator stems from the apparent lack of simulator use in the domain of surgical training and teaching as compared to other professions such as aviation, aerospace, heavy equipment (i.e trucks and construction vehicles) operation, and more. While it is true that there is a growing body of research dedicated to the development of software/hardware medical simulators and trainers, existing solutions are typically prohibitively expensive, highly specialized, and produced in low volumes, reducing their effectiveness as a training aid through inaccessibility. Existing surgical simulators such as the Neuro-touch developed by the NRC are typically designed to carry out a multitude of different scenarios for a given task; however, they generally have a very narrow scope and fail to expand on some of the broader tasks that may be desired. The result is that while existing simulators can be highly effective in augmenting traditional training methods, costs and availability mean that access and time spent using the simulators may be hampered. The full capability of simulators is therefore not being realized in all cases.

Existing research on the efficacy of surgical simulators for both training and evaluation is lacking and is perhaps one of the reasons why the medical profession has seen such a low adoption rate of simulator use in this capacity as compared to some of the other professions mentioned. There is a clear need to evaluate not only the possible efficacy of surgical simulators in both an evaluation and training role, but also to evaluate the capacity in which such simulators could operate.

Of particular interest within the domain of surgical simulators is their potential application in evaluating user performance objectively over a wide range of clinically relevant tasks and procedures. The ability to gather all of the pertinent information attained through a user's interaction with a simulator while performing a task presents a unique opportunity to then analyze the collected data for both evaluation and constructive feedback.

Similar to the explosion of the video game industry over the last couple of decades, computer driven simulators have also made their way into numerous different industries and applications. The continual increase in computer hardware performance matched with declining production costs has made high performance computing more affordable and thus more attractive for applications such as simulators.

The ideal case would seem to be a Surgical simulator that is as prevalent and accessible as any modern video game (different platforms) while offering comparable fidelity and educational value. It is easy to see how parallels could be drawn between video games and Virtual Reality (VR) Simulators; Identifying the differences tends to be a bit more difficult. Looking to the definition of what a simulator is: *"a machine with a similar set of controls designed to provide a realistic imitation of the operation of a vehicle, aircraft, or other complex system, used for training purposes."*[22] we see that the only real distinction between a simulator and a game is the purpose/way in which the software (pertaining to video game or simulator) is used. Video Games are generally seen as programs that provide entertainment to a player whereas Simulators are viewed as educational programs that provide the player/user with training. This important distinction or lack there of lends itself to the theory that the same methods/tools (i.e Video Game Engines) used to produce video games can be applied to the development of Simulators. The initial focus of developing such a simulator pertains to the simulation of Endoscopic procedures though as mentioned above, the scope in which a Surgical Simulator operates need not be limited to a single procedure.

1.2 Research Objectives

The overarching goal of this thesis is to investigate not only the feasibility of developing low(er) cost surgical simulators using the openly available tools developed and tested by the video game industry, namely game engines, but also to investigate their efficacy as a teaching and training aid. Furthermore, it is also the goal to research what methods can be employed to obtain the means necessary to objectively evaluate user performance in scenarios requiring the execution of clinically relevant skills and procedures.

The focus of this research is primarily concerned with the simulation of endoscopic procedures such as the Endoscopic Third Ventriculostomy (ETV) which is discussed further in Section 2.1. The last research objective of this thesis is to investigate the role of Human Interface Devices (HID) in simulator design and performance, namely the comparison between a traditional gamepad style input and the Leap Motion Controller which is considered to be a more natural user interface. Of particular interest is trying to ascertain what form of input provides users with the most valuable experience when it comes to improving user performance in the skills needed to perform complex surgical procedures in a clinical setting.

1.3 Structure of Thesis

This thesis focusses on the design and implementation of an endoscopic targeting tasks simulator (ETTS) with the intent of evaluating human performance and trying to ascertain the efficacy of such a system in improving clinically relevant skills.

Chapter 2 provides a background overview of some of the key themes that are relevant and discussed throughout the thesis. In particular we start by taking a look at the ETV procedure and how it can be modelled using a targeting-based simulator approach, followed by an overview of Virtual Reality (VR) based simulators. Other topics include looking at existing work in the area of serious games, game engines and their applications, and finally we wrap up with a review of Fitts' Law and its application in evaluating human performance.

Chapter 3 is focusses on outlining the requirements of a targeting-based simulator, looking at the different roles the simulator needs to operate in. A use-case analysis is used to review what features are needed and the different behaviours the system must take on depending on the type of user.

Chapter 4 provides a comprehensive overview of the simulator's architecture and design, focussing on the render engine that makes up the heart of the ETTS system. Topics covered include a look at the innovative structures used for capturing and storing user session data for analysis and evaluation. This chapter also discusses the methods used in generating the ellipsoid mesh targets that allow for the measurement of human performance.

Chapter 5 is centred around the user interaction aspect of the ETTS system, specifically looking at the different input modalities that can be used and the implementation that goes along with it. These input modalities include a support for using a USB gamepad, Leap Motion, and voice control.

Chapter 6 provides a discussion of the results obtained from running trials of the ETTS system. In particular it looks at how Fitts' Law can be applied to evaluate user performance. Of particular interest is looking at the simulator's efficacy in improving user performance over time, this analysis is done using Fitts' Law and resolving a user's performance to an index of performance that can be used to track a user's skills overtime.

Chapter 7 provides a conclusion of the research conducted and looks at how the results discussed in Chapter 6 compare with the expectations set forth.

Chapter 2

Surgical Simulator Design Background

In Chapter 1 an outline of this thesis was provided, defining the motivation behind the research, setting the objectives for the research being done, and lastly discussed structure of the coming chapters. In this chapter, some background knowledge of the target procedure that an ETTS system can replicate is provided, which is the ETV. Also, background knowledge on VR simulators in the surgical domain, serious games for teaching and training purposes, game engines and their role in simulator development, and lastly an overview of Fitts' Law and its use in evaluating human performance will be discussed.

2.1 ETV: Endoscopic Third Ventriculostomy

The surgical procedure of concer in the evaluation of a simulator implemented using the techniques employed by game engines, is the Endoscopic Third Ventriculostomy (ETV) [14]. This particular procedure is of great interest as it can be decomposed into several procedural sub-tasks[10], that can be clearly defined, a quality which lends itself well to the design of any simulator or software engineering problem for that matter.

The ETV procedure is considered an alternative to the placement of Cerebrospinal Fluid (CSF) shunts in the treatment of hydrocephalus [15], a condition wherein the patient experiences an abnormal buildup of CSF within the cranial ventricles, posing potentially serious risks due to the pressure exerted on the surrounding brain tissues. If the condition is left untreated risk factors can include a lack of normal development and lead to neurological conditions such as: "ataxia, visual impairment, mental and growth retardation, and seizures" [25].

Both the ETV and CSF shunt placement procedures work to drain excess fluid from the ventricular system, resulting in the decrease of ventricle volume and thus result in the reduction of intracranial pressure [32]. While it is not possible to label one approach more preferable over the other in all cases, there is research that suggests that the high failure rate of shunts [8], [24] has lead to the increased popularity of ETV whenever possible. At the same time there is clear evidence showing a higher immediate risk associated with ETV procedures at the cost of offering improved longterm outcomes [24]. Due to the complexity and risks associated with performing ETV's, intensive training is required, usually relying on the the use of cadavers [24]; this suggests that alternative methods for training residents in a safe environment could help improve the performance of practitioners in a clinical setting, ultimately leading to improved patient outcome.

The ETV procedure works to increase natural drainage of CSF within the ventricular system through fenestration of the ventricle's floor. Figure 2.1 illustrates how the ETV procedure is performed, showing the endoscope entering through the third ventricle and against the floor.

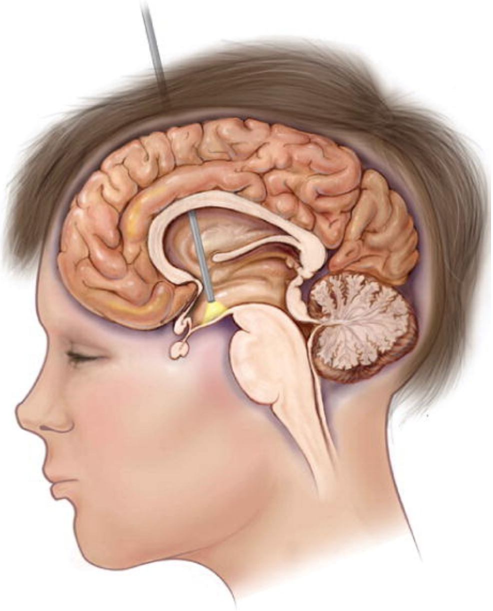


Figure 2.1: Illustration showing how fenestration of ventricle floor is carried out during ETV. Vogel [32]

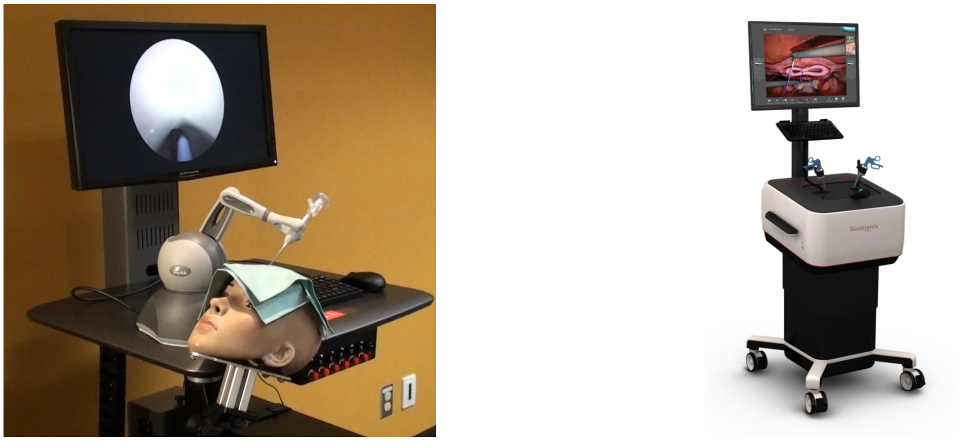
During an ETV a surgeon begins by placing a burr-hole in the patient's skull through which a trocar/sheath is inserted for the passage of endoscopic tools and other surgical instruments throughout the procedure. Once the endoscope has been inserted through the trocar/sheath, the tool is passed through frontal horn of the lateral ventricle from where it is then navigated into the third ventricle by way of the *Foramen of Monro*. An initial perforation of the ventricle floor is made before a catheter balloon is inserted allowing for full fenestration of the floor and the opening to be widened (approximately 5mm) [9].

The ETV procedure represents a complex surgical task that requires a great deal of training and skill to perform. The pertinent skills that can be embodied in an endoscopic surgical simulator include, though are not limited to: understanding of patient specific anatomy and different presentations, interpretation of anatomy through different imaging modalities, burr-hole placement, and fenestration of ventricular floor. Skills required though often not covered by existing surgical simulators include: multi-personnel training, complications encountered during the procedure, and planning.

2.2 VR Simulators

The application of VR simulators in the domain of endoscopic and laparoscopic surgical procedures presents a unique opportunity to enhance existing methods of training [27] and evaluation of surgical clinicians [11]. The unpredictable nature of varying patient anatomies and anomalies combined with potential complications encountered [20] during procedures makes training and exposure for novice clinicians difficult. The key opportunity and motivation behind the exploration of VR surgical simulators lies in their ability to present users with more realistic representations of incredibly complex procedures and scenarios that would either be too dangerous or too expensive and/or too time consuming in real life, in the context of training surgical clinicians.

Two existing surgical VR simulators worth noting are the *NeuroTouch* [4] developed by the National Research Council of Canada and the *LapMentor* developed by Simbionix [26]. Both simulators provide the ability for users to interact with a haptic interface to carry out complex surgical procedures in a safe environment with the goal of improving skills that are transferrable to the operating room (OR). Figures 2.2a and 2.2b show the *NeuroTouch* and *LapMentor* respectively.



(a) NeuroTouch ETV interface, Mcgill [1]

(b) LapMentor Interface, Simbionix [26]

Figure 2.2: Illustrations of the NeuroTouch ETV and LapMentor Simulators

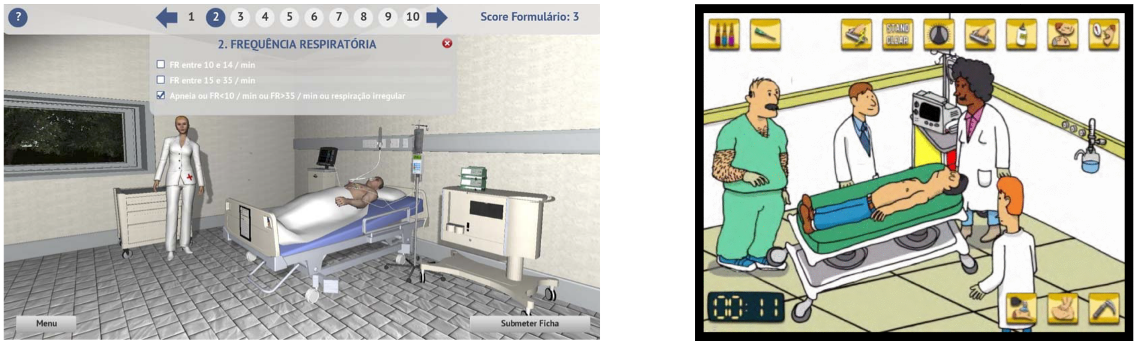
Both simulators support the ability to add modules that provide additional resources for training and evaluating user performance. The shortcoming of these simulators is that their prohibitive cost and limited production make it difficult to provide accessibility to the platforms to all those that would benefit. The *LapMentor* is perhaps one of the strongest examples of demonstrating the effectiveness of using VR surgical simulators to improve training in the domain of laparoscopic procedures [2] [12]. There is however conflicting research on the efficacy of these two prominent systems with respect to their ability of providing a significant improvement to the clinically relevant skills that are transferrable to the OR [16]. Further to the efficacy of the simulators, there is also compelling research suggesting that the haptic interfaces used by these two commercially available simulators are a costly option with no clear benefit to user performance [29]. The lack of evidence demonstrating the effectiveness of haptic interfaces suggests that low(er) cost VR simulators providing greater availability could be developed with the same effectiveness or greater than what is currently available.

2.3 Serious Games

The use of games and in particular video games to enhance traditional teaching methods has shown a steep rise in recent years, finding its way into numerous educational topics and levels. There is evidence demonstrating that the value of educational games is not equal in all areas, suggesting that some areas such as engineering and medicine are better suited to the use of this genre of video games as compared with others [3].

The genre of Serious Games is generally used to describe those video games that provide some degree of educational value or simulation of events. The difference between simulators, video games, and Serious Games can be difficult to distinguish as they all stem from the same core technologies and on the surface may even appear similar. The main distinction appears to lie in what is taken away from the user experience; whereas video games are generally used for recreational entertainment, simulators are used to provide training and experience on virtual facsimiles of equipment/machines and or procedures, while serious games can be viewed as a hybrid of the two. Serious Games strive to bridge the gap between education and entertainment, using the allure of video games to provide the user with an engaging experience that has some educational component.

This is an important realization as it further reinforces the hypothesis of being able to develop Serious Games using the readily available tools and techniques of the video game industry. The use of Serious Games to augment traditional teaching methods in the area of medicine has already been seen with such projects as *Critical Transport* [23], *Central Hospital - Master of Resuscitation* [19], and *Second Life* [33]. Each of these projects make use of Serious Games to provide different educational topics within the domain of medicine using different platforms. An important observation in the design and implementation of these games is that the level of fidelity varies greatly from one implementation to another. Interesting research concerning the level of both visual and cognitive fidelity in Serious Games has shown that depending on the type of task, the level of visual fidelity may prove irrelevant when it comes to improving user performance [31]. Figures 2.3a and 2.3b exemplify the disparity in the apparent visual fidelity between *Critical Transport* and *Central Hospital* respectively.



2. FREQUÊNCIA RESPIRATÓRIA

☐ FR entre 10 e 14 / min
☐ FR entre 15 e 35 / min
☑ Apneia ou FR<10 / min ou FR>35 / min ou respiração irregular

(a) Critical Transport UI [23]

(b) Central Hospital UI [19]

Figure 2.3: Figures illustrating the apparent visual disparity between the serious games: Critical Transport and Central Hospital

Critical Transport is a game designed to help educate medical students about choosing the appropriate mode of transport for patients who are critically ill or in need of attention. Research into the use of this Serious Game showed promise in its ability to improve user's test scores based on the knowledge acquired through their experience with the game [23].

Central Hospital - Master of Resuscitation, an online Serious Game developed by *Stand Clear*, allows medical professionals of different disciplines to enhance their domain knowledge through a problem solving approach. Users are presented with simulated problems that are clinically relevant and are required to apply their domain knowledge to work on a problem solving task [19].

The *Second Life* program is a massive online platform that allows users to interact with each other through a virtual world that is completely customizable. The platform itself struggles to meet the definition of a game, but instead is designed as a tool in which games among other things can be developed. A study investigating the potential applications of the *Second Life* VR world platform for medical education was conducted to test how such a system could be used to convey medical knowledge to students. The study used the virtual environment to conduct a training seminar with healthcare professionals on the topic of type two diabetes in order to understand the effectiveness in using such an approach for medical education. The results of the experiment revealed that the use of Serious Games (or in this case the platforms used by such games) showed potential in improving traditional teaching methods through the adoption of such VR platforms/games as *Second Life* [33].

2.4 Game Engines

The development of the game engine by the video game industry represents a shift in the way that games are now developed, embracing the fundamentals of object oriented programing (OOP) through the abstraction of the game engine from the video games themselves. Development of game engines represents a tremendous expenditure of both time and work and so naturally it only makes sense to build a platform that is as portable and reusable as possible.

The evolution of game engines and the video game industry has witnessed a shift away from studios developing their own proprietary engines in exchange for licensing established engines that have already been developed. Removing the burden of developing a game engine from scratch has resulted in the increased development of titles by indie developers, demonstrating the feasibility of game development outside of large production AAA studios.

The game engine itself serves as the framework on which games are developed and often offer a suite of tools to aid the developer in utilizing the features offered [5]. These tools often include the following:

Scripting: While most engines are closed source (e.g. the *Evolution* (see below) and *Unity* engines), the support for scripting by the user allows for behavioural programming and customization of tools in the case of Unity. Generally there is support for multiple scripting languages that the user can choose from.

Editor: The editor serves as a graphical environment in which the user can place objects and decorations within the scene/level and design the virtual world, often through a drag and drop interface. Configuration of scene objects and attributes, along with lighting and material attributes, is generally performed through the use of an editor.

Animation: An important aspect of most genres is the ability to have animated character movement driven either by artificial intelligence (AI) logic (through scripting in most cases) or through user input. Animation has rapidly evolved to include support for such features as inverse kinematics (IK), scripted gestures (e.g running and jumping), and an ability to interpolate between pre-programmed animations to provide more realistic movements. Animations are often configured through the use of an animation tree that operates with the use of a state machine design.

Physics: The improving performance of modern graphics hardware and the graphics processing unit (GPU) has spurred the development of physics engines such as *Nvidia's PhysX*. These additions to the game engine provide a powerful ability in the realtime calculation and recreation of realistic physics. This allows for the impressive modelling of rigid body deformation, cloth and particle effects.

Audio: The use of sounds and music in games is as important as any gripping movie and as such is a major part of game engines. More recently the support for three dimensional (3D) audio has been added to engines to further enhance the realism of sounds by allowing moveable audio sources in relation to the player's position, while adding directionality.

Some Form of AI: The use of AI in video games extends back to their infancy and is used to provide computer controlled behaviour of non-player characters (NPC). Some set of features for the support of AI behaviour is generally included in most game engines, with the ability to augment it further through the use of scripting and/or behaviour/decision trees.

The Unity engine [30] offers developers a comprehensive suite of tools and, due to its ever increasing user support base, the number of additional modules offered is impressive. Customization of Unity's interface, and that of the games developed on top of it is done through the use of scripting, with support for a number of languages to suit the developer's preference. A main drawback of the engine is the closed source nature that restricts the level of customization to that which has been exposed through the scripting interface and as a result user input devices are largely abstracted to a higher level that prevents the low level interfacing with custom hardware. Other restrictions (some dependent on licensing of proprietary features) include absolute control over the rendering pipeline, some physics related features, and implementation of soft body deformation.

Evolution Engine (EE) [6]is a video game engine developed at Digital Extremes (DE) in London, Ontario. The primary target genre of the EE is for first person shooter (FPS) games and is designed to support multiple gaming consoles. Early in our research phase we had the opportunity to collaborate with the team at DE and were given access to a branch of the EE with the goal of researching its applications as a potential candidate for the use of a game engine based surgical simulator, specifically for the simulation of endoscopic procedures such the ETV.

One of the challenges with working with an existing engine such as Evolution is they are typically optimized for a single genre of game development; therefore, adaptation to alternative applications can be taxing.

Through analysis of the features needed for the simulation of endoscopic procedures, the following were identified:

- The implementation of a live in-game camera with imaging properties similar to that of an endoscopic camera
- The ability to apply user input to the control and manipulation of endoscopic tools in an avatar's hands
- The need to reconstruct the process of interpreting pre-operative scans (multi-modal imaging: mainly MRI, CT) for the purpose of identifying burr-hole placement and targeting approach

While this list is not exhaustive, it does represent the basic needs for developing a framework on which endoscopic procedures can be modelled and evaluated.

One can see the challenges involved in adapting a FPS purposed engine such as EE to support features listed above, namely the articulation of an avatar's hands and the ability to augment a pre-existing rendering pipeline to support the addition of one or more cameras in parallel to the pre-existing avatar's camera (in order to recreate the functionality of an endoscopic camera). Research into the control of avatar movement, specifically upper body joints of the player skeleton, revealed that movement is accomplished through a complex blending of pre-compiled animations facilitated by an animation tree. This is problematic because it restricts a user's arms and hands to a finite number of pre-determined movements, and is not representative of the fine movements observed in endoscopic procedures. The apparent lack of support for an auxiliary camera(s) in the pre-existing rendering pipeline also makes difficult the modelling of a second camera as would be used to represent an endoscopic camera whose output is displayed on a monitor in an operating room setting. While the existing architecture did make allowances for the use of mirrors through the use of portal theory, this is still handicapped by dependence on the player camera's perspective.

The initial effort was focused on the implementation of a second in-game camera with the intent of using it to model an endoscopic camera in a clinical setting. The design of this solution focused on leveraging the traversal of the scene graph is traversed by the rendering portion of the engine. By way of its design, the engine is optimized to render portals (windows, mirrors, doorways) only when they are in view of the player's camera. The equivalent portal for our purpose is when the player is observing the monitor on which the images of the endoscopic camera are being displayed. In the same way that portals are viewed by rendering the scene from the perspective of each portal encountered in the scene graph, before collapsing the resulting views back into the perspective of the player's view point, our method for adding an auxiliary camera(s) works by rendering the scene not from the perspective of our new type of portal but instead from the perspective of the endoscopic tool to which the imaginary camera is bound. The results of this rendering pass are then stored in an auxiliary render target and not to the back buffer as with the other passes. The stored view of the camera pass within the render target is then available to be used as a texture that is then applied to any number of surfaces within the simulator, in our case the target surface is that of an object representing a computer monitor's screen.

The second goal in researching EE's viability as a platform for the development of endoscopic surgical simulators (ESS) was to look at how full and yet efficient control of a surgical avatar's upper arm and hands could be accomplished. For this undertaking the possible application of Inverse Kinematics (IK) in programmatically computing avatar joint manipulation in realtime was researched. One of the challenges faced when resorting to the manipulation of individual joints (Forward Kinematics) is that it proves to be quite a taxing process for the user and detracts from the natural nature of the movement we want to recreate. Inverse Kinematics allows the end-effector (in our case the hand or tool position of the avatar) to be set to the desired position while utilizing IK to compute the required position of the connected joints that satisfies the request. Typical applications of IK in video game environment are generally limited to the trivial computation of foot placement on uneven terrain or the act of an avatar reaching out to pick up an object. By contrast, the use of IK needs to be greatly extended to support the dynamic nature of the input representing the desired position of the avatar's hand. The approach used in this implementation was to use the Leap Motion Device (LMD), an input device that tracks the position and orientation of a user's hands within the field of its sensors. Modifying the EE to allow for the input from the LMD, the position data collected during each pass of the animation tree of the avatar is applied to a modified version of the pre-existing IK portion of the animation engine (AE). The modified IK module uses triangular IK to calculate the position of two adjacent joints of a third, end-effector joint. Using the palm position as tracked by the LMD the end-effector is configured to be the wrist joint, thus using IK to solve for the position and rotations of the connected elbow and shoulder joints. This allows for the realtime mapping of a user's hand movements as captured by the LMC to that of the wrist, elbow and shoulder joints of the avatar within the simulator. Figure 2.4 illustrates the results of the early research conducted.

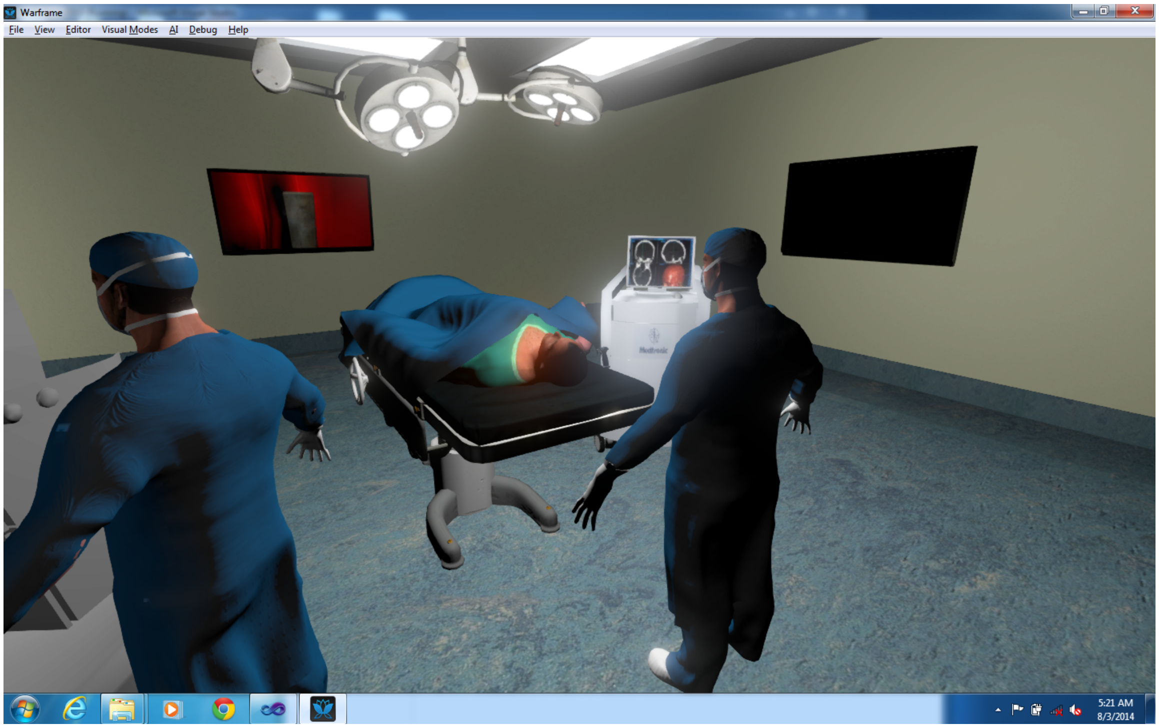


Figure 2.4: Figure showing initial OR recreation using Evolution Engine

The conclusion of the initial research on existing game engines revealed that while it is possible to recreate a virtual representation of an OR for the use of a surgical simulator, there is a clear need to investigate the efficacy of endoscopic surgical simulators in being able to improve clinically relevant skills that are transferrable to an OR. This requires the implementation of a targeting tasks-based simulator in order to evaluate the lower level skills and subtasks that can be decomposed from the complex surgical procedures we aim to simulate. The closed-source nature of existing game engines and their focus on higher level development makes development of a targeting tasks simulator capable of answering these research questions more difficult than a purpose-built platform.

2.5 Fitts' Law: Targeting Performance

The crucial objective in designing and validating any simulator is the capture and analysis of user performance, using a robust and objective means for measuring a user's skill level. The widely accepted method for evaluating human performance as it relates to targeting tasks is Fitts' Law [34]. Fitts' paradigm provides a performance metric used to relate speed and accuracy for targeting tasks against an index of difficulty (ID). The ID is the relation between the target area size and the distance to the target from the initial starting point. The metric we are concerned with is the index of performance (IP) which is the relation of the ID and the movement time (MT) [35]. The result, and what Paul Fitts proposed, is there exists a tradeoff between speed and accuracy that follows a curve along the user's IP for a given targeting task having a set ID [7]. Figure 2.5 illustrates the classic setup of Fitts' 1D targeting task, relating target amplitude/distance *A* and target width *W*.

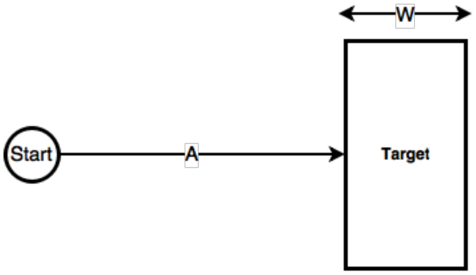


Figure 2.5: Figure showing Fitts' classic 1D pointing task

The ID for a given task is evaluated using Equation (2.1) and expresses the difficulty of a task in *bits*. The terms *A* and *W* in the equation represent the distance to the target and the target area (width in 1D) respectively and are illustrated in Figure 2.5. It becomes quite apparent that increasing the target size (*W*) or decreasing the distance to the target (*A*) will result in a lower ID value, indicating that the task is less difficult than one where the target size is small(er) and/or a greater distance from the starting point.

$$ID = \log_2(\frac{A}{W} + 1) \tag{2.1}$$

The original equation used for calculating the IP that Fitts proposed simply related the ID and MT terms which gives a metric with *bits/s* or *bps* as given by Equation (2.2). This rather simplistic approach models the performance of simple targeting tasks by suggesting that experienced users will have a shorter MT and therefore higher IP than novices. The equation also suggests that it can predict a user's MT for a given ID, which has important implications in a number of applications and fields including robotics and human-computer interaction (HCI) research.

$$IP = \frac{ID}{MT} \tag{2.2}$$

A caveat of the equation is that it evaluates the user's IP based on what they were asked to do, rather than what they actually did[28]. The preferred solution and what the ISO 9241-9[13] standard is to modify Fitts' original equation for determining a problem's ID by using the effective target size and distance, A and W respectively. The effective target distance/amplitude (A) is simply the mean distance that users must move to hit the target, while W_e is given in Equation (2.3) and represents the effective target size.

$$W_e = 4.133 * SD_x \tag{2.3}$$

Assuming a normal distribution of strike points around the target's position, Equation (2.3) scales the effective target size so that 96% (or 4.133 standard deviations)[17] of selected points will be considered as falling within the target area. This modification to the calculation of ID allows us to more closely model the task that the user actually performed as compared with what they were asked to do.

$$\begin{gathered} MT = a + b * ID_e \\ MT = a + b * \log_2(\frac{A_e}{W_e} + 1) \end{gathered} \tag{2.4}$$

The widely accepted form of Fitts' original equation that is used (and used in this thesis according to convention) is shown in Equation (2.4). The terms a and b are determined through empirical analysis using linear regression, with *1/b* representing the performance coefficient that allows for the comparison and differentiation between users of varying skill levels.

The ability to measure a user's performance for a given targeting task as it relates to the difficulty of the problem is of great significance as it allows us to score user performance objectively. Fitts' methodology also provides a means for validating a simulator's construct validity through the identification and differentiation of experts and novices. The choice of Fitts' Law for the objective evaluation of human performance lends itself well to the ETV procedure we intend to simulate as it can be decomposed into a hierarchy of subtasks that ultimately can be reduced to a set of targeting tasks. The choice of target shape also lends itself well to Fitts' methodology as the ellipsoid target's eccentricity is directly correlated to the target area size used in Fitts' ID.

Chapter 3

Requirements and Modes of Operation

3.1 Overview

The requirements of the ETTS can be categorized into three different modes of operation based on the desired use cases. These modes of operation or use cases are deconstructed into: A Scenario Designer, Targeting Simulator, and an Evaluation and Debriefing Tool. The way in which the program behaves depends on the context in which it is being used; for example when used as a Scenario Designer, the user should have the ability to import and place objects into a scene, set lighting and material properties, and have all of the available tools and options at their disposal. Running in simulation mode, the objective is to ensure that the user is only able to complete the trials as laid out by the scenario creator and the design of the scenario can't be modified to provide an unfair advantage. When used as an Evaluation and Debriefing Tool, the goal is to be able to examine a completed session by importing the acquired data and play it back for review as a training aid and to evaluate a user's performance. As per the three different modes of operation discussed above, we view the system and its requirements from the perspectives of three actors who from herein will be referred to as: *The Designer*, *The Candidate*, and *The Evaluator*.

3.2 Scenario Designer

3.2.1 Scenario Creation

The scenario designer mode of the ETTS system is used for the generation of specific scenarios in which simulation candidates will complete and be evaluated against. The two core aspects of the scenario designer include configuring the scene by placing objects within it and the configuration of environmental conditions such as lighting and material properties. A typical scene will contain a set of objects that are visualized by importing meshes and placing them within the virtual world created for the simulation. Objects within the scene can be defined as light sources, decorations (meshes that are not designated as a light, camera, target, or tool), targets, or tools (object used for pointing to a desired location in space) used for completing targeting tasks.

At the designer's disposal is the ability to configure ambient and diffuse material properties on objects defined within the scene. Additionally objects can be bound to a selectable input device such as the Leap Motion device or a gamepad controller. If desired, textures can be bound to an object's material which will override any ambient or diffuse lighting presets. Objects have the option of being designated a tool or target which causes them to be used in the calculation of the candidate's overall performance score when completing the designed trial. The shader effects that are used in the scene can be changed on a scene-by-scene basis to augment how the scene will be rendered. Shader effects of particular interest include global illumination, Depth of Field (DOF), and Diffuse vs. Ambient. The use of orthographic slice views can also be toggled on or off depending on whether the designer wishes to make the added information available to the candidate. Figure 3.1 illustrates the use case diagram of the ETTS system from the perspective of the designer.

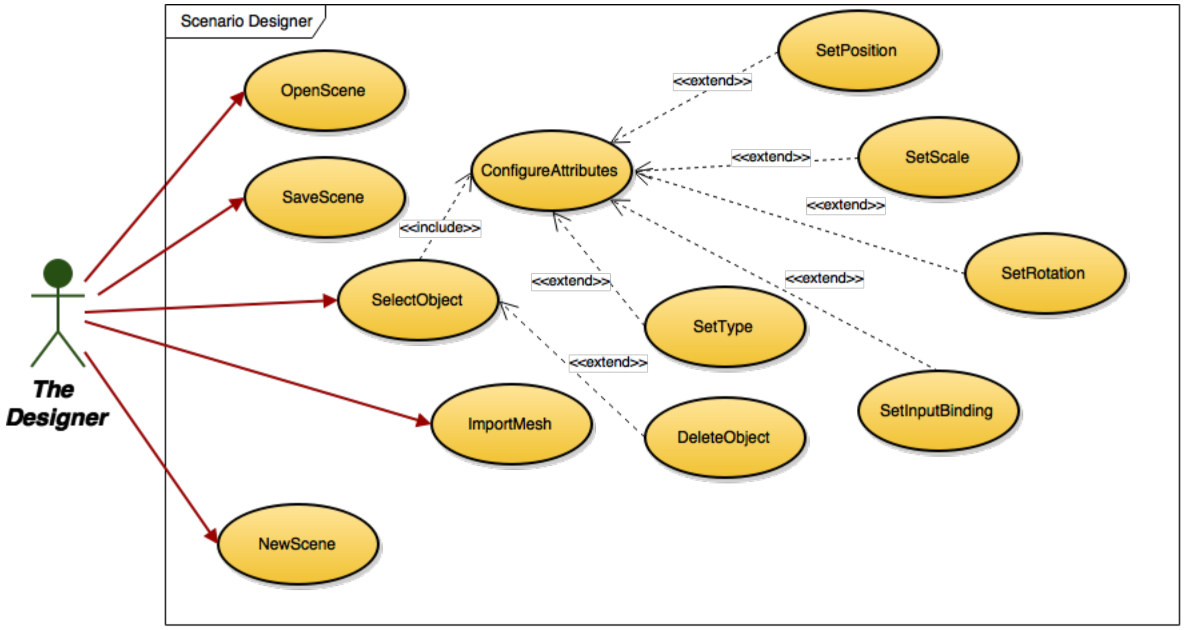


Figure 3.1: Scenario Designer Use Case Diagram.

3.2.2 Trial Dataset Creator

The goal with the ETTS system is to simulate numerous different scenarios to candidates for the training and evaluation of performing targeting based tasks; as such, there should be a means for creating a randomized dataset of scenarios. The task of creating a single scenario can be tedious and time consuming and as such it is desirable to enable a designer to create a batch of scenarios at once. Setting the initial and boundary conditions of a template scenario, the designer should then be able to create a collection of scenarios whose configurations have been randomly varied according to the conditions set. Each of the placeable objects in the scenario creator can have an initial position, scale, and rotation defined. Additionally boundary conditions for the position, scale, and rotation of objects can be set to define the maximum and minimum values for these attributes. Figure 3.2 illustrates the use case diagram of the ETTS system from the perspective of the designer for the batch generation of trial datasets.

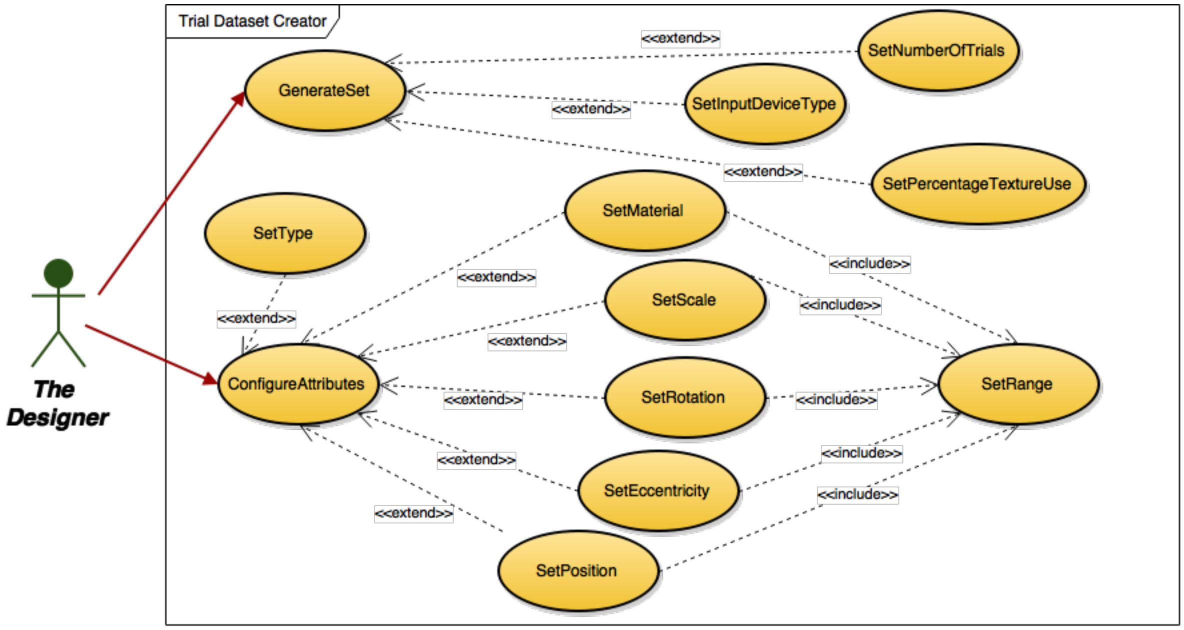


Figure 3.2: Trial Dataset Generator Use Case Diagram.

3.3 Targeting Simulation

The goal of the targeting simulation is to provide a varying array of targeting task-based scenarios to candidates for training and evaluating performance. Scenarios should be clinically relevant, requiring users to perform targeting tasks on objects that accurately portray anatomical structures that might be encountered in a clinical setting, yet must not be restricted to such cases. The candidate must be able to interact with the simulator to augment the viewpoint of the scene and objects contained within it, as well as provide a form of input for the control of a targeting device or tool. If enabled by the designer, the candidate will have at their disposal the control of orthographic slices that can be controlled to identify an object's shape and orientation by traversing the slices in three dimensions. The use of orthographic slices is meant to be analogous to what would be deployed in a clinical setting where preoperative scans are used for surgical planning and navigation.

3.3.1 User Input

Of great interest is the evaluation of user performance as it relates to a user's level of experience with targeting tasks analogous to those found in a clinical setting. Further interest lies in researching the impact on user performance attributed to different input modalities; specifically through the use of a gamepad controller as compared with a Leap Motion device which tracks the position and orientation of a user's hands within the sensor's field.

Leap Motion

Input from the Leap Motion controller should be captured to allow both single and two-handed interaction with the simulator. Position and rotational information should be taken into account for controlling the position of a targeting tool, camera placement and orientation. Mapping of hand motion should also be flexible to allow reconfiguration to suit a candidate's preferences with respect to direction of motion (i.e look inversion and movement depending on whether upward motion should cause camera to move up or down).

Game Pad Controller

The gamepad controller should allow the user to control the position and rotation of both the targeting tool and camera within the simulator. Input from the controller should be used to allow full control over simulator controls including but not limited to: starting and stopping of trials, access to help screens, and advancing through trial dataset. The mapping of the controls on the gamepad controller must also be flexible to allow users to change their preferences with respect to look inversion (both in X and Y directions). Stick and trigger sensitivity is another parameter which should be configurable to match a candidate's personal preferences.

The goal in designing any user interface, especially one concerned with evaluating a user's performance should be to provide as natural and intuitive a mapping as possible. One of the goals with evaluating performance across different input modalities is to establish what serves targeting based tasks best, not just in simulator performance but transferrable skills.

3.3.2 Data Acquisition

An important aspect of the simulator is the collection of data generated through user trials for the purpose of evaluating their performance. Information about the candidate such as user ID should be recorded in an anonymous format. Simulator parameters and attributes should be recorded once a candidate begins a new trial and, written to a file upon completion for use in post-processing, analysis, and evaluation. Data should be collected in an efficient fashion that contains all pertinent information, while optimizing space complexities to minimize storage and memory requirements. The data collected should allow for the reconstruction and playback of a candidate's actions for a given trial session. It should also possess the necessary information for the analysis of performance as it relates to speed and accuracy in both positional and rotational errors. The format in which the data is stored should be easily read and available to be imported into a spreadsheet program for further analysis outside of the ETTS environment. Figure 3.3 illustrates the use case diagram of the ETTS system from the perspective of the candidate.

3.4 Analysis and Evaluation

The third mode of operation for the ETTS system should be used in the analysis and evaluation of a candidate's performance. Valuable information can be attained through the review of a candidate's trial session and as such a requirement of the ETTS should be to include some means of playing back the sequence of events recorded during a candidate's trial. Key

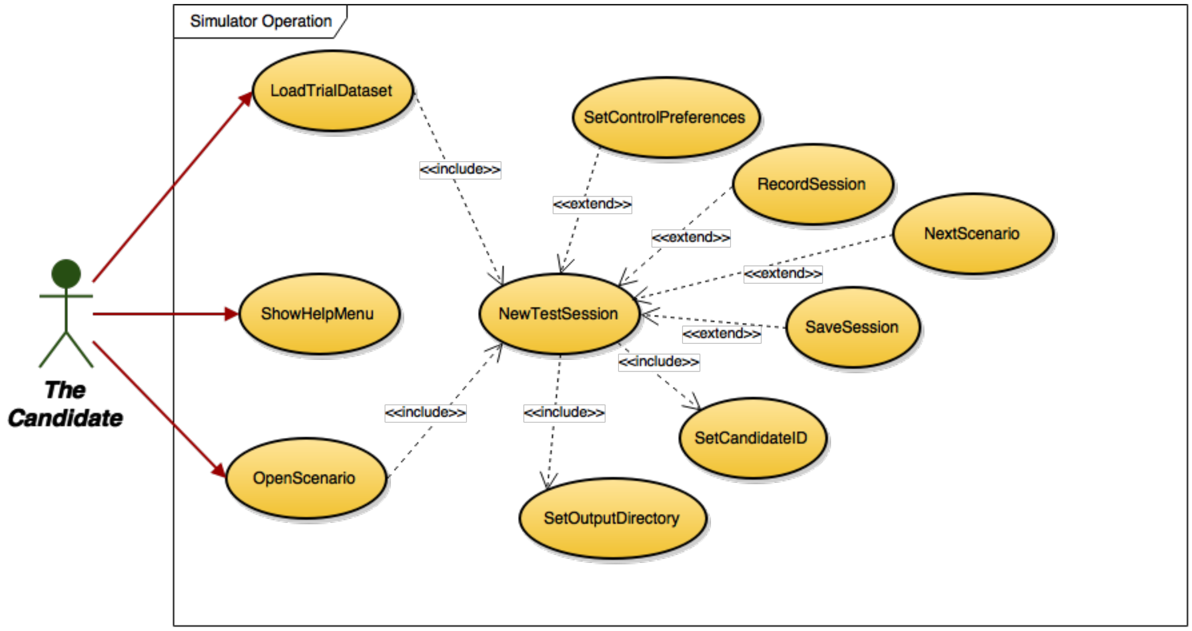


Figure 3.3: Simulator Use Case Diagram.

to the ETTS system is the evaluation of user performance in targeting tasks; both in accuracy and speed, evaluated using Fitts' Law to assign a score. Providing the user with a graphical representation of their performance must therefore be a requirement of the system.

3.4.1 Scenario Playback and Teaching

The ability to review the actions taken by a user during a simulation trial can be an invaluable teaching tool evaluating user performance and approach, not unlike an athlete reviewing video to learn from their mistakes. We therefore propose that visual playback of a candidate's attempt for a given trial be available for review. The candidate should be able to load a visual record of their session and have the following capabilities: play, pause, seek forward and backward, and be able to interact with the scene; changing the viewer's perspective at any point during the playback and even manipulate the objects within the scene (such as tool placement).

3.4.2 User Performance Evaluation

The metrics on which a candidate's performance is evaluated are based on speed and accuracy, where accuracy relates to the error in position and rotation between the target and the candidate's placement of tool(s). Using Fitts' Law, user performance can be resolved to a single score identified as their index of performance. Requirements of the ETTS must therefore include an ability to present the user with their score as evaluated using Fitts' Law's index of performance. In conjunction with the scenario playback module described above, the user should also be able to review their error in position and rotation as a function of time in the form of a graph. The graph should be synchronized in time with the visual playback of the candidate's trial session to provide a time indexed account of accuracy that can be seen in the visual playback of the candidate's actions. Figure 3.4 illustrates the use case diagram of the ETTS system from the perspective of the evaluator.

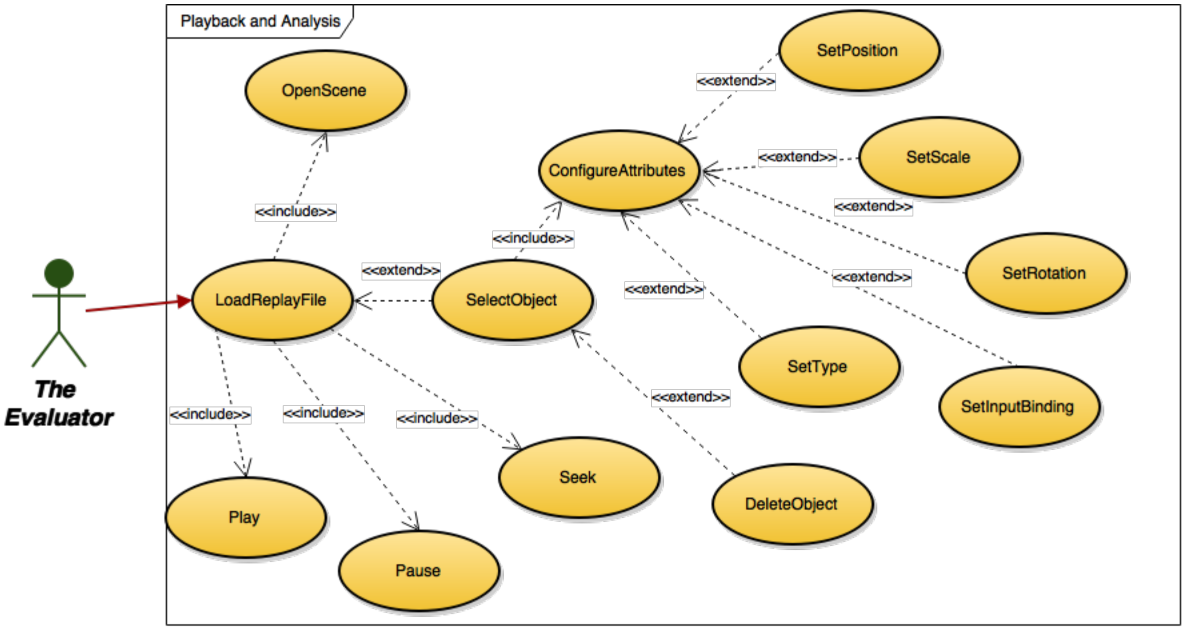


Figure 3.4: Playback and Analysis Use Case Diagram.

Chapter 4

Targeting Simulator: Render Engine

4.1 Targeting Tasks Simulator: Overview

The goal in developing a targeting simulator was to design an environment that allowed for the simulation and evaluation of user performance across a wide array of scenarios and differing input modalities. While such a program could be made using existing Game Engines such as the Evolution Engine developed at Digital Extremes or the widely adopted Unity Engine, developing a simulator from the ground up with a specific focus on targeting tasks afforded us with certain advantages.

A big motivation for developing the ETTS was to have a platform that allowed for the experimentation of different input modalities such as game pads, voice, and the Leap Motion device for the purpose of evaluating user performance across these different input devices. The full range of control in the design of ETTS also allows for the implementation of various visual effects and full control over the rendering pipeline. This approach allows for the rapid prototyping and testing of features directly related to the targeting tasks being researched; without the hinderance of licensing proprietary features or working around non-open-source implementations that make Engine modifications cumbersome.

4.2 Development Environment

The ETTS is implemented for use on Apple's OS X operating system. The language used for the ETTS is Objective-C, a variation on the traditional C programming language. The choice to use Objective-C is based on a desire to learn a new language as well the ease in which C and C++ code can be incorporated into the program's framework. The language uses a unique syntax that lends itself well to the strict use of object oriented programming practices; which is crucial for the good design of any large software project. Developing in the Xcode IDE also allows for the easy creation of Graphical User Interfaces; an important component in the design of a Virtual Reality (VR) based simulator.

The primary means for versioning is a locally hosted git repository with a mirrored repository on a locally hosted web server that hosts an integrated project management suite for tracking changes, bugs, and features. The use of git facilitates collaboration and the tracking of changes.

4.3 Architecture

4.3.1 Objective-C Specific Considerations

One of the unique properties of Objective-C as compared to other C like languages is that it forces the developer to follow a strict adherence to the object oriented design principle when structuring their project. The way in which this is achieved is by treating everything as an object, thus ensuring a highly organized and structured implementation. Objective-C also makes use of Delegates that are used whenever an event is generated. Memory management uses a method of reference counting which relies on objects to track their references to know when the occupied memory can be freed.

4.3.2 Modular Design

The structure of the ETTS system is designed to be highly modular both for organization and maintenance as well possible recycling of code. There are two forms of modular design employed within the architecture of the ETTS: Classes and Frameworks. Two frameworks were developed as separate projects and compiled into reusable frameworks namely: The Game Pad Manager Framework and The Speech Controller Framework. These frameworks are highly reusable modules that can be dropped into the ETTS application not unlike a plugin or added feature. OpenGL is used for rendering the simulator view with shaders written in GLSL.

The role of the Game Pad Manager framework is to handle the input from a USB Gamepad controller (in this case Xbox 360 Controller developed by Microsoft) and organize the input events into a data structure that can be read and used by the main ETTS program.

The Speech Controller framework serves to concentrate the functionality of Apple's speech recognition system into a module that aggregates the relevant functions into an easy to use interface for translating voice commands into actionable function calls. This module is particularly useful alongside the Leap Motion interface, as it allows for hands free control of the simulator while using both hands to complete targeting tasks.

Rendering of the simulator view is done using OpenGL; an open source, widely adopted and supported cross-platform graphics language. OpenGL makes use of accelerated hardware to render the objects and scene that form the Virtual Reality (VR) representation of the scenario being carried out by the user. While OpenGL handles the rendering instructions on the GPU, the architecture of how a scene is organized and stored along with the flow of the render pipeline and any user interaction must be handled by the developer. Using the next generation of OpenGL which no longer uses a fixed pipeline allows flexibility in how scenes and objects are rendered and various effects applied, but it passes the burden of managing lighting, camera and scene matrices, and shaders onto the developer.

Both frameworks and all of the major classes are discussed in greater detail in subsequent sections below.

4.3.3 Front-End

The front-end of the ETTS uses a Graphical User Interface (GUI) designed using Apple's Cocoa API and tools. Behind the GUI sits a main Application Delegate whose role is to handle events generated by the various GUI controls. The Cocoa api also allows for programmatically generated interfaces and controls while removing the burden of having to write the code for firing the various events or polling for user input.

4.3.4 Back-End

The backend of the ETTS is primarily focussed on scene management and the rendering pipeline. There are a number of supporting modules that support the main OpenGL controller which include: Shader Bank, Material Bank, User Input Responder, Leap Motion Control, Game Pad Control, and Mesh Parser.

4.4 Render Pipeline and Flow

The flow of the render engine module of the ETTS system leverages several design properties to optimize performance and resource management while placing an emphasis on a modular design and variable path execution. The render engine makes use of OpenGL's programmable pipeline which in turn affords greater flexibility and control over the rendering process but comes at the *cost* of requiring the implementation of modules that handle shaders, lighting, object structures, and model-view and projection matrix operations.

A multi-pass rendering approach is used to allow for the use of multiple programmable shaders that can be stacked one after the other to produce a myriad of different effects. Each render pass loads a compiled shader program consisting of a vertex and fragment shader and configures the scene and shader variables for the specific pass. The result of each pass can be rendered to the back-buffer for direct display onto the screen or be stored to auxiliary render target that stores the rendered scene as a resource for display or sampling in subsequent passes as a texture.

The design of the ETTS' main user interface (see Figure 4.7) supports the use of four viewports with the option of using a single perspective viewport instead. Each viewport is rendered using the same multi-pass rendering approach as previously discussed with specific passes being skipped depending on the enabled shaders and the viewport's configuration. For example, when the orthographic views are used during simulation mode to provide orthographic slice information, diffuse lighting is disabled and thus requires only ambient illumination.

The use of the *dirty-flag* design pattern is popular among applications in game engine design and is used throughout the ETTS framework to optimize scene rendering performance. The principal behind the popular design pattern is to only draw what is needed and avoid unnecessary re-rendering of the same scene. This simplistic optimization is highly effective in reducing the load on CPU and GPU when the scene being rendered is static or goes unchanged for lengths of time. The approach works by maintaining a flag that when set, instructs the render engine that the scene/view is dirty and therefore needs to be re-rendered. Once the scene has been re-rendered, the flag is returned to a clean state and the render engine effectively returns to an idle state until new changes require the scene to be rendered once more. Examples that may produce a *dirty-flag* could include any change to a scene object's appearance or position, the location or orientation of the camera, changes to scene lighting, enabled shaders and or shading style. Figure 4.1 provides and abstracted overview of the flow and pipeline style approach used by the ETTS render engine.

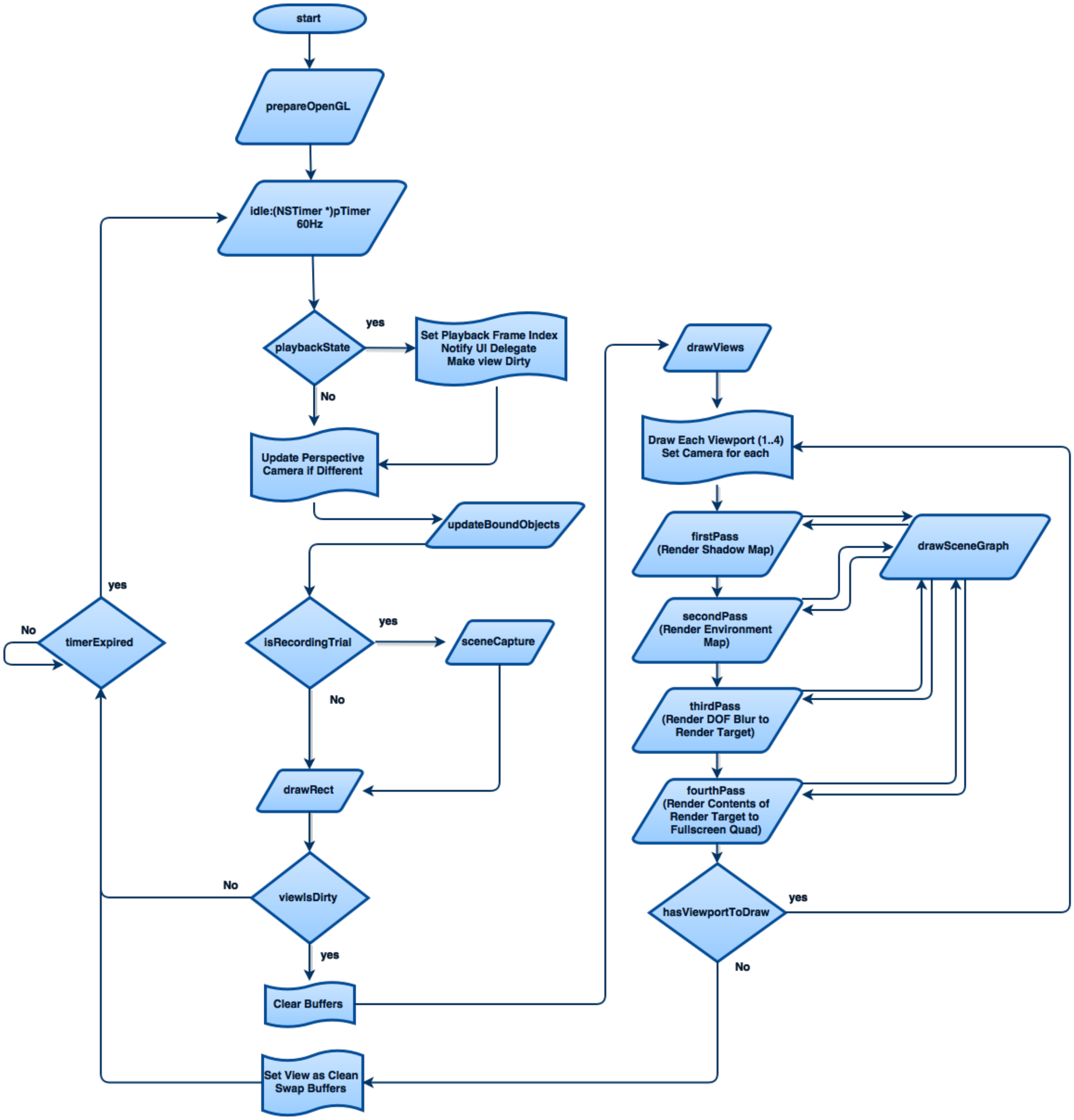


Figure 4.1: Abstracted Flowchart of the Render Pipeline and Flow.

The refresh interval of the render engine is driven by an internal event timer that is configured to provide a refresh rate of 60Hz or render 60 frames per second (fps). Each time an event is generated by the timer, the render engine begins a new cycle by updating scene object states if in playback mode (Section 4.8.4) and retrieving new user input if available from any attached devices that are bound to objects being rendered in the scene. If the ETTS is being used in its simulation mode of operation, a snapshot of the scene's current state is appended to the current trial session data structure (see Section 4.8.3 for further information). If the scene has been marked dirty, the current colour and depth buffers are cleared before the scene data is rendered again.

Listing 4.1: Configuration of Perspective Viewport

```
switch (vp) {   //Switch/Case used to decided which viewport to render to (ie. multiple passes)
  case 0:
    if ([self numberOfViewports] == 1)
    {...}
    else
    {   //Perspective View - Placed Top/Right
      glViewport(quadW, quadH,quadW - 4,quadH - 4) ;
      fullScreenQuadWidth = quadW;
      fullScreenQuadHeight = quadH;
      glScissor(quadW + 2, quadH + 2,quadW - 4,quadH - 4);
      glEnable(GL_SCISSOR_TEST);
      glClearColor(0.01,0.01,0.01,1.0) ;   //Smokey Black
      glClear(GL_COLOR_BUFFER_BIT);
      [self setLookAt:mainCamera Upx:UPx Upy:UPy Upz:UPz];
      [self setProjection];
      currentViewport = PERSPECTIVE;
    }
    glEnable(GL_CULL_FACE);
    break;
```

Listing 4.1 illustrates how each viewport, starting with the perspective view, is configured before starting the multi-pass rendering of the scene from that particular viewport's camera. When all four viewport's are enabled (perspective along with orthographic top, front, and side), each is rendered into a defined section of the back-buffer, preventing one viewport from overwriting another. Parameters such as the position and orientation of the camera along with the projection matrix are configured before the scene is prepared for rendering.

Listing 4.2 similarly shows how each of the three orthographic viewport's are configured in contrast to the perspective view. The construction of a orthographic view frustum and camera placement varies slightly from its perspective counterpart. The orthographic slice feature as observed in Figure 4.7 requires that back-face culling be disabled so that these faces can be seen as the slice planes move through the target meshes.

The main method used by the render engine to traverse and draw the scene graph is the *drawSceneGraph* method which takes the id of a loaded shader program as its only argument and is used by all passes to render the contents of the scene graph. Listing 4.3 illustrates a subsection of this function that is responsible for preparing each object and sending the prepared buffers to the GPU for rendering.

In order to optimize render performance of the ETTS system, required array buffers are allocated and filled upon object allocation. This design choice has the trade off of occupying more memory but has the benefit reducing the time, in the main render loop of the engine, to prepare the buffers of each object. As each object is loaded to be rendered in the scene, the

Listing 4.2: Configuration of Orthographic Viewport

```
case 1:     //Orthographic Top View - Placed Top/Left
  currentViewport = TOP;
  glViewport(2, quadH + 2, quadW - 4,quadH - 4) ;
  glScissor(2, quadH + 2,quadW - 4,quadH - 4);
  glEnable(GL_SCISSOR_TEST);
  glClearColor(0.01,0.01,0.01,1.0) ;   //Smokey Black
  glClear(GL_COLOR_BUFFER_BIT);
  zoomValue = [delegate_handle topLeft_zoom];
  w = (int)(quadW / zoomValue);
  h = (int)(quadH / zoomValue);
  cX = -[delegate_handle topLeft_origin].x;
  cY = 1000;
  cZ = [delegate_handle topLeft_origin].y;
  [self setOrthoViewLeft:-w Right:w Bottom:-h Top:h];
  [self setLookAtEYEx:cX EYEy:cY EYEz:cZ CENTERx:cX CENTERy:0 CENTERz:cZ
      Upx:0 Upy:0 Upz:-1];
  glDisable(GL_CULL_FACE);
  break;
```

prepared buffers are bound and used to provide the object's vertex, vertex normals, and texture data to the GPU pipeline. Shader variables are set to point to the appropriate texture resources for sampling in the shading phase of the render pipeline. Once the object's configured buffers are passed along to the GPU's pipeline using the *glDrawArrays* function call, the object's buffers are unbound before repeating the process for the remaining scene objects.

4.5 Scenes and Scene Objects

The role of scenes in the ETTS is to serve as an object that contains all of the pertinent information needed to construct a virtual world within the simulator. Whenever a draw call is made on the backend of the system, the scene graph is traversed; each object being prepared for rendering with their appropriate properties and transforms being defined. Components of the scene can be categorized into three groups based on functionality: Lights, Camera(s), and Objects. All three types are capable of being represented visually in the simulator environment, however outside of debugging purposes it should not be necessary to draw the camera nor the light(s) within the scene.

It is important that the design of the scene framework be efficiently constructed to minimize impacts on performance as the scene plays a critical role in the rendering pipeline and has the potential to slow down the rendering process if not structured properly. For the sake of portability and reusing scenes (not having to configure every time a new session is started), the scene data structure is designed to be serializable, making it possible to save the state of a scene for later use or even use it as a template for a set of derived scenes. The support of transparency effects by ETTS requires that the objects within the scene data structure be easily sortable to aid in efficiently sorting objects by depth/distance from the viewer to ensure proper render order.

Listing 4.3: Preparation of object buffers for submission to GPU for rendering

```
glBindVertexArray([currentObject getVAO]);
[currentObject getModelMatrix:mM];
[self updateUniforms:shaderID];
[currentObject prepareForRenderingNORMALS:vertexNormal];
[currentObject prepareForRenderingPOSITION:position];
GLint myLoc = glGetUniformLocation(shaderID, ”frontMaterial.ambient”);
glUniform4fv(myLoc, 1, objectAmbient);
myLoc = glGetUniformLocation(shaderID, ”frontMaterial.diffuse”);
glUniform4fv(myLoc, 1, objectDiffuse);
if ([currentObject isUsingTexture] && currentPass != SHADOW && ![[[
    currentObject attributes] valueForKey:@”globalIllum”] boolValue] &&
    [[[objectMaterial attributes] valueForKey:@”DiffuseMap”] boolValue]){
  GLint pos = glGetUniformLocation(shaderID, ”diffuseMap”);
  if (pos >= 0){
    glProgramUniform1i(shaderID, pos, 1);
    GLint uvID = glGetAttribLocation(shaderID, ”textureCoords”);
    [currentObject prepareForRenderingUV:uvID];
    GLuint diffTex = [[[[objectMaterial attributes] objectForKey:@”
        lowLevel”] valueForKey:@”diffuseMap”] unsignedIntValue];
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, diffTex);
  }
}
glDrawArrays(GL_TRIANGLES, 0, [currentObject nVertices]);
[currentObject cleanupBuffers];
```

4.5.1 Structure

The structure of a scene object within the ETTS system uses a mutable array for the storage of objects making up the scene. Objects can be one of three types: Entity, CameraObject, LightSource. Each of these classes maintains a set of attributes that are accessible in key value pairs. Identification of objects is made easy thanks to the built-in functionality of the Objective-C language that allows you to test the class type of an object before performing a dynamic cast of the object into its correct type. The use of a mutable array lends itself well to the need for sorting objects based on some pre-defined comparator (in our case depth/distance from the camera) and of course the ability to easily add and remove objects from the scene while still being easy to serialize.

All of the objects defined within the scene have their attributes, indexed by object uid, stored as key-value pairs in the scene structure. Both the Graphical User Interface (GUI) and the backend renderer reference this portion of scene data structure as a common model. The renderer iterates through the scene data array, retrieving objects for rendering that have been sorted into correct render order for accurately handling transparency. The elements that make up the view portion of the ETTS Model View Controller (MVC) framework directly reference the per-object attributes of the model.

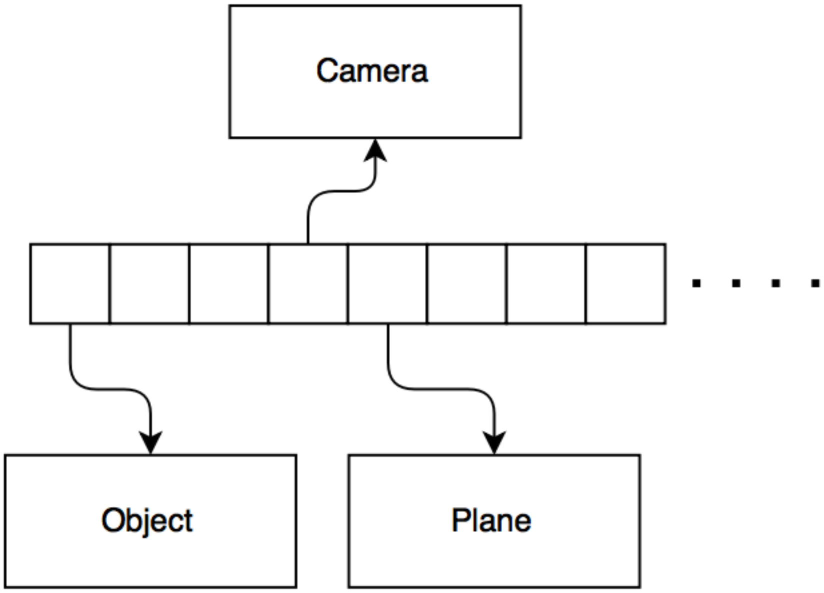


Figure 4.2: Figure depicting basic structure of scene graph using mutable array with indexes pointing to individual scene objects

Entity

Entity is a class of object that is used for any general purpose object in a scene that does not serve as a material, camera, or light source. In common with all objects, the Entity class includes two dictionaries storing constraints in one and attributes in the other. Entities have the ability to be designated as a target or tool or neither, in which case it is treated as a decoration within the scene. See the diagram below for the structure of the Entity class.

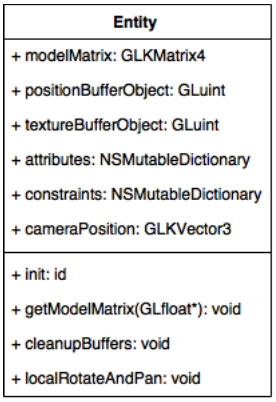

Entity
+ modelMatrix: GLKMatrix4
+ positionBufferObject: GLuint
+ textureBufferObject: GLuint
+ attributes: NSMutableDictionary
+ constraints: NSMutableDictionary
+ cameraPosition: GLKVector3
+ init: id
+ getModelMatrix(GLfloat*): void
+ cleanupBuffers: void
+ localRotateAndPan: void

Figure 4.3: Entity class diagram

CameraObject

The CameraObject class is a special type of object that is used to represent the perspective view of the scene. Housed within the class are the methods and structures used for handling the view matrix and transformation functions. The attributes section includes parameters for the camera's gaze and up vectors which are not included in any other class of objects. Figure 4.4 details the architecture of the CameraObject class structure.

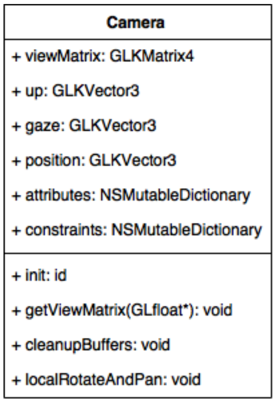

Camera
+ viewMatrix: GLKMatrix4
+ up: GLKVector3
+ gaze: GLKVector3
+ position: GLKVector3
+ attributes: NSMutableDictionary
+ constraints: NSMutableDictionary
+ init: id
+ getViewMatrix(GLfloat*): void
+ cleanupBuffers: void
+ localRotateAndPan: void

Figure 4.4: CameraObject class diagram

LightSource

The LightSource class is another special type of object that is used for the modelling of light sources within the scene. They are a fairly primitive type of object, not needing any specific attributes for visualization. As with other object types, transformation matrix operations are handled and stored by the LightSource object. In addition to setting intensity of the light source, the colour and biased projection-view matrix (used in shaders for directional lighting) can be defined. The class structure of the LightSource type is illustrated in Figure 4.5.

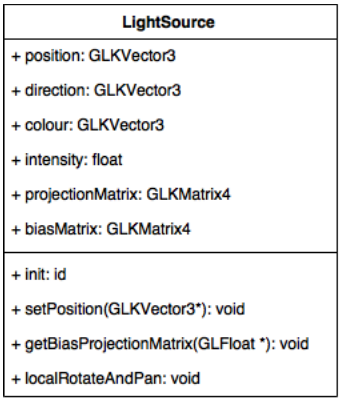

LightSource
+ position: GLKVector3
+ direction: GLKVector3
+ colour: GLKVector3
+ intensity: float
+ projectionMatrix: GLKMatrix4
+ biasMatrix: GLKMatrix4
+ init: id
+ setPosition(GLKVector3*): void
+ getBiasProjectionMatrix(GLFloat *): void
+ localRotateAndPan: void

Figure 4.5: LightSource class diagram

4.5.2 Serialization

Serialization is the process of taking the representation of a complex data structure in memory and converting it into a serialized stream of ascii or binary data that can then be written out to a file, thus preserving a data structure's state. The process of serializing the scene data structure of the ETTS system is used so that the information stored in a scene can be preserved for later use. As discussed in the structures section above, scene data is stored in a mutable array where the elements of the array are occupied by objects of one of the three mentioned types (Entity, LightSource, and CameraObject). The order in which these objects are stored in the array is inconsequential to the serialization process as they will be re-sorted into correct order upon extraction when imported for use. An example of a serialized scene file encoded for storage using JSON is illustrated in Listing A.1 in Appendix A.

The method used for serializing the scene data into a stream of encoded ascii characters works by leveraging the key-value storage of object attributes within the scene structure and uses JSON encoding to save the result to disk. The function below in Listing 4.4 illustrates the process by which the scene is serialized. The function works by first allocating two arrays; one for scene data and another for object constraints. Iterating through all of the scene objects first, object attributes and constraints (stored as key-value pairs in their own dictionaries within the objects) are appended to their respective arrays. The same process is repeated for the camera object and for the materials, represented as material objects stored in a bank (see section on Material Bank). The resulting structure (still only in memory) is two populated arrays which are then added as objects to a temporary dictionary under the keys sceneData and constraints. The data is then serialized using JSON encoding with the builtin serializer.

Listing 4.4: JSON Serialization of Scene

```
-(NSData*) saveScene: (NSError*)error
{
    NSMutableDictionary *dic = [[NSMutableDictionary alloc] init];
    NSMutableDictionary *sceneData = [[NSMutableDictionary alloc] init];
    NSMutableDictionary *constraints = [[NSMutableDictionary alloc] init];

    for (NSUInteger i = 0; i < [sceneObjects count]; i++)
    {
      //Check if object is of type Entity
        if ([[sceneObjects objectAtIndex:i] isKindOfClass:[Entity class]])
        { //Perform Dynamic cast of object into Entity to extract
          //attributes and constraints
        Entity *object = [sceneObjects objectAtIndex:i];
        [sceneData setValue:[object attributes] forKey:[[object attributes] objectForKey:@"uid"]];
        [constraints setValue:[object constraints] forKey:[[object attributes] objectForKey:@"uid"]];
        }
    }
    //Add Camera's attributes and constraints
    [sceneData setValue:[mainCamera attributes] forKey:@"C-0"];
    [constraints setValue:[mainCamera constraints] forKey:@"C-0"];

    for (NSUInteger i = 0; i < [self.materialBank count]; i++)
    {
      //Add Material attributes and constraints
        Material *m = [self.materialBank materialForSlot:i];
        [sceneData setValue:[m attributes] forKey:[NSString
            stringWithFormat:@"Material-%lu", (unsigned long)i]];
        [constraints setValue:[m constraints] forKey:[NSString
            stringWithFormat:@"Material-%lu", (unsigned long)i]];
    }
    //create the hierarchy that will be serialized by adding two
        dictionaries
    [dic setValue:sceneData forKey:@"sceneData"];
    [dic setValue:constraints forKey:@"constraints"];
    //Perform JSON Encoding and return result
    NSData *jsonEncodedData = [NSJSONSerialization dataWithJSONObject:
        dic options:NSJSONWritingPrettyPrinted error:&error];
    return (jsonEncodedData);
}
```

4.5.3 Extraction and Importing

The extraction and importing process of saved scene files works to reverse the serialization process discussed in Section 4.5.2; recreating the serialized scene and objects within it. The architecture of the ETTS is heavily influenced through the use of dictionaries to store object attributes as key-value pairs. This design lends itself well to the serialization and extraction processes, leveraging the key-value storage nature of the JSON encoding used for the saved scene files.

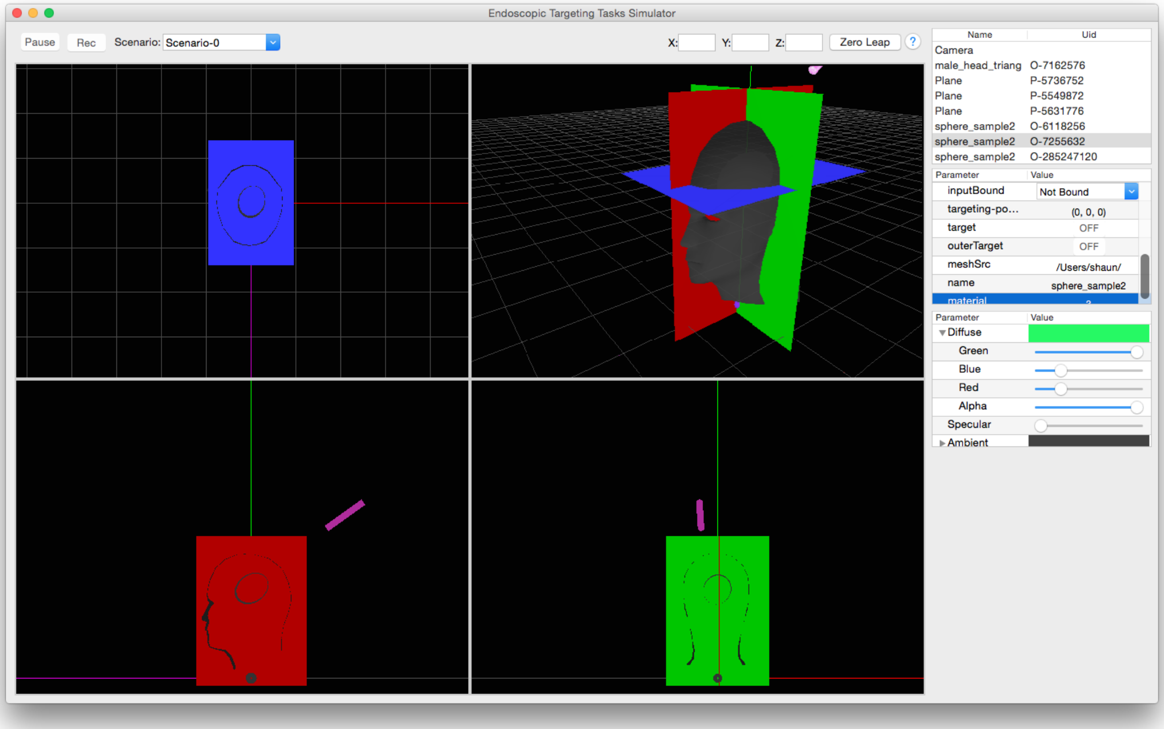
Each of the core object types (Entity, Camera Object, and Light Source) use dictionaries to store object attributes and constraints. Saved scene files are imported by parsing through the scenario hierarchy through the separation of scene data and constraints, where scene data stores scene objects and their attributes. Object types are identified based on their prefix and sorted into one of five designations: P-Plane, O-Object, C-Camera, L-Light, and M-Material; each having their own specific subset of attributes and related constraints.

Object extraction follows a similar process to that of importing mesh files and uses the OBJ parser discussed in Section 4.10. The distinction being the application of the initial conditions and constraints imported from serialized object state rather than using default settings.

Camera, Light, and Material extraction is a simpler case, only requiring the need to allocated the appropriately typed objects and set their attributes and constraints. Materials have the added requirement of ensuring that allocated materials are inserted into the same slot of the material bank to ensure objects reference the same material properties.

4.5.4 Depth Sorting

The support for rendering objects of variable degrees of translucency requires extra consideration in the design of the rendering engine of the ETTS. The main concern lies with the order in which objects are drawn into the back buffer so that objects positioned behind translucent ones aren't skipped over due to improper ordering and z-buffer testing. One possible solution, which has been implemented in the design of the ETTS renderer is to sort the objects within the scene, providing a back to front draw order. This results in drawing the most distant objects from the camera first, while those lying closest to the camera's perspective being drawn last. The figure below illustrates the need for correct depth sorting; when no depth sorting is applied, the solid coloured orthographic slice planes that intersect the translucent manikin head are clipped and not rendered.



Scenario: Scenario-0

Name	Uid
Camera	
male_head_triang	O-7162576
Plane	P-5736752
Plane	P-5549872
Plane	P-5631776
sphere_sample2	O-6118256
sphere_sample2	O-7255632
sphere_sample2	O-285247120

Parameter	Value
inputBound	Not Bound
targeting-po…	(0, 0, 0)
target	OFF
outerTarget	OFF
meshSrc	/Users/shaun/
name	sphere_sample2
material	

Parameter	Value
▼Diffuse	🟩
Green	
Blue	
Red	
Alpha	
Specular	
▶Ambient	■

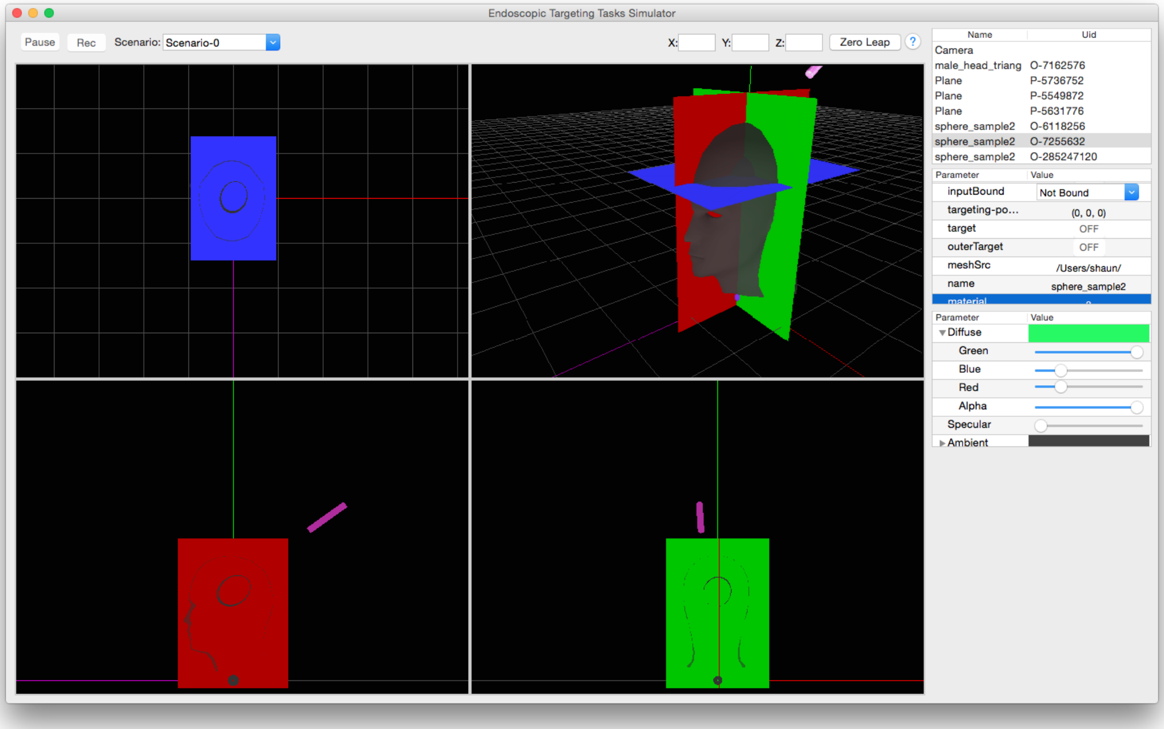
Figure 4.6: No application of object-based depth ordering. Notice how the planes that are intersecting the manikin head of the perspective view are not drawn within the bounds of the translucent mesh.

The naive approach for setting the render of objects would be to assign each a layer number that sets the draw order manually; obviously this is far from ideal as setting the order manually for large and complex scenes would be tedious and time consuming. The other need for a better sorting approach is that the order in which objects are drawn into the scene will ultimately depend on the position of the camera. The approach used in the design of the ETTS rendering system is to order objects in the scene relative to their distance from the camera.

Objects within a scene can be sorted in either alphabetical order (based on the assigned name) or as a function of the distance to the camera object. The Entity class is responsible for handling the comparator functions that provide these two sort variations. For the purpose of sorting objects based on their distance to the camera object, each one stores a vector representing the camera's position in world space. Upon each update to the camera or object's position, a new distance value is calculated using equation (4.1) below.

$$d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} \tag{4.1}$$

Using this singular value, distance-based scene sorts can be made quickly to ensure that objects are ordered back to front. The result of this form of scene sorting can be seen in the figure below; different from the previous example that did not sort objects into any particular order, the orthographic slice planes which lie furthest from the camera and now be seen through the translucent manikin mesh.



Name	Uid
Camera	
male_head_triang	O-7162576
Plane	P-5736752
Plane	P-5549872
Plane	P-5631776
sphere_sample2	O-6118256
sphere_sample2	O-7255632
sphere_sample2	O-285247120

Parameter	Value
inputBound	Not Bound
targeting-po...	(0, 0, 0)
target	OFF
outerTarget	OFF
meshSrc	/Users/shaun/
name	sphere_sample2
material	

Parameter	Value
▼Diffuse	🟩
Green	
Blue	
Red	
Alpha	
Specular	
▶Ambient	⬛

Figure 4.7: Same scene rendered using the discussed method for sorting the scene based on object depth. Notice how the planes that are intersecting the manikin head of the perspective view are now drawn within the bounds of the translucent mesh.

The one limitation with this method of scene organization for rendering transparent/translucent objects is that sorting takes place only at the object level. For smaller or sparsely populate scenes, this form of object-based sorting is suitable. It should however, be noted that for cases that involve large objects or densely populated scenes, other measures could be taken to segment the objects into smaller fragments that are themselves sorted for correct render order. The shortfall with object-based sorting is that while the centre of an object may be further away from the camera's position, it may have fragments that extend towards the camera and in front of objects labelled as being closer.

4.6 Shader and Materials Bank

4.6.1 Shader and Material Overview

The roles that shaders and materials play in the ETTS framework are crucial to the rendering of scenes. Shaders are programs written in a language that can be compiled and run by the Graphics Processing Unit (GPU) of the computer and are responsible for providing instruction on how objects should be rendered. Materials on the other hand are data structures that store information describing the material properties assigned to objects; properties which are passed along the rendering pipeline to the shaders to provide the necessary effects.

4.6.2 Shader Structure and Management

The graphics API used for the ETTS is OpenGL, a cross-platform library that provides hardware accelerated rendering support. Shaders for the ETTS system are written in the OpenGL Shading Language (GLSL for short) and are classified into two types of shaders; vertex shaders and fragment shaders.

Vertex shaders are the first stage of the shading hierarchy in the rendering pipeline and are responsible for taking in vertex, camera, lighting, and texture information and is used to apply the necessary transformations that convert the incoming structures into the proper frame of reference. Vertex shaders are not just limited to applying model-view and projection transformations, similar to C like programs they can be used to perform other operations such as configuring texture coordinates, depth values in camera space, or even calculating the projection of shadows from provided light source(s). Operations performed by the vertex shader are passed along as variables (such as floats and vectors) to the next stage of the shading process, the fragment shader.

Fragment shaders take the geometrical information passed from the vertex shader stage along with optional material information such as diffuse, ambient, and specular properties to compute the output fragment colour and depth values. The fragment outputted represents a pixel area defined by the rasterization process and is written to a bound buffer such as a back-buffer for display to the user's screen or another offscreen buffer that could for example be used as a texture source for subsequent stages of the rendering/shading process as is done for global illumination.

Shaders written in GLSL are compiled as an executable program that is loaded and run by the GPU to provide varying effects. Multiple shaders can be stacked together in multi-pass rendering techniques to produce ever more complex effects such as that used for the Depth of Field (DOF) shaders available in the ETTS which can mimic the effect of a wide aperture by blurring objects outside of the focal point. Since there can be many different shaders applied and each object in a scene may require a different combination and configuration of shaders, efficient management of the available shader programs is a must. While some shaders such as the general purpose fill vertex and fragment shader is required by multiple objects repeatedly and others such as the one used for drawing a textured quad is only used once, it makes sense to avoid the need to compile multiple instances of the same shader programs.

The approach used in the ETTS system works similar to the flyweight programming pattern where a commonly repeated object can be shared as a singly allocated resource rather than instantiating multiple identical copies that occupy additional memory. The implementation works by creating a Shader Bank that serves as a central structure for referencing all available shader programs that have been compiled and loaded. The main advantage with this approach is that components using a common shader need only hold a reference to the compiled program rather than each requiring a separate instantiation and allocation of the program.

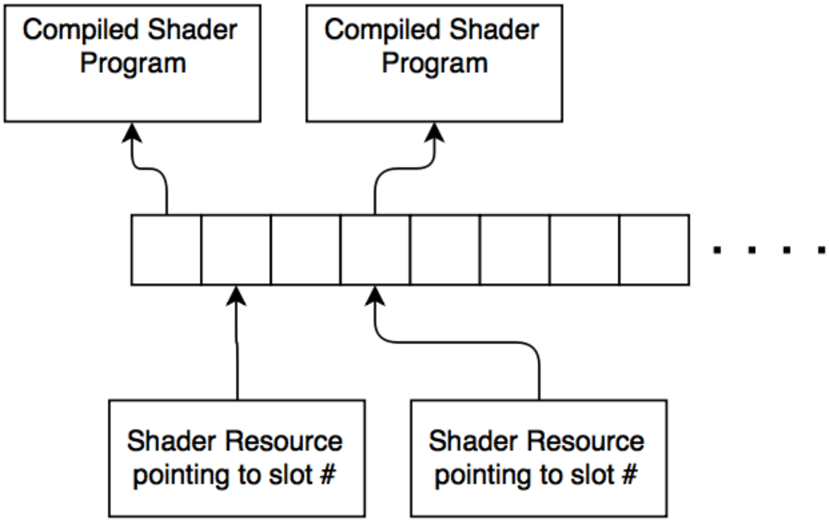

Compiled Shader Program
Compiled Shader Program
. . . .
Shader Resource pointing to slot #
Shader Resource pointing to slot #

Figure 4.8: Figure depicting structure of shader bank. Objects along with the render enging hold shader resources which specify the slot number corresponding to the compiled program.

The bank structure as represented in the figure above, also affords an interesting property whereby shaders can be modified and compiled during runtime of the ETTS and changes are updated immediately. This is because components reference only the slot in the bank where a shader is stored, rather than the compiled shader itself. The result is that the shader occupying the slot can be modified or swapped and the changes are automatically propagated through without the need to restart the ETTS. The performance benefits of such an approach are clear from a resource perspective as each shader is compiled and stored only once, regardless of the number of components referencing it.

4.6.3 Material Structure and Management

The properties used to define an object's appearance and how it reacts to light sources within the scene, are organized into the Material class. Much like real-world materials, the Material class defines an object's ambient, diffuse, and specular lighting properties. Figure 4.9 shows the structure of the Material class.

In addition to setting being able to set a material's ambient and diffuse colours, texture files can be used to provide detailed sampling of diffuse colour maps, normal maps, specular maps; important properties used by the shaders to provide realistic renderings of objects and lighting in the scene.

Organization of materials follows a very similar implementation to that of the shader bank system described in section 4.6.2 employing a bank style system that stores materials in slots

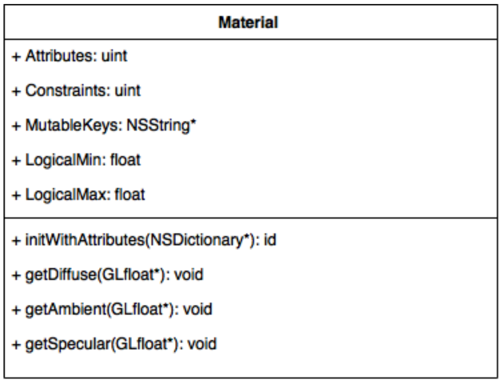

Material
+ Attributes: uint
+ Constraints: uint
+ MutableKeys: NSString*
+ LogicalMin: float
+ LogicalMax: float
+ initWithAttributes(NSDictionary*): id
+ getDiffuse(GLfloat*): void
+ getAmbient(GLfloat*): void
+ getSpecular(GLfloat*): void

Figure 4.9: Class diagram of the Materials object

for dynamic updating and shared use of materials by multiple objects. The advantages of this design are the same, offering improved space and runtime performance; saving space by allowing multiple objects to make use of a single material resource. Another advantage with this flyweight/shared-resource approach is that pre-configured materials can be designed and dropped into use by any object without having to reconfigure them each time a new scene is created.

4.7 Collision Detection

Collision detection is used by the ETTS for the detection of tool placement during targeting-based tasks so that user performance can be logged and evaluated. Collision detection also serves the purpose of user feedback by preventing the tool and other objects from passing through one another as well allowing for context based control of tools; optionally switching the method of control from one of translation to that of rotation upon surface contact with a designated collider (i.e skull of manikin).

The implementation of the collision detection system works on the principle of triangular intersection with a ray and is modelled after the Möller-Trumbore intersection algorithm. The algorithm works by casting a ray out into space from a given origin point and following a set direction. A triangular shaped area defined by three vertices, with a face normal calculated using the crossproduct of two vectors formed by the triangle is then used to check for an intersection with the ray. Upon finding an intersection, the distance along the ray's path that marks the point of intersection is then returned and can be evaluated to reveal a point in three dimensions.

Object meshes are configured to use triangular polygons in the ETTS framework which lends itself well to the application of this particular algorithm that offers fast computation of the intersection between a ray and triangle in three dimensions. Collision detection is further simplified due to the fact that we can easily model the endoscopic tool (or similarly shaped object in non-clinical applications) as a ray that has an origin and direction. The triangles being tested for intersection are of course the faces of the triangulated meshes rendered in the scene.

The use of the ray-triangle intersection data can be used to map the projection of the source of the ray (endoscopic tool) onto the surface of the target(s) (skull and target within). When designing a simulator scenario, objects within the scene can be marked as having different roles in the collision detection process, such as a tool which acts as the primary ray source, an outer target (i.e skull mesh) and inner target (i.e ventricle mesh). In this mode of operation the simulator calculates the intersection point as projected onto the outer target from two rays which emanate out from the tool object and inner target object. The direction used for the inner target object is that of it's longest axis which should serve as a the ideal targeting path. With the intersection of the inner target's longest axis with the outer target computed (pre-processed), this point on the outer target is then used to mark the ideal targeting point that a candidate could achieve. Together with the tool object's tip position, the distance and rotational error can be recorded for use in evaluating user performance. Figure 4.10 provides a visualization of this process.

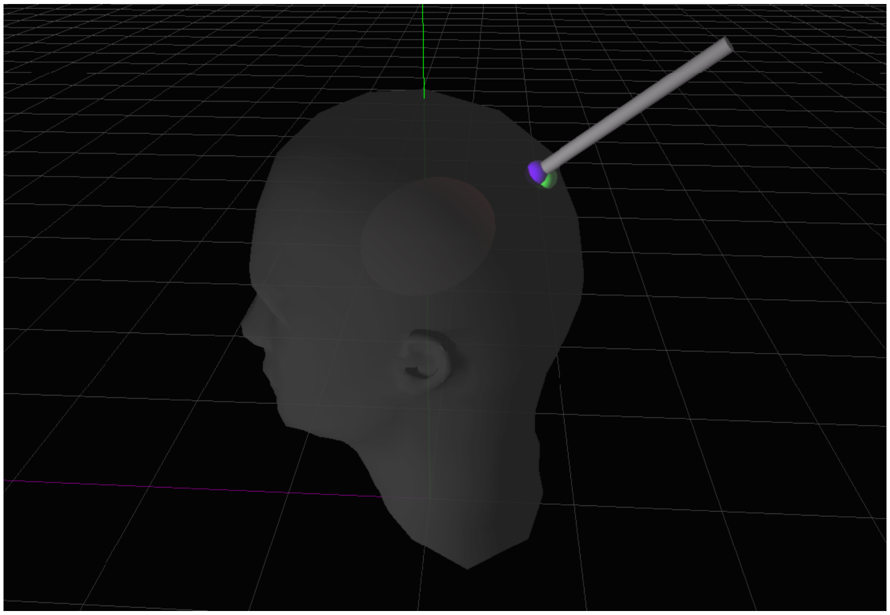


Figure 4.10: Figure depicting results of collision detection system. Purple marker represents the tool's projected contact point on the skull surface, while the green marker represents the ideal position to achieve zero error.

The other way in which the collision detection system is utilized is to provide context based control of the targeting tool. The option exists to augment the candidate's available control of the targeting tool depending on the context of the situation; by default tool manipulation can be rotated and translated freely. Enabling context driven tool control restricts movement of the tool to translation only until contact is made with the outer target (i.e skull mesh) at which point the applied input is used for rotational control over the tool; thus separating the acts of translation and rotation into two separate and sequential tasks.

4.8 Trial Data

4.8.1 Structure and Overview

Data collected during a user's trial session is stored in an array of dictionaries not unlike the structure used to describe the scene of a scenario. Each entry in the array is a dictionary which is a key-value pair structure that is used to hold the current state information of objects tracked in the scene. The information stored is a simplified version of what is stored in the scene data structure in Listing A.1 in Appendix A, this is done to minimize the size of the session data and optimize the efficiency of the simulator. When in a recording state, scene data is collected during each render pass (approximately 60Hz or 60fps) and appended to the trial data data structure, using the delta time since the start of the recording as an index. Upon completing a trial, the trial data data structure is serialized into a JavaScript Object Notation (JSON) and Comma Separated Values (CSV) formatted files that are then written to disk.

While the two files contain the same data describing a user's session, JSON is generally regarded as a more human readable format and can be used by the ETTS to import past user sessions for playback and analysis within the simulator package. The main drawback with JSON encoding over CSV is that it tends to be a bit more verbose and thus requires more space to store the same information as a CSV file. Listing 4.5 shows a compressed excerpt of a JSON encoded trial data session. The data within the JSON encoded file is organized as sets of all the tracked objects in the scene, recording the individual object states for a given time index. Indexing the data by time as opposed to by frame number allows for smoother playback of events since linear interpolation can be used if desired to determine an object's in-between value (applicable to a moving object's position for instance) where the rate of playback does not match that of the capture rate.

The reason for using CSV encoding alongside JSON is simply for the convenience of being able to import the data into a spreadsheet or graphing package external to the ETTS. The listing below illustrates an example of the data structure for a sample trial data session when encoded using CSV. As can be seen in this simplified version of a trial data file, the first line acts as a header, defining the names of the various attributes whose values or recorded. The values themselves are delimited using commas, forming rows of data for each entry in the log file. Within each entry is a value under the heading of Time which represents the index for each entry as the delta time from when the recording started.

4.8.2 Dataset Generation

The primary focus of the ETTS system is the application in evaluating user performance as it relates to targeting-based tasks, specifically clinically relevant scenarios formed around endoscopic procedures. Creation of a single scenario within the ETTS environment is a trivial task, requiring the user to simply import the mesh objects that make up the scene and position them as needed. Material properties, lighting conditions, and initial tool and camera placement are all easily configured using the provided scenario designer. In order for user performance to be evaluated and any trends studied, a large number of scenarios need to be generated, all having slight variations in their difficulty and approach. Creating large sets of scenarios can be quite tedious when completed one at a time.

Listing 4.5: Compressed sample of trial session data

```
{
  "P-5549872" : {...},
  "Time" : 0.03721201419830322,
  "O-6118256" : {
    "targeting-position" : {
      "X" : 0,
      "Y" : 0,
      "Z" : 0
    },
    "position" : {
      "X" : 12.66267,
      "Y" : 47.21453,
      "Z" : -5.62711
    },
    "rotation" : {
      "X" : 0,
      "Y" : 0,
      "Z" : 0
    }
  },
  "P-5736752" : {...},
  "P-5631776" : {...},
  "O-285247120" : {...},
  "O-7162576" : {...},
  "C-0" : {...},
  "O-7255632" : {...},
  "O-7182976" : {...}
},
```

The Trial Dataset Generator (TDSG) module of the ETTS system is used to create a collection of randomized scenarios formed around a single template scene. This collection of generated scenarios is referred to as a Trial Dataset (TDS) and represents the collection of scenarios that candidate's will complete and have their performance evaluated against. The initial template scenario is used by the TDSG as an initial condition for all of the scene objects, the attributes of which are then varied to produce a set of similar yet distinct scenarios.

Figure 4.11 illustrates the various options available to the user in generating the desired set of randomized scenarios. Selecting an object from the scene objects list of the ETTS' main interface updates the currently selected object within the TDSG window and allows the user to set initial designate objects as either a *Tool*, *Target*, or neither. The selection of *Tool* or *Target* status is mutually exclusive and a given scenario can and must have one of each assigned before a dataset can be generated.

Listing 4.6: CSV example of trial session data

```
O-T/position/X, O-T/position/Y, O-T/position/Z, Time,...
-0.491689,    51.717903,    -10.440434,   0.047799,...
-0.491689,    51.717903,    -10.440434,   0.084509,...
-0.491689,    51.717903,    -10.440434,   0.101944,...
-0.491689,    51.717903,    -10.440434,   0.118137,...
-0.491689,    51.717903,    -10.440434,   0.135029,...
-0.491689,    51.717903,    -10.440434,   0.151682,...
-0.491689,    51.717903,    -10.440434,   0.168353,...
-0.491689,    51.717903,    -10.440434,   0.184880,...
```

Configurable options for objects (as opposed to the camera, lights, and orthographic slice planes) include position, rotation, scale, material properties, and in the case of a *Target* type object, eccentricity. In all of the configurable attributes, each component (i.e X, Y, Z for position, scale, and rotation) can be configured to be either static or variable. Component attributes designated as being variable allow the user to set constraints for the TDSG, serving as minimum and maximum bounds.

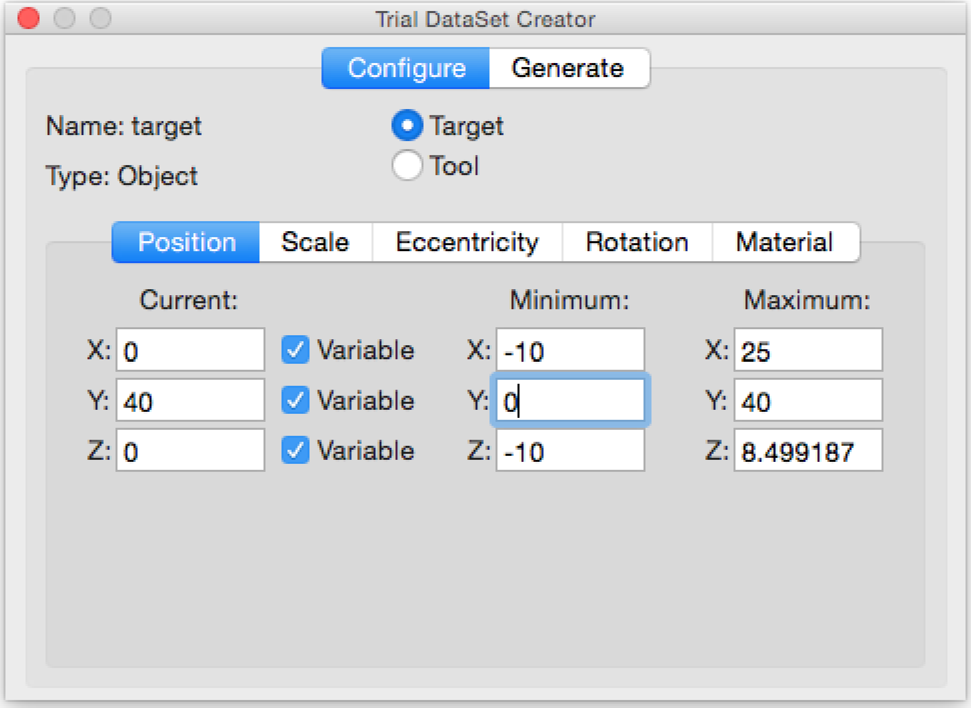
Selecting a scene object to serve as the scenario target automatically adds the eccentricity configuration tab to the TDSG options and is illustrated in Figure 4.12. As can be seen in the figure, the object's eccentricity can be set either using the provided slider or numerical input field. Assuming that the target object has a variable eccentricity, the user can also provide the minimum and maximum values, constrained to the range [0,1]. Target type objects also restrict the scaling of the object, locking the object's scale attributes together ensuring no deformation takes place prior to adding the desired level of eccentricity. Section 4.9.2 discusses the process of generating ellipsoid targets using the provided configuration options from the TDSG.

Generation of a randomized dataset is performed once all configurations have been made in the TDSG window and the user has selected the number of scenarios to generate and the type of user interface device (Gamepad or Leap Motion). Listing 4.7 illustrates how the template scenario is used to derive new scenarios of varying configurations.

The creation of newly derived scenarios works by leveraging the key-value storage of scene and object attributes, first replicating the template scene and then iterating through object and scene attributes. The function checks which attributes are marked as variable (set in the TDSG dialog) and using the provided constraints, proceeds to assign randomized values within the provided ranges. The modified duplicate of the template scenario is then added to an array of scenarios that represents the trial dataset to be serialized and saved. See Section 4.9.2 for further information on the generation of ellipsoids of varying eccentricities.

4.8.3 Capturing Trial Session Data

A crucial aspect of the ETTS system is the ability to capture the session data produced while running user performance trials, both for playback and analysis. The general idea behind the data acquisition process is to generate *snapshots* of the scene's state with sufficient periodicity



Name: target

Type: Object

● Target
○ Tool

Current:		Minimum:	Maximum:
X: 0	☑ Variable	X: -10	X: 25
Y: 40	☑ Variable	Y: 0	Y: 40
Z: 0	☑ Variable	Z: -10	Z: 8.499187

Figure 4.11: Trial Dataset Generator window - example of available configuration options.

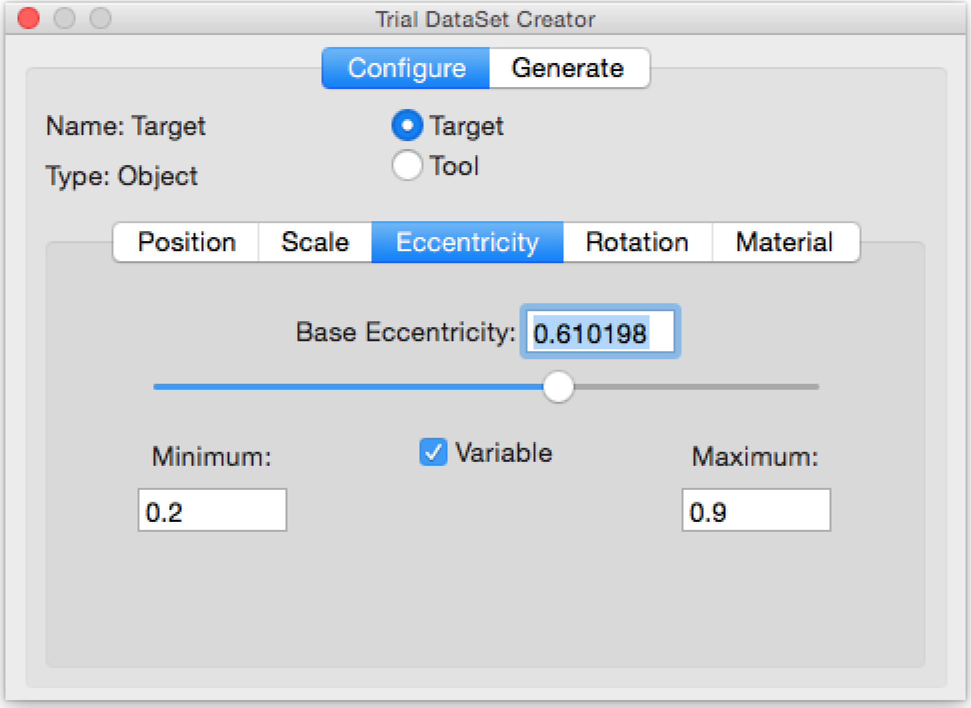
to produce an accurate account of a user's actions so that their performance may be evaluated and scored. At the same time steps must be taken to ensure that the acquisition of data doesn't hamper the simulator's performance and the data produced optimizes the amount of storage space required to convey as much information as needed.

Upon starting a new trial, the user is prompted to set the output directory for the session capture files and enter their assigned user ID (for anonymous tracking). The session capture process begins by signalling the render engine to switch to a recording state. This is done with the *beginTrialCapture* function shown in Listing 4.8.

The *beginTrialCapture* function starts by allocating a dynamic array to store the individual scene snapshots throughout the trial. At the same time, the start time is recorded allowing subsequent snapshots to be stored with a time index from the start of the trial being recorded. Lastly the render engine is set to a recording state that augments the render flow to capture subsequent frames of data.

The method used within the ETTS system to capture and store session data is similar in fashion to that used for serializing and saving scenes as discussed in Section 4.5.2 and a sample excerpt of the resulting data is shown in Listing 4.5. Listing 4.9 details the *sceneCapture* function that is called from the main render loop while the simulator is in a recording state.

The scene capture function works by first allocating a dictionary (key-value pair) object to store the state information of the scene. Iterating through the scene objects list, the function adds a stripped down hierarchical representation of each object in the scene, including only position and rotation related attribute states (flexibility exists to track an arbitrary number of



Name: Target

Type: Object

● Target

○ Tool

Base Eccentricity: 0.610198

Minimum: 0.2

☑ Variable

Maximum: 0.9

Figure 4.12: Trial Dataset Generator window - example of available eccentricity configuration options.

attributes per object). Following the general frame structure, as illustrated in Listing 4.5, the current state of the scene camera's attributes are captured along with the frame's timestamp (delta time from start of recording action). The last step of the scene capture function is to append the contents of the current frame to the trial capture storage array before returning to the main render loop.

4.8.4 Session Playback and Analysis

The primary focus of the ETTS platform is for the evaluation of user performance in targeting tasks and specifically those that are relevant within the domain of endoscopic surgical procedures. An important function of the ETTS system is its ability to be used as a teaching aid, providing immediate visual feedback to the user user in an interactive environment that allows for the playback of saved trial sessions.

Figure 4.13 illustrates the interactive playback and analysis module of the ETTS system that runs in parallel to the regular simulator interface as shown in Figure 4.14. The module allows a user to open a trial replay file (saved with a proprietary *.rply* extension) that is generated whenever a trial scenario is completed. Upon importing the trial replay file, the scenario scene is automatically reconstructed in the main ETTS view and the distance and rotational error data is computed for the trial as a function of time, allowing it to be plotted as shown in Figure 4.13. The interface allows the user to start, stop, and pause continuous playback of the recorded

Listing 4.7: Excerpt from trial dataset generation process

```
NSArray *subKeys = [[attributes objectForKey:attributeKey] allKeys];
for (int l = 0; l < [subKeys count]; l++)
{
  if ([[[constraints objectForKey:attributeKey] valueForKey:[NSString
      stringWithFormat:@"~%@", [subKeys objectAtIndex:l]]] boolValue])
  {   //Whether attribute is variable or fixed

     float max = [[[constraints objectForKey:attributeKey] valueForKey:[
         NSString stringWithFormat:@"+%@", [subKeys objectAtIndex:l]]]
         floatValue];
     float min = [[[constraints objectForKey:attributeKey] valueForKey:[
         NSString stringWithFormat:@"-%@", [subKeys objectAtIndex:l]]]
         floatValue];
     float randomValue = (((float)(arc4random() % ((unsigned)RAND_MAX +
         1)) / RAND_MAX) * (max-min)) + min;

     [[attributes objectForKey:attributeKey] setObject:[NSNumber
         numberWithFloat:randomValue] forKey:[subKeys objectAtIndex:l]];
  }
}
```

Listing 4.8: Function used to start capturing trial data

```
-(void) beginTrialCapture: (NSMutableArray*) sceneData
{
     sceneData = [[NSMutableArray alloc] init];
     currentTrialCaptureStorage = sceneData;
     currentTrialStartTime = [NSDate date];
     [self setIsRecordingTrialData:true];
}
```

session, while the playback and analysis module displays an advancing account of the user's targeting performance, synchronized to the playback of the ETTS view. The scene remains fully interactive throughout the entire playback and analysis stage, allowing the user to seek to any point in the playback while at the same time still being able to pause and take full control of the scenario. When playback is paused, objects, the camera, lights, and material properties can all be moved and manipulated. Upon resuming playback or seeking to another time index, all changes are lost and the scene shown in the main ETTS view is reconfigured to match the exact state as recorded at the corresponding time index.

There are numerous advantages that arise through the support of trial session playback and analysis, transforming the ETTS system from a standalone simulator to a suite of tools that can be used to help improve the skills required by a user to complete complex targeting tasks. The debriefing nature of the playback and analysis module allows the user to observe a visual record of their actions and presents the captured session in a manner that makes it easy for users to draw a direct correlation between their actions and the resulting targeting accuracy.

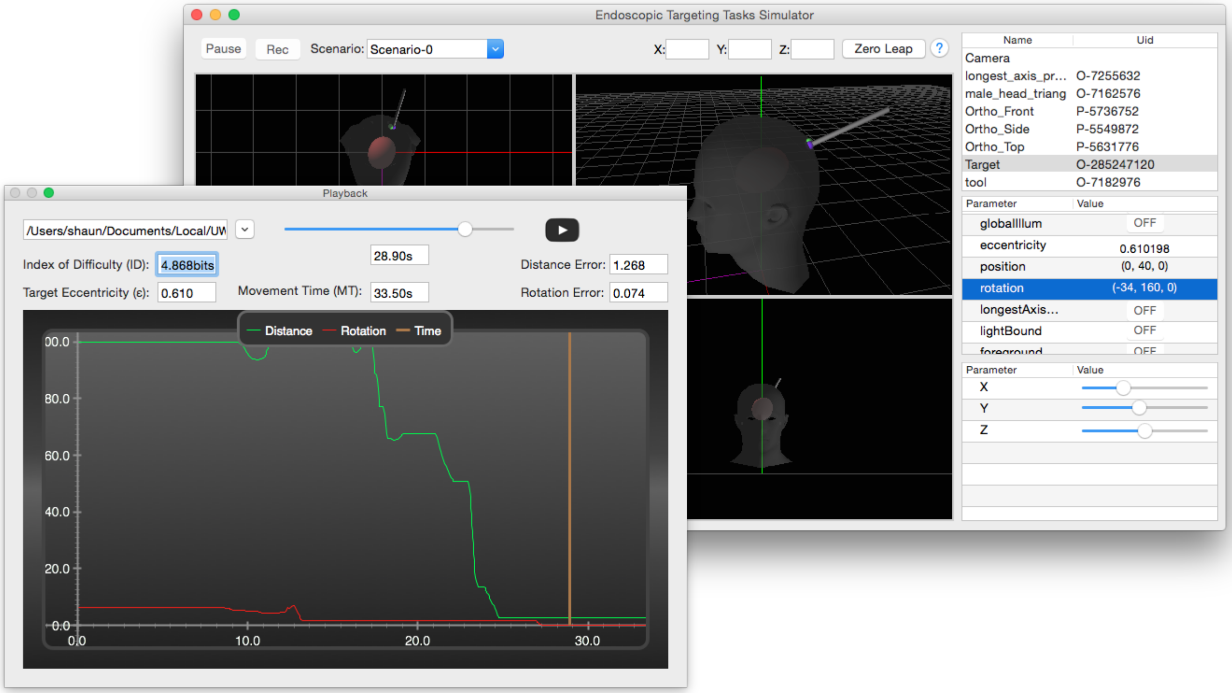
Listing 4.9: Function used to capture scene state

```
−(void) sceneCapture{
    NSMutableDictionary *entry = [[NSMutableDictionary alloc] init];
    [entry setObject:[NSNumber numberWithDouble:−[currentTrialStartTime
        timeIntervalSinceNow]] forKey:@”Time”];
    for (NSUInteger i = 0; i < [sceneObjects count]; i++){
        if ([[sceneObjects objectAtIndex:i] isKindOfClass:[Entity class
            ]]){
            Entity *object = [sceneObjects objectAtIndex:i];
            NSString *key = [[object attributes] objectForKey:@”uid”];
            [entry setObject:[[NSMutableDictionary alloc] init] forKey:key
                ];
            [[entry objectForKey:key] setObject:[[[object attributes]
                objectForKey:@”position”] copy] forKey:@”position”];
            [[entry objectForKey:key] setObject:[[[object attributes]
                objectForKey:@”targeting−position”] copy] forKey:@”
                targeting−position”];
            [[entry objectForKey:key] setObject:[[[object attributes]
                objectForKey:@”rotation”] copy] forKey:@”rotation”];}}
    [entry setValue:[mainCamera attributes] forKey:@”C−0”];
    [currentTrialCaptureStorage addObject:entry];}
```

Implementation of the playback and analysis module is split between the process of importing and preparing the replay data and the operation of providing continuous playback and seek data. The *.rply* files themselves follow a similar JSON encoded structure that is discussed in Sections 4.8.1 and 4.5.3. The format of the files is essentially a hybrid between the exported scene file and trial session file, taking the trial session file and appending a footer containing the scene's data, organized into a list of constraints and a list object scene data (as discussed in Section 4.5.2.

Once the scene data and constraints have been parsed out of the trial session replay file, an array is dynamically allocated to store the time indices that will be used to playback the individual frames of data. The parameters used in the analysis portion of th ETTS system are retrieved from the imported file, starting with the objects that represent the tool and target within the scene. Listing 4.10 illustrates how the two key objects are identified and their UID's saved.

With both the tool and target objects identified, the next step is to traverse all of the saved frames of data to extract the object attributes used in the analysis of targeting error as a function of time. Listing 4.11 illustrates the process of extracting the necessary attributes from each time index and then using the information to produce a normalized error for the rotation and position components as a function of time. While the tool's tip position (denoted by the attribute key *position*) is sufficient for calculating the position error, there would be a constant offset error for each scene if we were to use the same attributed key for the target's position. This is due to the fact that the target's position represents its coordinates that lie within the outer-target/skull. What we are instead interested in is the ideal entry position that represents the target's longest axis projection onto the outer-target/skull surface. It is this intersection point of the target's longest axis with the skull that represents the ideal entry point that a user can achieve.



Scenario: Scenario-0

Name	Uid
Camera	
longest_axis_pr...	O-7255632
male_head_triang	O-7162576
Ortho_Front	P-5736752
Ortho_Side	P-5549872
Ortho_Top	P-5631776
Target	O-285247120
tool	O-7182976

Parameter	Value
globalIllum	OFF
eccentricity	0.610198
position	(0, 40, 0)
rotation	(-34, 160, 0)
longestAxis...	OFF
lightBound	OFF
foreground	OFF

/Users/shaun/Documents/Local/UW

28.90s

Index of Difficulty (ID): 4.868bits

Target Eccentricity (ε): 0.610

Movement Time (MT): 33.50s

Distance Error: 1.268

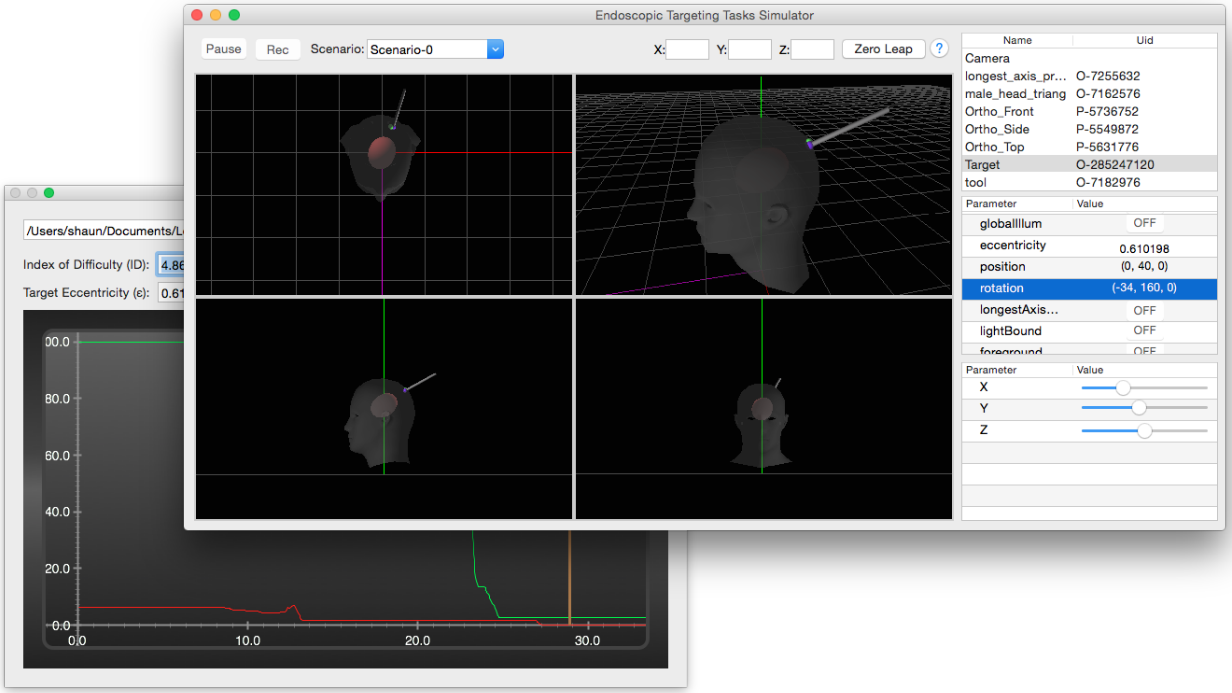
Rotation Error: 0.074

Figure 4.13: Playback and Analysis of ellipsoid targeting task.

Listing 4.10: Excerpt of tool and target isolation in replay file

```
playBackTimeIndices = [[NSMutableArray alloc] init];
NSString *targetObjectKey;
NSString *toolObjectKey;
for (int j=0; j < [sceneObjects count]; j++)
{
  if ([[sceneObjects objectAtIndex:j] isKindOfClass:[Entity class]])
  {
    if ([[[(Entity*)[sceneObjects objectAtIndex:j] attributes] objectForKey:@”target”] boolValue]){
      targetObjectKey = [[(Entity*)[sceneObjects objectAtIndex:j] attributes] objectForKey:@”uid”];
    } else if ([[[(Entity*)[sceneObjects objectAtIndex:j] attributes] objectForKey:@”inputBound”] boolValue]){
      toolObjectKey = [[(Entity*)[sceneObjects objectAtIndex:j] attributes] objectForKey:@”uid”];
    }
  }
}
```

When the ETTS is in a recording state, object’s identified as being a *target* make use of the collision detection system discussed in Section 4.7 to store an additional attribute, *targeting-position*. The value stored under this key represents the coordinates of the intersection between



Endoscopic Targeting Tasks Simulator

Scenario: Scenario-0

/Users/shaun/Documents/L

Index of Difficulty (ID): 4.86

Target Eccentricity (ε): 0.61

Name	Uid
Camera	
longest_axis_pr...	O-7255632
male_head_triang	O-7162576
Ortho_Front	P-5736752
Ortho_Side	P-5549872
Ortho_Top	P-5631776
Target	O-285247120
tool	O-7182976

Parameter	Value
globalIllum	OFF
eccentricity	0.610198
position	(0, 40, 0)
rotation	(-34, 160, 0)
longestAxis...	OFF
lightBound	OFF
foreground	OFF

Figure 4.14: Playback view of ellipsoid targeting task.

the *target's* longest axis and the outer-target/skull, a point that lies on the surface of the skull mesh. The calculation of positional error therefore makes use of the tool's tip position and the target's projected *target-position*.

The rotational component of both the tool and target objects are retrieved in similar fashion to the position components. Since we are focused on the longest axis of the ellipsoid target, a degree of ambiguity exists for there are two possible intersect points of an ellipsoid's longest axis and the surrounding outer-target/skull. As an example, in the case where the target ellipsoid is oriented with its longest axis running sagittally (front to back), there is an intersection and thus possible solution located near the anterior of the skull with another located in the posterior section. For the purposes of removing any ambiguity of this nature, only the intersection point that lies within the top hemisphere of the skull surface is considered.

Using the knowledge that ellipsoid target eccentricity is applied as a scaling of the object's *Z-Axis*, the correct longest axis intersection direction is defined as that which emanates out from the centre of the object and along the positive *Z-Axis*. Listing 4.11 illustrates how the *targetRotation* components are corrected by rotating 180° about the *Y-Axis* and applying a reflection about the *X-Axis*.

The other ambiguity that exists with respect to the orientation of the tool object is its roll component which is the rotational angle around the *Z-Axis* in the object's local coordinate system (this assumes that the tool mesh has been modelled with its forward facing direction aligned along the positive *Z-Axis*). For our purposes we are not at this time concerned with the roll angle of the tool object as the targeting tasks being simulated are only intended to measure

Listing 4.11: Excerpt of replay analysis preprocessing

```
for (int i=0; i < [frames count]; i++)
{
  NSDictionary *frame = [frames objectAtIndex:i];
  if (![toolObjectKey isEqualToString:@””] && ![targetObjectKey isEqualToString:@””])
  {
    NSDictionary *tool = [[frame objectForKey:toolObjectKey] objectForKey:@”position”];
    NSDictionary *target = [[frame objectForKey:targetObjectKey] objectForKey:@”targeting-position”];
    NSDictionary *toolR = [[frame objectForKey:toolObjectKey] objectForKey:@”rotation”];
    NSDictionary *targetR = [[frame objectForKey:targetObjectKey] objectForKey:@”rotation”];

    GLKVector3 toolPosition = GLKVector3Make([[tool objectForKey:@”X”] floatValue], [[tool objectForKey:@”Y”] floatValue], [[tool objectForKey:@”Z”] floatValue]);

    GLKVector3 targetPosition = GLKVector3Make([[target objectForKey:@”X”] floatValue], [[target objectForKey:@”Y”] floatValue], [[target objectForKey:@”Z”] floatValue]);

    GLKVector3 toolRotation = GLKVector3Make([[toolR objectForKey:@”X”] floatValue], [[toolR objectForKey:@”Y”] floatValue], 0);

    GLKVector3 targetRotation = GLKVector3Make(-[[targetR objectForKey:@”X”] floatValue], [[targetR objectForKey:@”Y”] floatValue]-180.0f, 0);
```

targeting accuracy within 5 DOF. The roll angle of the tool object is therefore removed by setting it as a constant having a rotational angle of 0°.

The rotational difference or error is computed by performing the dot product of the *tool* rotation vector and the rotation vector that defines the direction of the *target's* longest axis. Computing the dot product of the normalized rotation vectors returns a scalar value in the range [-1, 1], with -1 corresponding to a rotational error of 100% or 180° out of phase with the longest axis vector. A scalar value of 1 conversely translates to no error, signifying that the two direction vectors are in fact co-directional. Listing 4.12 illustrates how the direction/rotation vectors are normalized before the dot product is computed. The resulting scalar value is then inverted and scaled to fit the rotational error values to the range [0,100] for plotting.

The process of calculating the position/distance error between the *tool's* position and that of the *target's* longest axis projection on the outer-target/skull is illustrated in Listing 4.12. The distance between the two points is calculated as a singular value representing the euclidean distance (see equation (4.1)) between them. Making the assumption that in a targeting-based task, the furthest distance from the target should be the initial starting position, decreasing as the tool is moved closer to the target, we take the maximum distance to be the tool's position at

Listing 4.12: Continuation of Listing 4.11 calculating normalized distance and rotation error

```
  float rotationDistance = 50 * (1 - GLKVector3DotProduct(
      GLKVector3Normalize(toolRotation), GLKVector3Normalize(
      targetRotation)));

  [plotDataR addObject:[NSDictionary dictionaryWithObjectsAndKeys:[
      NSDecimalNumber numberWithDouble:[[frame valueForKey:@"Time"]
      doubleValue]], [NSNumber numberWithInt:CPTScatterPlotFieldX], [
      NSDecimalNumber numberWithFloat:rotationDistance] , [NSNumber
      numberWithInt:CPTScatterPlotFieldY],nil]];

  float distance = GLKVector3Distance(toolPosition, targetPosition);

  if (i == 0)
  {
    initialDistance = distance;
  }

  float normalizedDistance = 100 * (distance / initialDistance);

  maxDist = 100;

  [plotDataTX addObject:[NSDictionary dictionaryWithObjectsAndKeys:[
      NSDecimalNumber numberWithDouble:[[frame valueForKey:@"Time"]
      doubleValue]], [NSNumber numberWithInt:CPTScatterPlotFieldX], [
      NSDecimalNumber numberWithFloat:normalizedDistance] , [NSNumber
      numberWithInt:CPTScatterPlotFieldY],nil]];
}
[playBackTimeIndices insertObject:[NSNumber numberWithDouble:[[frame
    valueForKey:@"Time"] doubleValue]] atIndex:i];
```

time index of 0s. This allows the distance error to be normalized to a range that can be plotted of [0,100], where 0 implies zero distance error and conversely 100 indicates an distance of 100% being that the tool is no closer to the target than from where it began.

The normalized values for the position and rotation errors are appended to the plot shown in Figure 4.13 along with the time index that corresponds to the current values. The length of the playback is recorded and used to set the length and scale of the slider used for seeking through the playback data and indicating the current playback position.

In addition to calculating the position and rotation errors for the session, parameters relating to the evaluation of user performance using Fitts' methodology (discussed in Section 2.5) are computed and posted to the user interface. The relevant parameters being the *ID* and *MT* for the trial. Listing 4.13 illustrates how the above mentioned parameters are determined.

Playback of a recorded trial session is accomplished through the render engine which augments its behaviour depending on whether it is in a playback state as set by the Playback and Analysis interface shown in Figure 4.13. Listing 4.14 shows the change in behaviour that occurs in the render engine when operating in a playback state.

When in a playback state, the render engine is responsible for advancing the playback

Listing 4.13: Calculation of Fitts' Law Parameters

```
//Calculate Index of Difficulty
float D = initialDistance;
float W = 1.0f / epsilon;
float ID = log2f(D/W + 1);
[self setIndexOfDifficulty:[NSString stringWithFormat:@"%.3fbits", ID]];
[self setMovementTime:[NSString stringWithFormat:@"%.2fs", [self
    playBackFileDuration]]];
```

Listing 4.14: Playback state behaviour of render engine

```
if ([delegate_handle playBackState])
{   //if Playing then advance frame
  if ([delegate_handle playBackCurrentFrameIndex] < ([[delegate_handle
      playBackSceneFrames] count] − 1))
  {
    [delegate_handle setPlayBackCurrentFrameIndex:([delegate_handle
        playBackCurrentFrameIndex] + 1)];
    [delegate_handle playbackSliderAction:self];
    [delegate_handle setOGLViewIsDirty:true];
  }
  else
  {
    [delegate_handle setPlayBackState:0];
    [delegate_handle setPlayBackCurrentFrameIndex:0];
    [delegate_handle playbackSliderAction:self];
    [delegate_handle setOGLViewIsDirty:true];
  }
}
```

frame data at the start of a new rendering cycle. Listing 4.14 illustrates the function calls that are made to the application delegate, controlling which frame data to load next and keeping the interface shown in Figure 4.13 synchronized to the frame being rendered. Once the frame index is advanced, the delegate is notified that it needs to update the slider position. If playback reaches the end of the recording, the index is reset back to the beginning.

The *playbackSliderAction* function is called by the render engine at the start of rendering a new frame while in a playback state. The function works to update the current scene state with the captured parameters of the current playback frame, this is easily accomplished by leveraging the same key-value pair dictionary storage of object and scene attributes. Upon updating the attributes of the objects within the scene to reflect the recorded state of the current frame, the position of the error plot's current time index is set to match the current time index being rendered. Marking the scene as *dirty* causes the render engine to render the newly updated scene. Listing A.2 in Appendix A details the process of using the captured frame data to continuously update the scene as playback advances. Binding the playback seek slider to the same *playbackSliderAction* function allows for intuitive interactive control of the current playback position by moving the slider to the desired position.

4.9 Ellipsoid Targets

4.9.1 Ellipsoids and Eccentricity

The focus of the ETTS system is the evaluation of user performance in scenarios requiring the user to perform a targeting task with clinically relevant goals and requirements. The choice of geometry used in modelling a suitable target that allows for the robust measurement of user performance metrics, while still presenting similar challenges found in a clinical setting, is an ellipsoid having variable eccentricity. The longest axis of the ellipsoid target represents in a clinical setting, the ideal entry point for endoscopic surgical tasks, minimizing the cross-sectional area. In terms of measuring targeting accuracy, the use of an ellipsoid's longest axis presents a multi-faceted challenge to users, testing a variety of skills including spatial reasoning in identifying the longest axis along with the required dexterity to manipulate a tool's orientation and position to match.

The longest axis of an ellipsoid target requires the accurate identification and movement in 5 DOF: Position in *X*, *Y*, and *Z* and Rotation in terms of *Pitch* and *Yaw*. An important asset with choosing an ellipsoid as the target shape is that its eccentricity has a direct correlation to the degree of difficulty in identifying the longest axis. The Eccentricity of an ellipsoid is measured as being the ratio of the foci distances from the centre and the length of the semi-major axis (major axis being the longest axis). Essentially a higher epsilon value (eccentricity) tending towards unity will produce a more pronounced scaling/stretching along the major axis, while an epsilon value tending towards a value of zero approaches a perfect sphere having uniform radius in three dimensions. As a result of this unique property, varying the level of eccentricity controls the single DOF for the ill-posedness of the task in which a user must target the longest axis of the target. Furthermore, this property of the ellipsoid allows for the generation of targets of varying difficulty as defined in Fitts' Law as the ID.

In addition to varying the eccentricity of the ellipsoid target, distinct challenges can be introduced by varying the scale/size, position, and rotation of the target to increase the level of difficulty and complexity of the scenario. This ability to vary the degree of difficulty that different geometries present is not unlike the inconsistency experienced in a clinical setting when dealing with real anatomical structures.

4.9.2 Ellipsoid Target Generation

The generation of ellipsoid shaped targets is performed by varying the scale, position, and eccentricity of an initial mesh that represents a sphere with a scale of unity. Configuration of the attributes used in generating the randomized dataset of targets is done through the interface shown in Figures 4.11 and 4.12. Target eccentricity can be constrained within a set of bounds by defining the desired maximum and minimum range of values. Listing 4.15 illustrates how the transformations are applied to the spherical template mesh.

The calculation of scaling values in *X*, *Y*, and *Z* based on the desired level of eccentricity is calculated using equation (4.2).

Listing 4.15: Transformation of eccentricity applied to spherical mesh to generate ellipsoid target

```
float max = [[[constraints objectForKey:attributeKey] valueForKey:[NSString stringWithFormat:@”+%@”, [subKeys objectAtIndex:0]]] floatValue];
float min = [[[constraints objectForKey:attributeKey] valueForKey:[NSString stringWithFormat:@”-%@”, [subKeys objectAtIndex:0]]] floatValue];
float randomValue = (((float)(arc4random() % ((unsigned)RAND_MAX + 1)) / RAND_MAX) * (max-min)) + min;
float e = [[attributes objectForKey:@”eccentricity”] floatValue];
float c = e * randomValue;
float b = sqrtf(randomValue * randomValue - c * c);

for (int l = 0; l < [subKeys count]; l++){
  if ([[subKeys objectAtIndex:l] isEqualToString:@”Z”]){
    [[attributes objectForKey:attributeKey] setObject:[NSNumber numberWithFloat:randomValue] forKey:[subKeys objectAtIndex:l]];
  }else{
    [[attributes objectForKey:attributeKey] setObject:[NSNumber numberWithFloat:b] forKey:[subKeys objectAtIndex:l]];
  }
}
```

$$\begin{aligned} e &= eccentricity \\ c &= e * randomValue \\ b &= \sqrt{(randomValue)^2 - (c)^2} \end{aligned} \tag{4.2}$$

Objects bearing the *target* flag apply the randomized scaling attributes as per the process shown in Listing 4.15. The variable *randomValue* represents the randomized component scale value, calculated separately for the three component dimensions. The variable ϵ is the randomized eccentricity value that falls within the defined range the user has set for target creation. Taking the product between the calculated scale and ϵ values produces an intermediate variable c. The scale of the major axis, Z, is set to the randomized scale value as with any other object in the trial generation phase, while the two minor axes have a scaling applied that represents the square root of the difference between the square of the major axis length and the square of the intermediate variable c.

4.10 Mesh Input: OBJ Parser

4.10.1 OBJ File Format

A crucial requirement of the ETTS system is the ability to import mesh files for use in the design of scenarios. Mesh files are used to describe the geometric properties of objects that have been modelled using a program such as Maya or Blender and are typically represented as a collection of triangles or quadrilaterals (not limited to just these two). While the method for storing mesh data in files varies, the supported file format used by the ETTS is the open and widely supported OBJ file format. First developed by Wavefront Technologies, the format specifies a structure for describing mesh files, storing information about their vertices, faces, normals, texture coordinates, and more. The openness and wide support for this relatively simple format makes it a perfect choice for incorporating the ability to import mesh files.

The structure of an ascii encoded OBJ file varies slightly depending on the type of mesh file generated by the modelling package and the accompanying information. The main supported components of the OBJ file format by the ETTS are vertices, vertex texture coordinates, face elements, and vertex normals. Mesh files exported in the OBJ format must be triangulated (as opposed to quadrilaterals) to be supported by the ETTS. The role of the OBJ parser is to extract the available vertex position, texture, normal, and face information that make up a given mesh and store them as structures understandable by the render engine. An example of the OBJ format for a triangulated mesh is shown in Listing 4.16.

Listing 4.16: Excerpt from OBJ Mesh file for a textured cube

```
# Example of CUBE Mesh

v -4.901622 -4.184651 3.879749
v 4.901622 -4.184651 3.879749
v -4.901622 4.184651 3.879749
v 4.901622 4.184651 3.879749

vt 0.375000 0.000000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.625000 0.250000

vn -0.666667 -0.666667 0.333333
vn 0.408248 -0.408248 0.816497
vn -0.408248 0.408248 0.816497
vn 0.666667 0.666667 0.333333

f 1/1/1 2/2/2 3/3/3
f 3/3/3 2/2/2 4/4/4
f 3/3/5 4/4/6 5/5/7
f 5/5/7 4/4/6 6/6/8
```

Vertex positions are specified in terms of their cartesian coordinates and are represented in the mesh file by the lines having the prefix ***v***. Each specification of a vertex has three components; x,y,z with a fourth optionally specified which represents w used for homogenous coordinates. Vertex components are separated by spaces and are organized with a single vertex per line.

Vertex Texture coordinates are stored on lines that have the prefix ***vt*** and contain two attributes; u and v with a third optional w used for homogenous coordinates. The uv parameters specify the mapped texture coordinates for each vertex over a normalized range of 0 and 1.

Vertex Normals represent the normal vectors calculated at a given vertex and are identified on lines having a prefix of ***vn***, storing the x, y, and z components of the vector. Vertex normals may or may not be in a normalized form which needs to be accounted for when preparing to render.

Lastly, faces are defined using lines prefixed with ***f*** and are used to represent the triangular shaped faces of the mesh. Each face is comprised of three vertices whose order (usually counterclockwise) is important as it distinguishes front of the face from the back (useful in culling). The storage of vertices, normals, and uv texture coordinates as separate sections of the mesh file allows faces to reference the three vertex-related attributes by index; an important optimization to consider when each vertex can be shared by up to three different faces. While vertex specification is mandatory for face structure declarations, vertex normals and texture coordinates are optional. The example OBJ mesh file shown in Listing 4.16 illustrates the use of all three vertex-related attributes; each delimited with a forward slash and each vertex with a space.

The OBJ file format also provides the means for declaring material information along with an ability to define multiple objects and even groups of objects. Presently the ETTS system does not leverage this capability and therefore is not discussed in further detail other than to mention that it is available.

4.10.2 Parsing and Importing

Following the format outlined in Section 4.10.1 for 3D object mesh files stored in an OBJ formatted ascii file, implementation of a parsing module is required to read, parse, and convert the files into objects that can be rendered in ETTS scenes. The parsing process works by reading in the contents of the desired mesh file as an array of strings that have been grouped based on their line prefix within the inputted text file. Accepted prefixes are those discussed in Section 4.10.1. Vertices, texture coordinates, and vertex normals are stored in separate arrays as vector objects, holding the inputted coordinate components (i.e x, y, z, w, u, and v). Triangular mesh faces are stored in an array of indices that map to the previously inputted vertices.

Due to the way OpenGL requires Vertex Buffer Objects (VBO's, see Section 4.4) to be organized before being passed to the Graphics Processing Unit (GPU), shared vertices are not allowed. This means that storage optimizations in the OBJ formatted mesh files that allow multiple faces to share and index common vertices (without listing duplicate vertex information) needs to be reversed and expanded. While this expansion of vertex information results increased memory use for prepared mesh structures, overall render performance is increased by allowing fast sequential processing of vertices and accompanying information (i.e normals and textures, see Section 4.4 for a comprehensive overview of the rendering process and relevant structures).

The last step of the parsing and importing process is the allocation of an Entity object (see Section 4.5.1 for class specification) and configuration of the resources used for rendering. Rendering performance is increased by allocating the necessary VBO structures during the mesh importing process, thus removing the need for resource allocation and configuration during rendering. The availability of diffuse texturing in the scenario editor for each object imported is dependent on whether valid texture texture coordinates are found during the importation process. Similarly, the choice of shaders used during the rendering process is impacted by whether the mesh file provides vertex normals, determining the type of lighting to use (ambient or diffuse).

Chapter 5

Targeting Simulator: User Interaction

5.1 Gamepad Controller

The gamepad controller used in for development of the ETTS system is a standard issue USB controller developed by Microsoft for their Xbox 360 gaming console. The reason for choosing particular controller is that it contains all the desired capabilities for use with the simulator as well it is representative of the type of hardware that target users would likely be familiar with and easily accessible for purchase. Figure 5.1 illustrates an example of the controller used, providing a brief description of the controls used.

An important consideration with this controller is the lack of native support on the Apple OS X operating system (as of version 10.10.3) and thus the requirement to use third party drivers and development of a custom framework to handle user input. A detailed discussion of the developed gamepad controller framework is located in Section 5.1.1.

The concept of context driven behaviours is applied throughout all aspects of the ETTS system but is most apparent in the user interface module. For the mapping of gamepad controls this design allows a finite number of controls to be multiplexed, providing different functionalities depending on the current context of use. The control context can be selected by the user, allowing for control of the endoscopic tool and camera in alternating fashion. Selection of camera control vs tool control is done by pressing either of the shoulder buttons, **Shoulder L** and **Shoulder R** respectively (as shown in Figure 5.1).

The gamepad (shown in Figure 5.1) contains two analog sticks that we will refer to as **Stick-L** and **Stick-R**, each providing two degrees of freedom (only two of the available three are currently utilized). The left stick, **Stick-L** is assigned to provide translation/position control in both vertical and lateral directions. The right stick, **Stick-R** is mapped to provide rotational control, in particular pitch and yaw. Input produced through the movement of either analog stick generates a linear relation between stick position and the sampled value.

In order to provide an expanded input range for position and rotational control of the camera and endoscopic tool, the analog input from the sticks is passed through an exponential ramp function that allows for incredibly fine movement when a small amount of deflection is applied, and very rapid/gross changes when a large deflection is used. Listing 5.1 illustrates how this transformation occurs, with two different ramp functions being applied, creating separate profiles for rotation and translation actions. The two ramp functions applied are illustrated

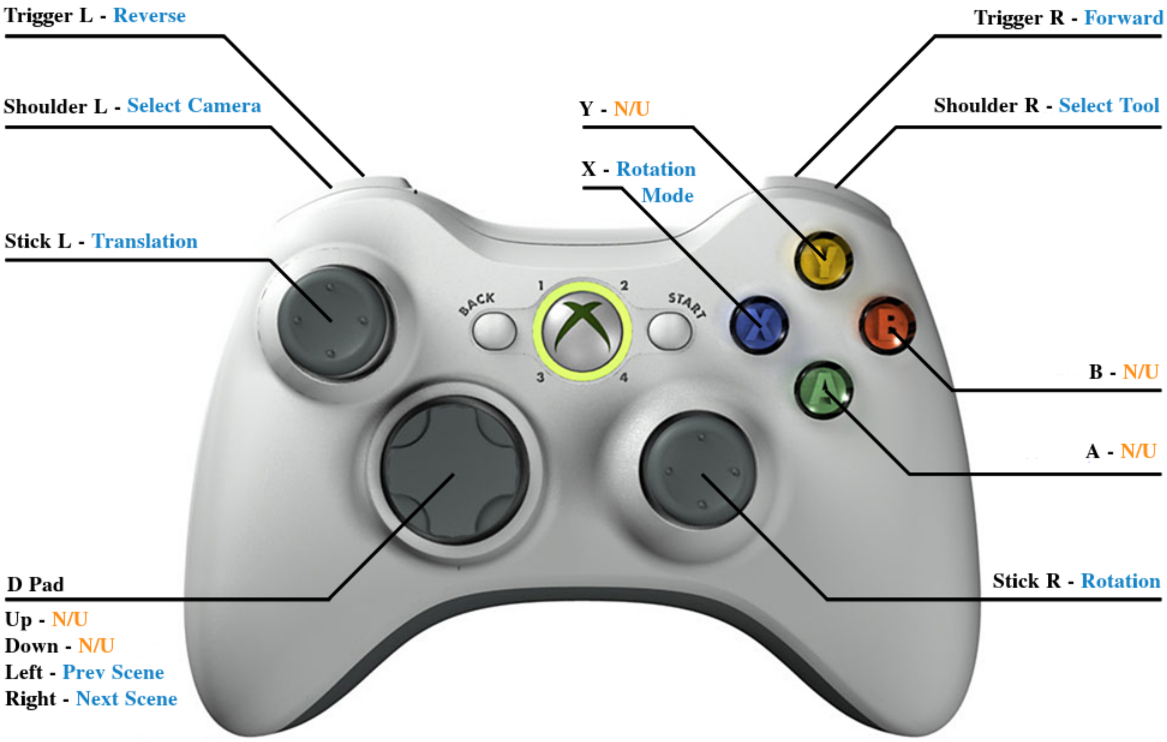
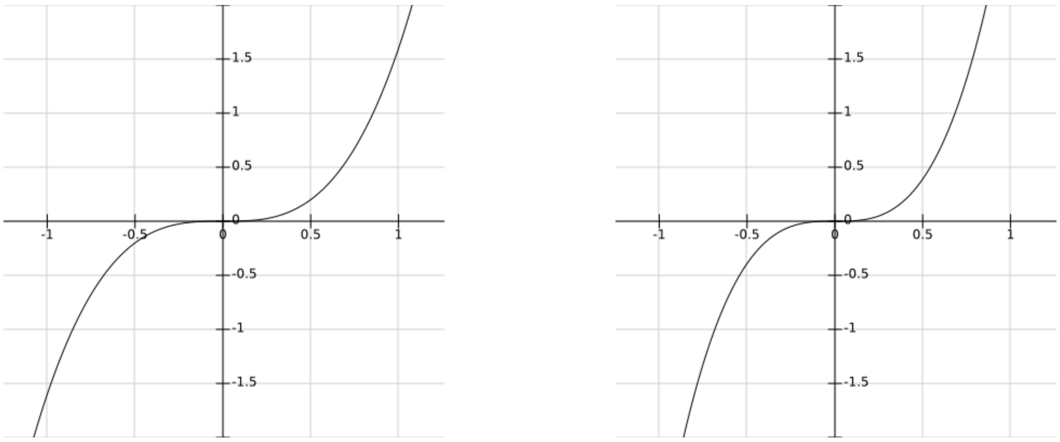


Figure 5.1: Gamepad Controller Mapping [18]

in Figure 5.2 where the x-axis denotes the raw input from the analog sticks and the y-axis is the transformed value.



(a) Camera translation input ramp

(b) Camera rotation input ramp

Figure 5.2: Gamepad input ramp functions for camera control

Similar input ramp functions are used to provide the same effect for the control of the en-

Listing 5.1: Applying Input Ramp Functions to Analog Sticks

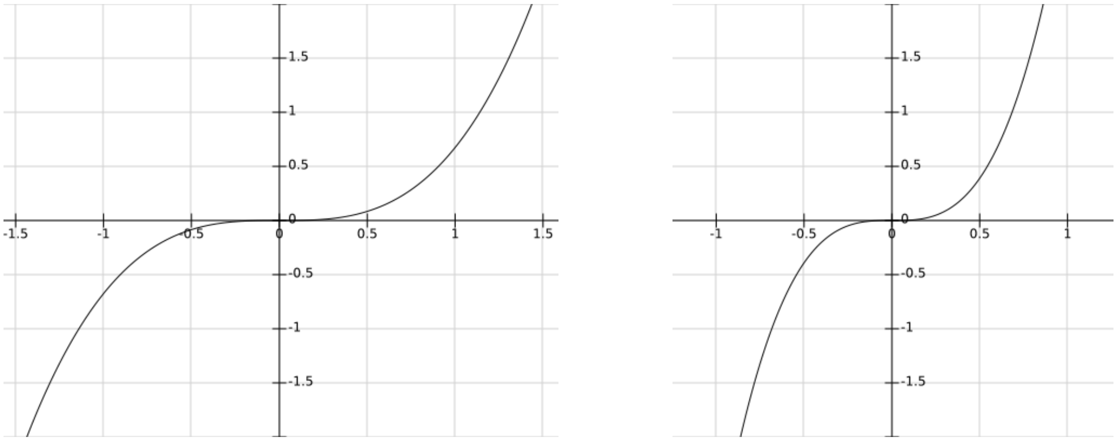
```
if (isControllingCamera)
    {
        [self setControllerObject_zoom:0.0f];
        pointL.y = 1.6f * powf(Y, 3);
        pointL.x = 1.6f * powf(X, 3);
        [self setController_origin:pointL];
        [self setController_zoom:(zoomIn-zoomOut)];

        pointR.x = powf(RX, 3);
        pointR.y = powf(RY, 3);

        rotation.x = -3.125f * pointR.x;
        rotation.y = 3.125f * pointR.y;

        [self setTopRight_rotation:rotation];
        [self setOGLViewIsDirty:YES];
    }
```

doscopic tool object, using a different profile for position input, following the general principal as used for the camera control input. The varied ramp functions are shown in Figure 5.3.



(a) Tool translation input ramp

(b) Tool rotation input ramp

Figure 5.3: Gamepad input ramp functions for tool control

The general equation of the input ramp functions used is a cubic as shown in Equation (5.1). The equation has a single real root of x=0 and relies on the first coefficient to set the desired range of sensitivity. Using a large coefficient will compress the slow/fine range of movements while a coefficient greater than 0 and smaller than 1 will expand the range.

$$y(x) = a * x^3 \tag{5.1}$$

The advantage of using ramp functions is the ability for the user to customize the sensitivity of the controls to suit their preferences. Configurable sliders in the controller preferences section of the ETTS system allows the operator to augment the default preferences, either compressing or expanding the range of fine and gross movements. One other application of variable input ramp functions is to vary input sensitivity settings based on contextual behaviours, such as using rapid/gross movements for burr hole targeting and slow/fine movements for intracranial targeting tasks.

The left and right triggers of the controller, denoted as **Trigger L** and **Trigger R** respectively in Figure 5.1 provide analog input ranging in value from 0 at rest and to 255 when fully depressed (8-bit sampling resolution). The input from these triggers are mapped to the remaining translational degree of freedom that can't be expressed using the two degree of freedom (DOF) analog sticks, advancing the camera and/or tool with **Trigger R** and reversing with **Trigger L**.

When used in a simulation state where a trial is being performed, pressing the left and right directions of the digital pad on the controller, denoted as **D Pad** in Figure 5.1, results in starting the next or previous scene. Each time the next scene is loaded, the recorded data from the previous one is serialized and written out of memory and into encoded data files (see Section 4.8 for further details)

The rotation of either the endoscopic tool or the camera can be performed in two very different ways depending on the rotation mode that the operator has set. The type of rotational model used can be toggled between the two styles by pressing the blue coloured **X** button (see Figure 5.1 for button layout).

World rotation (the default rotational mode for camera control) causes the camera to make yaw and pitch rotations about a bound object such as the manikin head (bound coordinates are set to use the origin of the scene by default), as though the camera was orbiting the bound coordinates. This allows the camera to encircle the target coordinates while always maintaining a gaze towards the centre around which it is rotating. Listing 5.2 illustrates how rotation of the camera object is applied when set to rotate around a bound object or set of coordinates.

Listing 5.2: World Rotation About a Bound Object

```
    GLKMatrix4 rotation = GLKMatrix4RotateWithVector3(
        GLKMatrix4MakeTranslation(PX, PY, PZ), (M_PI/180.0f) * ROTy,
        GLKVector3Make(viewMatrix.m00, viewMatrix.m10, viewMatrix.m20));

    rotation = GLKMatrix4RotateY(rotation, (M_PI/180.0f) * ROTx);

    viewMatrix = GLKMatrix4Multiply(viewMatrix, rotation);

    viewMatrix = GLKMatrix4Translate(viewMatrix, -PX-viewMatrix.m00 * PH
        , -PY-PV, -PZ-viewMatrix.m20 * PH);
```

Local rotation (the default rotational mode for tool control) causes the tool to make yaw and pitch rotations in local space, within its own frame of reference. The point of rotation is defined to be the origin of the mesh representing the object; in the case of the endoscopic tool mesh, this is the tool's tip position. Listing 5.3 illustrates how rotation of the camera object is applied when set to rotate within its own frame of reference, performing local rotation.

Listing 5.3: Local Rotation of Camera Object

```
    viewMatrix = GLKMatrix4Translate(viewMatrix, [self CAM].x, [self CAM].y, [self CAM].z);

    GLKMatrix4 test = GLKMatrix4RotateWithVector3(GLKMatrix4MakeTranslation(0,0,0), (M_PI/180.0f) * ROTy, GLKVector3Make(viewMatrix.m00, viewMatrix.m10, viewMatrix.m20));

    viewMatrix = GLKMatrix4Multiply(viewMatrix, test);

    viewMatrix = GLKMatrix4RotateY(viewMatrix, (M_PI/180.0f) * ROTx);

    viewMatrix = GLKMatrix4Translate(viewMatrix, -[self CAM].x - viewMatrix.m00 * PH, -[self CAM].y - PV, -[self CAM].z - viewMatrix.m20 * PH);
```

5.1.1 Gamepad Controller: Framework

The lack of native support for the USB Gamepad controller used for the development and testing stages of the ETTS system (see Figure 5.1 and Section 5.1) made it necessary to develop a framework that handles the user interaction with the controller. The Gamepad Manager Framework (GMF) is a portable module that offloads the direct event handling and interfacing from the simulator and in turn provides a protocol for actions to be sent to the simulator when new input is available. This removes the need for input polling and device interrupt handling. Data interpreted from the controller is made accessible to the simulator through a key-value pair dictionary housed within a new class of object, **Gamepad** (see Figure 5.4 for class diagram and structure).

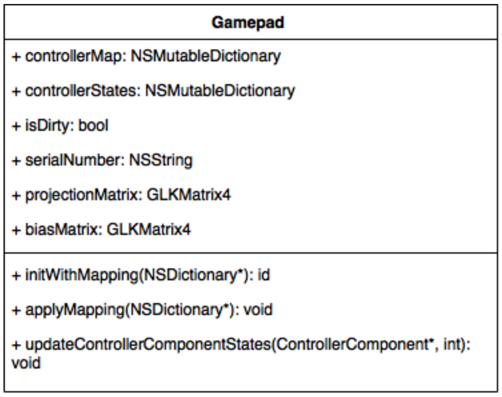

Gamepad
+ controllerMap: NSMutableDictionary
+ controllerStates: NSMutableDictionary
+ isDirty: bool
+ serialNumber: NSString
+ projectionMatrix: GLKMatrix4
+ biasMatrix: GLKMatrix4
+ initWithMapping(NSDictionary*): id
+ applyMapping(NSDictionary*): void
+ updateControllerComponentStates(ControllerComponent*, int): void

Figure 5.4: Class diagram of Gamepad object

The design of the framework is made to be as versatile as possible by leveraging input mappings for control bindings, allowing in theory for any usb gamepad controller to be installed and configured and not limiting hardware support to the Xbox 360 controller showed in Figure 5.1. This is done through the use of Apple's **IOKit** and specifically the **IOHIDLib** library which provides a framework in itself to interface with Human Interface Devices (HID's) in general. Data from the connected devices can be read in the form of pages that provide component states (i.e button up or down, analog value of stick position, etc...) as well general device information such as manufacturer, device serial number and the classification of device.

Using a separate application, developed for the sole purpose of generating a input map file, a serialized structure of the various controls available along with their characteristics can be saved as a profile to be used by the GMF to support a specific Gamepad device. An example of the stored profile for the device used (Figure 5.1) can be seen in Listing A.3 of Appendix A.

The Gamepad inputs can be visualized and organized into a set of controller components, each having the same structure and attributes. For this, a new class of object, **ControllerComponent** is used to represent these different sub components of a connected device. The object's structure is illustrated through the class diagram shown in Figure 5.5

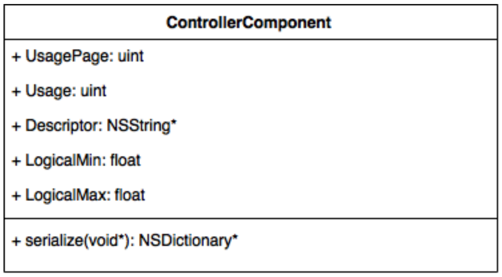


Figure 5.5: Class diagram for ControllerComponent object

Each Controller Component defines a *Usage*, *Usage Page*, *Descriptor*, *Logical Min*, and *Logical Max*. These attributes are detailed as follows:

Usage: Defines the index of component using an integer value, providing each control component (button, stick, trigger, etc..) with a unique identifier in the form of an integer.

Usage Page: An integer value that defines the class or type of input that the control belongs to; for example a usage page value of 1 is used to identify analog inputs (as seen in Listing A.3 of Appendix A) while a value of 9 identifies the type of input as a button.

Descriptor: A string used to provide a textual name or identification of the component (i.e D-RIGHT for the right direction of the digital pad) .

Logical Min/Max: These values are used to store the minimum and maximum values respectively, that the component can take on.

Whenever the GMF receives an event from the connected controller, the element responsible for generating it is recorded and used to populate a new Controller Component if no record exists in the Gamepad object's list, or simply update the state information of the existing one if it does. One additional step that the GMF takes before notifying the simulator about the new data is to perform pre-processing of analog input data; In particular, there is a degree of drift that exists in the analog components when at rest. The solution used for removing the erroneous noise is to create a dead zone through the use of thresholding. An example of this thresholding process is shown in Listing 5.4.

Listing 5.4: Preprocessing of Analog Stick Input

```
if ([CC UsagePage] == 1 && [CC LogicalMax] > 255)
        {   //Analog Stick Input
            float max = [CC LogicalMax];
            float min = [CC LogicalMin];
            float value = val;

            value = -2 * (val - min) / (max - min) + 1.0f;

            if (abs(value * 10) < 1.5f)
            {   //Handle Dead Zone Drift
                value = 0.0f;
            }
            [self.controllerStates setObject:[NSNumber numberWithFloat:value] forKey:[NSString stringWithFormat:@”%u”, [CC Usage]]];
        }
```

The first step is to normalize the input between [-1, 1] which standardizes the values, making the framework more robust to different controller versions or even types that may have different sampling resolutions. Once normalization is performed, thresholding is used to create a dead zone around the analog device's at rest state, thus removing erroneous input caused by analog inputs drifting around their origin. One obvious tradeoff is that if set too large, valid input is also discarded, thus the values used are chosen through empirical testing and calibration. While at a testing stage this method is acceptable, it should be noted that performance and robustness could be improved by allowing the user to calibrate the controller each time it is used. The last step is to update the state of the current component to yield the new value.

Use of the GMF requires that the application delegate implements the functions that the framework exports, which are: *newDeviceAvailable*, *hasDataToBeRead*, and *deviceRemoved*.

Listing 5.5: Handling of New Device Available Event

```
-(void)gamePadManager:(GamePadManager*)GPM newDeviceAvailable:(NSString*)serialOfDevice
{
    NSLog(@”Added”);
    if ((int)[inputDevice indexOfItemWithObjectValue:@”Gamepad”] < 0)
    {
        [inputDevice addItemWithObjectValue:@”Gamepad”];
    }
}
```

Listing 5.5 illustrates the function called whenever a new device is connected and the subsequent event is generated by the GMF. The addition of a new gamepad device adds an instance of the new device in the ETTS, making it available as a bindable input device.

Listing 5.6: Handling of Has Data To Be Read Event

```
-(void)gamePadManager:(GamePadManager*)GPM hasDataToBeRead:(NSString*)
    serialOfDevice
{
    GamePad *GP = (GamePad*)[[GPM controllers] objectForKey:
        serialOfDevice];
    bool XB = [[[GP controllerStates] objectForKey:@"3"] boolValue];
    CFTimeInterval elapsed = CACurrentMediaTime() - startTime;
    if(isControllingCamera && XB && elapsed > 0.3f) //Debounce
    {
        startTime = CACurrentMediaTime();
        [[self OGLView] toggleGlobalCamera];
    }
```

Changes to the Gamepad controller inputs are communicated to the simulator by calling the notifying function *hasDataToBeRead*, an excerpt of which is shown in Listing 5.6. An example of the input handling performed within this event handler was discussed in Section 5.1. Another example of the operations performed upon the *hasDataToBeRead* event being called for button presses is the debounce operation shown in Listing 5.6. This type of filtering is applied to ensure that multiple button press events aren't generated in error on a single event.

Listing 5.7: Handling of Device Removed Event

```
-(void)gamePadManager:(GamePadManager*)GPM deviceRemoved:(NSString*)
    serialOfDevice
{
    NSLog(@"GamePad Disconnected!");
    if ((int)[inputDevice indexOfItemWithObjectValue:@"Gamepad"] >= 0)
    {
        [inputDevice removeItemWithObjectValue:@"Gamepad"];
    }
}
```

Listing 5.7 details the function called whenever a connected device becomes unavailable and the resulting event is generated by the GMF. Handling of this event is used to remove the device from the ETTS's list of valid input devices.

5.2 Leap Motion Controller

The support for user interaction through the movement and orientation of a user's hands is accomplished with the use of the Leap Motion Device (LMD). The device makes it possible to track hands and tools within the sensor's field of view, providing position and rotation data on items being tracked. Figure 5.6 provides an illustration of the device along with the coordinate system used for recording position data.

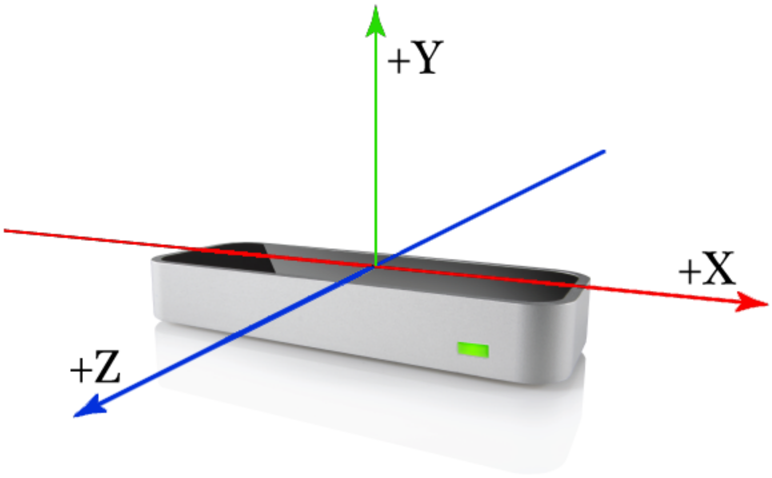


Figure 5.6: Figure showing Leap Motion Device and Coordinate System [21]

Support for the LMD in the ETTS system is through the *LeapMotionController* class, which is used to handle the input events generated by the device. The LMD works by notifying the *LeapMotionController* whenever a new frame of data is available to read. Contained within the frame of data (a data structure defined within the Leap Motion API) is an array of hands (if present) which themselves are data structures that hold among other things the position of the centre of the palm and the corresponding *pitch*, *yaw*, and *roll* angles.

Position and Rotation information captured from the LMD is utilized by the ETTS to manipulate the corresponding attributes of the objects to which the LMD input is bound to. The default operation of the LMC is configured to use two-handed input, binding the user's left hand to camera control and the right hand to tool manipulation. Configuration of the ETTS attributes editor (can be seen in Figure 4.14) allows for flexible binding options with respect to the user's preference of bindings. This is done to take care of any perceived bias caused by the dominant side (left handed vs. right handed) of the user. It also adds forward compatibility with an ability to easily bind LMD input to objects other than the designated tool and camera of the scene (i.e. Endoscopic camera or trocar control).

Listing 5.8 illustrates how the LMC updates position and rotation data within the ETTS for objects bound to the LMD input device. The particular example is for controlling the simulator camera, the same process is repeated for tool control. To reconcile the difference in the world space used by the LMD and that of the ETTS, calibration values that represent a constant offset is used and can be set by the user at the begin of a new trial. This allows the user to define where they feel the origin should be in relation to their hand's position over the sensor. After setting the rotation angles corresponding to the hand's *pitch* and *yaw* angles, the render engine is notified that the current view is dirty and thus needs to be refreshed.

Listing 5.8: LMC processing of position and rotation data

```
[[delegate_handle cameraPosition] setX:(1.0f*[currentTool tipPosition].x
    - [[delegate_handle LMCCalibrationValues] X])];
[[delegate_handle cameraPosition] setY:(1.0f*[currentTool tipPosition].y
    - [[delegate_handle LMCCalibrationValues] Y])];
[[delegate_handle cameraPosition] setZ:(1.5f*[currentTool tipPosition].z
    - [[delegate_handle LMCCalibrationValues] Z])];
if (fabs([currentTool direction].yaw - _prevCameraRotation.x) > 0.001f)
{
  [[delegate_handle cameraRotation] setX:([currentTool direction].yaw +
      _prevCameraRotation.x) * 180.0f/M_PI];
  _prevCameraRotation.x = [currentTool direction].yaw;
}
if (fabs([currentTool direction].pitch - _prevCameraRotation.y) > 0.001f
    )
{
  [[delegate_handle cameraRotation] setY:-([currentTool direction].pitch
      + _prevCameraRotation.y) * 180.0f/M_PI];
  _prevCameraRotation.y = [currentTool direction].pitch;
}

[delegate_handle setOGLViewIsDirty:true];
```

5.3 Voice Control

The use of voice control is required as an additional user input when the ETTS is being used in conjunction with two-handed input provided by the LMC. The functionality of the voice control module serves to provide hands free control of the simulator by simply issuing audible commands into the microphone of the computer being used. A listing of these commands is provided below with a brief description of each:

Begin: issued to start recording current trial session.

Next: instructs the simulator to conclude the current scenario, export and save the captured session data files and advance to the next scenario.

Previous: instructs the simulator to conclude the current scenario, export and save the captured session data files and advance to the previous scenario.

Done: this command will conclude the current scenario, export and save the captured session data files.

Quit: issued to tell the simulator to exit simulation mode and return to the editor.

Speech recognition makes use of Apple's *SpeechRecognizer* library that is openly available for development in applications requiring speech recognition for commands and supports operation in both online and offline settings. A separate framework, *SpeechController* was implemented as a wrapper to the *SpeechRecognizer* library in order to preserve the modularity of the ETTS system. This *SpeechController* framework handles the event handler methods associated with using speech recognition as well serves to initialize the speech recognition system upon starting a new trial and configures the keywords available to the user.

5.4 Multi Touch Trackpad Support

The target platform for the Endoscopic Targeting Tasks Simulator is the Apple OS X operating system and is designed to leverage additional input capabilities supported by the operating system, such as the adoption of multi touch trackpad input devices. These devices expand on the simple capabilities of a traditional mouse by allowing the use of gestures to change the context in which the user interface performs. The two main gestures supported are pinch zoom and two-finger drag. The view to which mouse and trackpad input is applied to will depend on the location of the pointer at the time the gesture begins. Selecting which camera's perspective to control is simply a matter of placing the mouse pointer over the appropriate viewport.

The pinch zoom gesture is one that users have likely become familiar with in recent years with it's wide spread use in smart phones and tablets to allow a user to make a closing or opening pinch gesture with two fingers to zoom the view in and out. This gesture has been applied in the ETTS user interface (when supported) to control the scene camera's translation along the viewing direction, dollying the camera forward and backwards.

The two-finger drag gesture is used to apply two different functionalities to the ETTS system providing translation control of the scene camera (both in perspective and orthographic views) in the remaining two degrees of freedom for translation. When the same gesture is combined with the control key and applied over one of the three orthographic views, vertical movement along the trackpad is used for the control of orthographic slice positions (when enabled); Slices are advanced or retracted based on the user's input.

Chapter 6

Results

6.1 Targeting Tasks: User Input

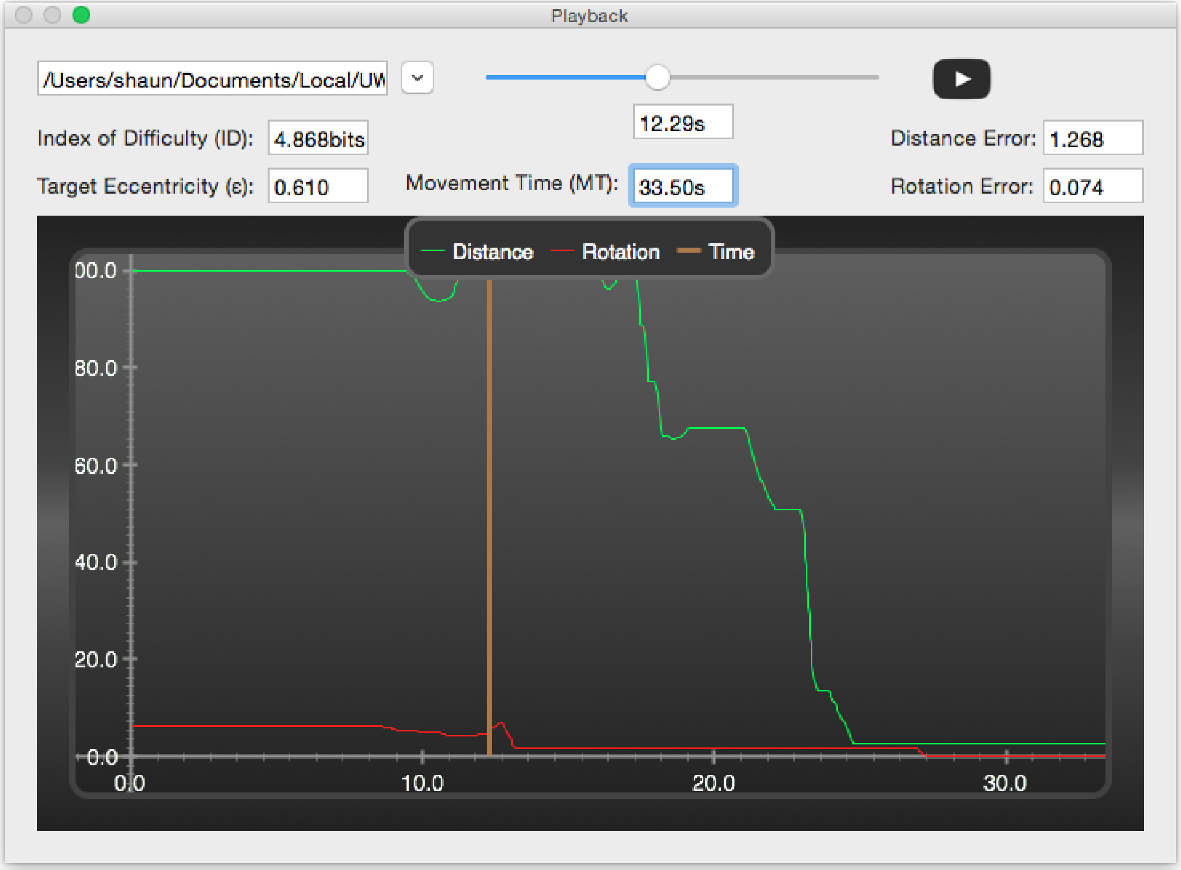
The goal with the development of the ETTS system was to develop a platform custom tailored to the purpose of evaluating user performance in targeting-based tasks. Of interest was the means to understand the impact of different input modalities not necessarily on user performance, but rather what impact they have on the overall efficacy of a targeting-tasks simulator designed to improve a user's skills that are required for the tasks at hand. In particular we looked at two forms of input that would be considered readily available to potential users of the ETTS, without incurring the relatively high costs associated with haptic interfaces available on commercial simulators. The input controllers chosen for study were the Leap Motion controller, which is considered to be a more natural and authentic form of input, and the Xbox 360 Gamepad, a popular peripheral that represents a design and layout likely similar to what most people would be familiar with that play video games.

The use of the Gamepad peripheral provided an effective means for giving the user control of the 5 DOF necessary to operate both the camera and tool objects. The one potential drawback of the gamepad over the LMD is its inability to provide simultaneous control of the camera and tool objects, leading to the multiplexed approach that requires the user switch between the target object being controlled. The result of this multiplexing can be seen immediately by studying the plotted analysis and playback view of the *Playback and Analysis* module shown in Figure 6.1

Referring to Figure 6.1 there are a few observations that can be made, most prominent of which is the delay of any change in position and rotation error at the start of the trial and throughout the recorded trial. This can be attributed to the instances corresponding to the user manipulating the camera's orientation. This time taken to manipulate the camera obviously detracts from the user's performance by consuming valuable time moving the camera rather than moving the tool into position.

The observation made is in the distinctive step pattern to both rotation and position error values, and upon further investigation reveals an interesting observation. While it is possible to both translate and rotate the tool simultaneously, the initial pattern seems to be that users perform these actions separately.

Lastly, as predicted by Fitts' methodology, there is a visible pattern within the rate of rota-



/Users/shaun/Documents/Local/UW

12.29s

Index of Difficulty (ID): 4.868bits

Target Eccentricity (ε): 0.610

Movement Time (MT): 33.50s

Distance Error: 1.268

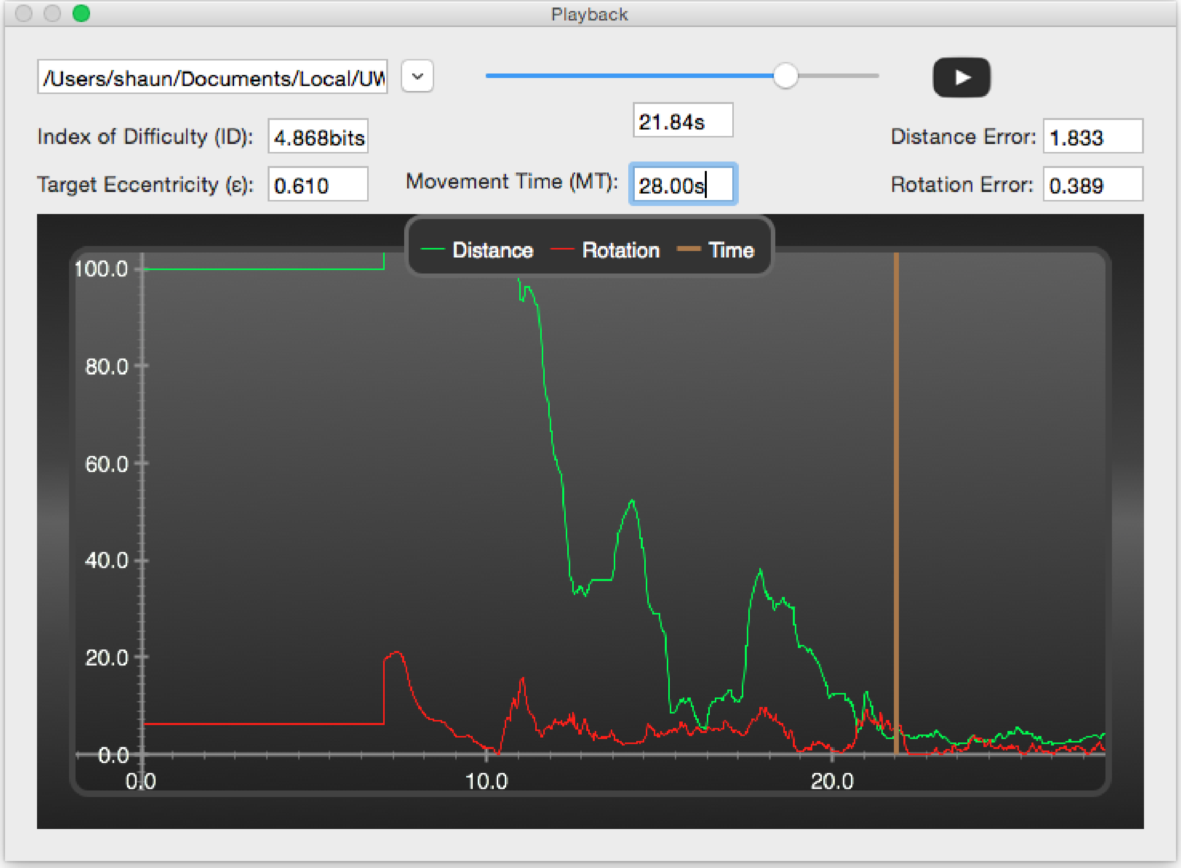
Rotation Error: 0.074

Figure 6.1: Analysis of ellipsoid targeting task using gamepad input device.

tion and translation as a function of time along with the stage in which each occurs. The initial trend seems to suggest that users start with a broad projectile approach, moving quickly at the start of the task and decreasing speed as the tool approaches the target, yielding to finer and slower movements to achieve greater accuracy. The other observation is that control of position and rotation seems to occur at different stages of the trial; the start of the trial seems to place emphasis on translation of the tool while rotation doesn't occur until the position of the tool passes some threshold of position error.

In contrast to the gamepad controller, the LMD offers a very different user experience with a focus on a more natural and authentic interface. The default configuration of the LMD is to make use of a two-handed style of input, mapping the user's left hand to camera control and the right hand to the targeting tool. A caveat of requiring the user to use both hands simultaneously is that an additional user input in the form of speech control must be used to allow for the manipulation of simulator controls (start, next, pause, etc...).

The main differences we were interested in evaluating was concerned with how the difference in input modalities would impact not only user performance but more importantly the efficacy by which clinically relevant and transferrable skills could be improved. The natural mapping of the LMD interface is designed to mimic as closely as possible the actual movements a trained clinician would make throughout the simulated surgical procedure (ETV). Figure 6.2 illustrates the results of the same trial presented in Figure 6.1 when performed with the LMD.



/Users/shaun/Documents/Local/UW

Index of Difficulty (ID): 4.868bits

Target Eccentricity (ε): 0.610

21.84s

Movement Time (MT): 28.00s

Distance Error: 1.833

Rotation Error: 0.389

Figure 6.2: Analysis of ellipsoid targeting task using Leap Motion input device.

Referring to Figure 6.2 it is immediately apparent that the user's approach in tool position and rotation is much smoother, suggesting that fine adjustments to rotation are being made as tool translation is occurring. The opposite also seems to occur, with the user making fine translation adjustments while seeking the correct rotational angle.

The other large discrepancy we noticed when comparing the LMD results with that of the gamepad, is that the former method lacks the distinctive pauses in tool manipulation that correspond to the user toggling between camera control. The result with the LMD trials is that the user appears to make continuous iterative adjustments to camera placement in order to optimize the perspective of the tool approach throughout the duration of the trial.

6.2 Playback and Analysis

The addition of the *Playback and Analysis* module of the ETTS system is designed to enhance the effectiveness with which the simulator can be used to increase user performance in targeting tasks. Taking the approach that the module can be used as a debriefing tool, the goal was to provide functionality that could lead to reduced training time required to see improvement in user skills.

The functionality of the *Playback and Analysis* provides users with the ability to not only replay past trial attempts, it instead provides an enhanced view of the user's trial through the use of an interactive design. Examples of the interface are shown in Figures 6.1 and 6.2.

The effect of having the user be able to directly relate their performance as a function of time while observing the playback of those actions remains to be seen through further testing and research, but the hypothesis is that this added ability will lead to improvement in performance in a shorter period of time and fewer simulations than without. Not only is the user able to identify key areas in the analysis plot as a function of time, they also have the ability to seek to the desired instance in time to review tool and camera placement. Furthermore camera position can at any point in the playback be manipulated to provide a better vantage point if one exists. The same ability applies to tool positioning, allowing the user to reposition the tool to better understand an improved approach.

6.3 Trial Data Acquisition

Of crucial importance to the design of the ETTS and for that matter any simulator is the ability to capture and store data that is generated when running trials. This acquisition of data is necessary for not only the evaluation of user performance but it is critical to the evaluation of the efficacy for the improvement of clinically relevant skills through the repeated use of the ETTS.

The requirement to collect trial data is accomplished by the ETTS system in accordance with a format that is virtually lossless in the resolution of data generated while striving to produce an assortment of formats that ensure ease of analyses through third party tools. Data is stored in both CSV and JSON formats making it easy for results to be aggregated together for further analysis.

The way in which data and user actions is captured and stored also lends itself well to the ability to playback and review trial attempts from within the ETTS platform as discussed in Sections 4.8.4 and 6.2. An example of the results of the data acquisition portion the the ETTS system is shown in the JSON encoded Listing 4.5 which represents the hierarchy of the scene graph recorded as a sequence of captured states.

6.4 Application of Fitts' Law

The ability of the ETTS system to provide an objective measure of both human and simulator performance is demonstrated through the application of Fitts' methodology, using the acquired data from a trial designed to test both input modalities over a broad range of target difficulties. The dataset used in the trial consists of 30 randomly generated scenarios that use an ellipsoid target having variable position, scale, rotation, and eccentricity. The control over the target's eccentricity has a direct correlation to the 1DOF afforded in Fitts' original experiments where the target width is varied to produce a range of targeting tasks having different indexes of difficulty (*ID*). Figure 6.3 provides a histogram of the range of target eccentricities used in the trial, while Figure 6.4 gives the randomized distribution of target eccentricities throughout the 30 scenarios. The range of target eccentricities chosen for the generation of the 30 scenarios was selected to be [0.3, 0.9], constraining the upper and lower bounds to exclude values that make identification of the longest axis either too difficult or easy.

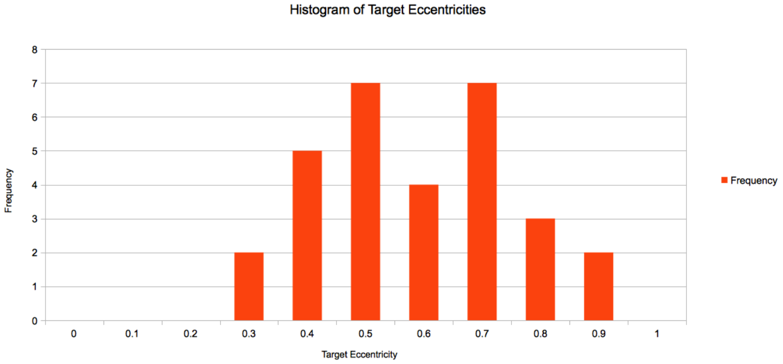


Figure 6.3: Histogram of target eccentricity values

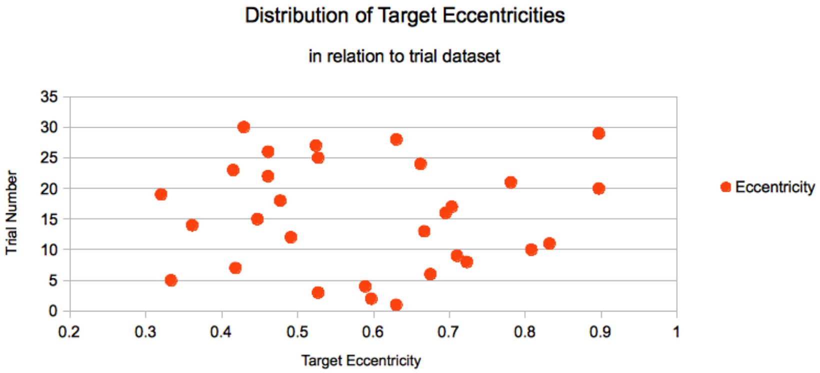


Figure 6.4: Distribution of target eccentricities throughout the dataset

The design of the scenarios requires that the user first identify the longest axis of the ellipsoid target using only the three orthographic slice plane views of the ETTS interface. Having identified the longest axis of the target contained within the skull mesh object, the user must then make use of the information from the three orthographic views to determine both the position and rotation of the target as projected onto the surface of the outer skull mesh. The targeting task is therefore composed of two phases: setup time (*ST*) which is the time taken to identify the longest axis, and movement time (*MT*) which corresponds to the time taken from when the tool is moved from its starting position to the final targeted position. Figures 6.5 and 6.6 provide a comparison between total time (*TT*), *MT*, and *ST* for both the Gamepad and Leap Motion input devices respectively.

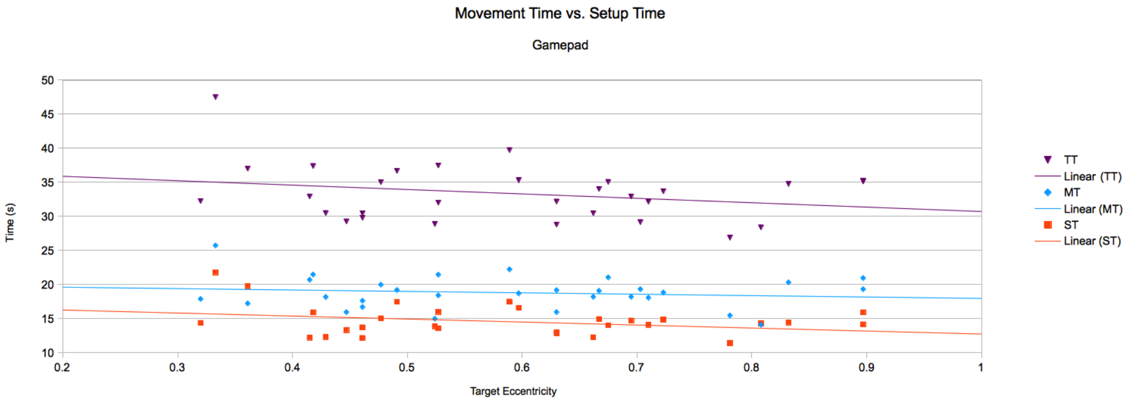


Figure 6.5: Comparison of *Total Time*, *Setup Time*, and *Movement Time* for Gamepad input device as a function of target eccentricity

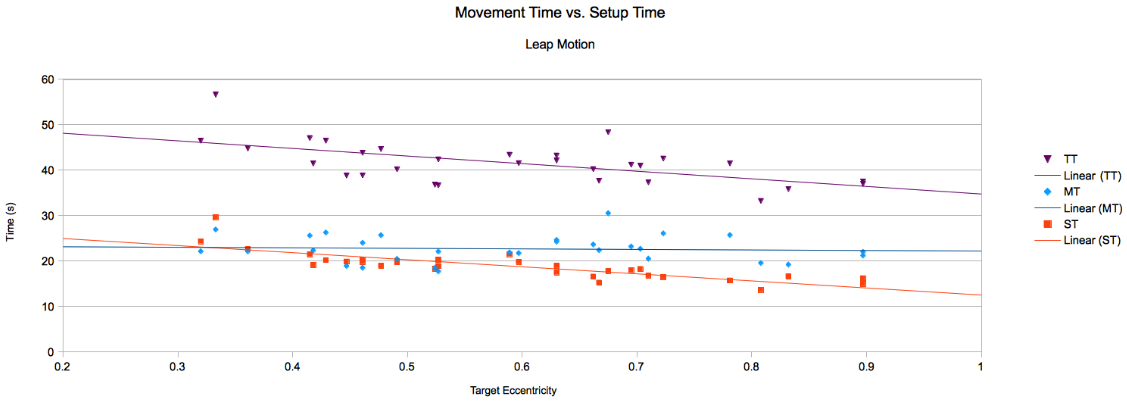


Figure 6.6: Comparison of *Total Time*, *Setup Time*, and *Movement Time* for Leap Motion input device as a function of target eccentricity

Comparing the results shown in Figures 6.5 and 6.6, it is apparent that *MT* is largely independent of both target eccentricity and input modality (Gamepad vs. Leap Motion). This observation agrees with the hypothesis which is that *ST* and therefore *TT* has a stronger dependence on target eccentricity as compared with the second, targeting phase of the task. The values used in the calculation of *TT*), *MT*, and *ST* represent the mean values for the individual scenarios of each trial.

Looking at the mean values for *TT*, *MT*, and *ST* for both input modalities, initial results show a 30% increase in *ST* when using the LMD. *TT* and *MT* also show increases of 21% and 25% respectively when using the LMD. Evaluating the two input modalities based solely on speed alone suggests that the gamepad device delivers superior performance when compared with the LMD.

Referring to Figures 6.7 and 6.8, the results of running three successive trials using both the gamepad and LMD controllers illustrate the change in user performance through the repeated use of the ETTS system. The trend of both input modalities suggests that repeated use of the ETTS system results in a noticeable change and improvement in the accuracy attained for both position and rotation errors.

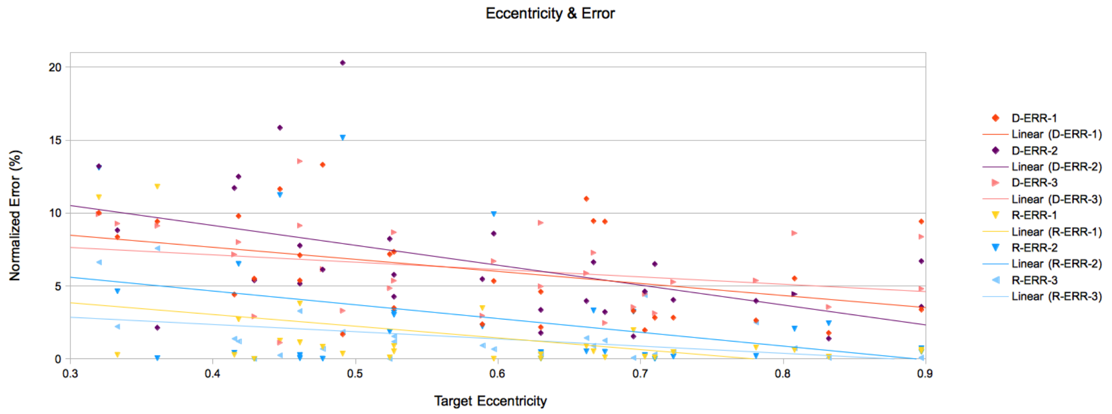


Figure 6.7: Results of normalized Distance and Rotation error (D-ERR and R-ERR respectively) recorded over successive trials as a function of target eccentricity using the gamepad controller

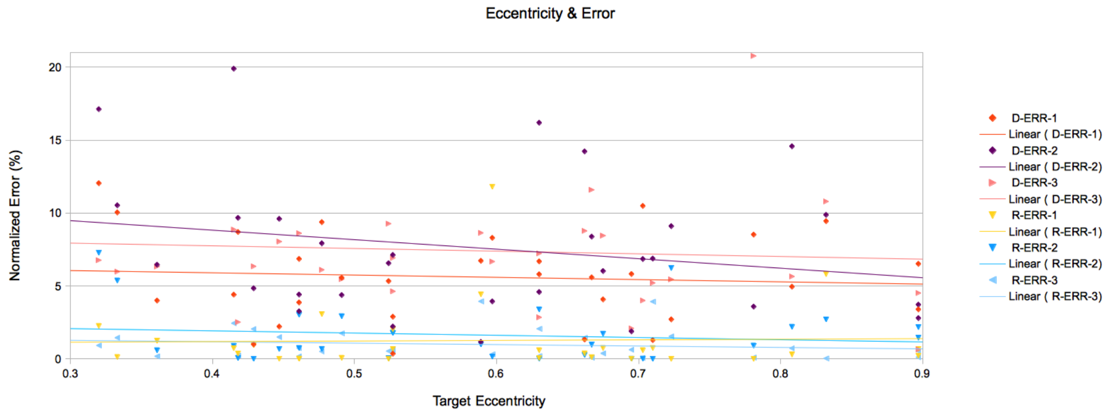


Figure 6.8: Results of normalized Distance and Rotation error (D-ERR and R-ERR respectively) recorded over successive trials as a function of target eccentricity using the LMD

The general trend for both position and rotation errors is to decrease, becoming more accurate as target eccentricity increases. Increasing target eccentricity provides a stronger cue for the identification of the longest axis, better indicating the position and rotation of the target located within the skull mesh object. The result of improved user performance is observed in Figures 6.7 and 6.8 with the slope of normalized error / target eccentricity becoming less pronounced and less sensitive to variations in eccentricities.

Using Fitts' Law for the analysis of user performance requires the calculation of a task's *ID* and *MT*. The form of Fitts' Law used for this research study was presented in Equation (2.4), where the target width is replaced with a calculated effective width using Equation (2.3). Using the data acquired from three successive trials for each input device, the effective target size and thus *ID*, is calculated and compared with target eccentricities in Figures 6.9 and 6.10.

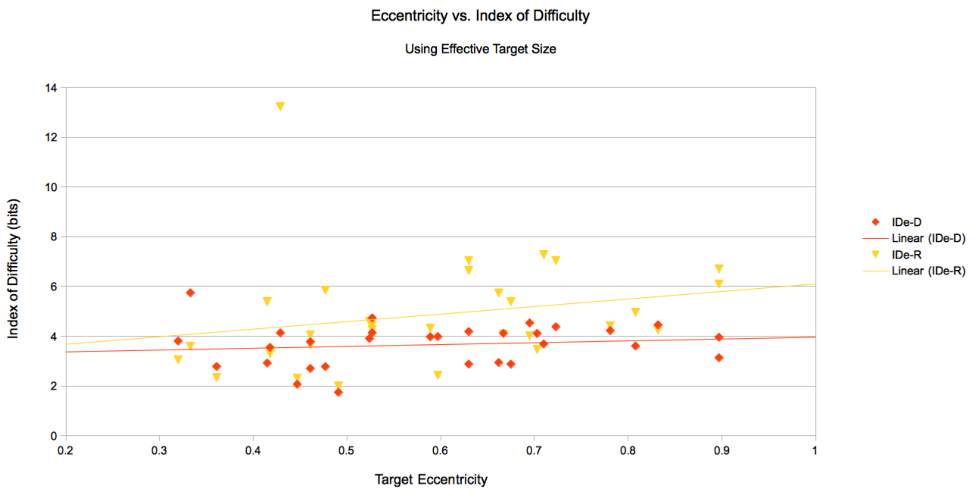


Figure 6.9: Plot showing the relation between the calculated distance and rotation indexes of difficulty and that of the target eccentricity using results from the gamepad trials

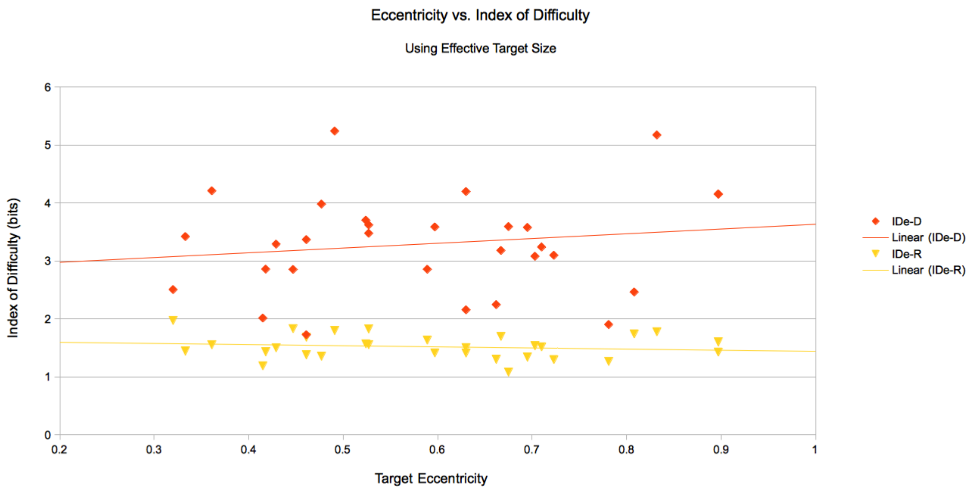


Figure 6.10: Plot showing the relation betweeh the calculated distance and rotation indexes of difficulty and that of the target eccentricity using results from the LMD trials

Referring to Figures 6.9 and 6.10, observed results show that for both input modalities, increasing the target eccentricity results in an increased index of difficulty for position selection. The relation of the index of difficulty to target eccentricity for rotation selection differs between the two input devices; results with the gamepad providing a positive correlation and the opposite for the LMD results. An unexpected observation with the LMD results is that unlike the calculated *ID* for position selection, the *ID* calculated for rotation tends to decrease as target eccentricity increases.

The use of effective target width/size for the calculation of *ID* ensures that the *IP* calculated represents the response for the task the user completed as opposed to what they were asked to do. Using Equation (2.3), Figures 6.11 and 6.12 illustrate the relation between normalized position and rotation error and the calculated *ID* for both input modalities.

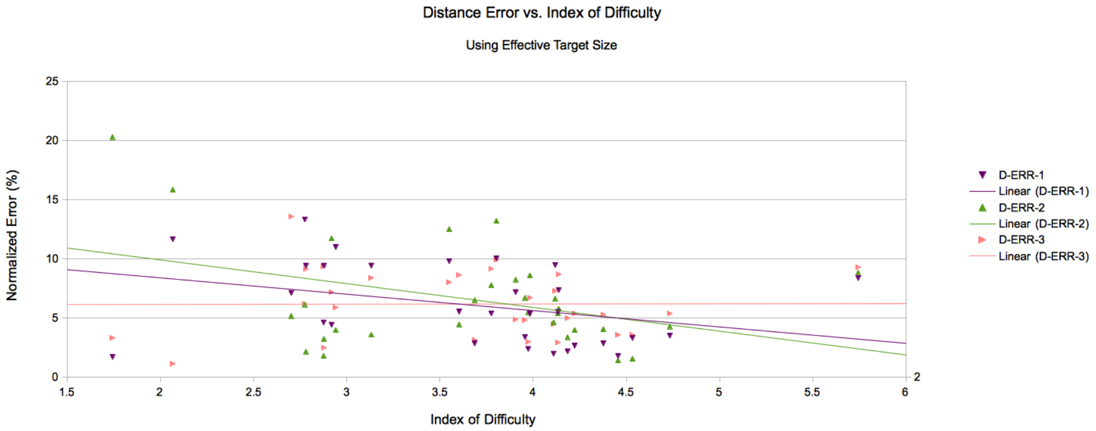


Figure 6.11: Plot showing the relation of normalized distance error to the calculated distance and rotation indexes of difficulties using results from the gamepad trials

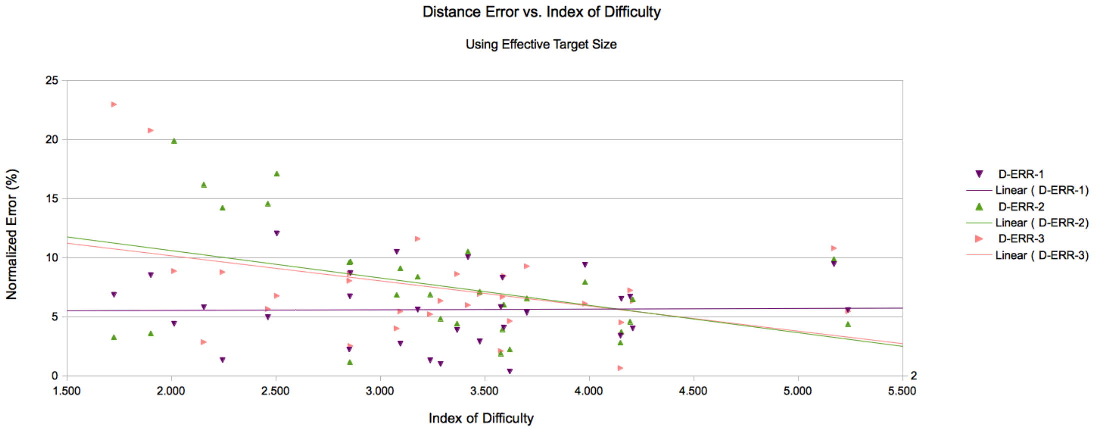


Figure 6.12: Plot showing the relation of normalized distance error to the calculated distance and rotation indexes of difficulties using results from the LMD trials

Initial results illustrated in Figures 6.11 and 6.12 demonstrate a clear trend of decreased error rates as the calculated *ID* increases. While this might seem to contradict initial expectations, substituting Equation (2.3) into Equation (2.1) provides a clear explanation for the observed trend. The scaling of target endpoints to ensure that 96% of the selections made are classified as hits within the target area, means that the variability of target endpoint selections is directly correlated with the *ID* calculation. Tasks that result in large variations of target endpoint placement are identified as having a lower *ID* than those that have very little variations in target endpoint locations. This observation is analogous to Fitts' original 1D targeting tasks where the target width is inversely proportional to the *ID*.

The other observation that is made from the results collated in Figures 6.11 and 6.12 is the change in variability of endpoint selection, as represented by the observed normalized error values for both input modalities. As previously discussed, a lower *ID* corresponds to greater variability of accuracy, while decreased variability is associated with a higher *ID* value. This observation supports the hypothesis that an increase in ellipsoid eccentricity results in a stronger cue for the expected accuracy of the task (effectively reducing the target width/size, W) and therefore requiring increased movement time and skill to complete.

The core objective for the design and implementation of the ETTS system is to evaluate human performance using Fitts' Law in order to provide an objective measure for targeting-based tasks that can be extended to the simulation of clinically relevant procedures and skills. The analysis of performance for the trials of both input modalities makes use of Equation (2.4) to provide an objective measure of performance in the form of *IP*, measured in bits per second (*bps*). *IP* is defined as the inverse of the slope, *b*, where *a* and *b* of Equation (2.4) are empirical parameters determined through the use of linear regression. Figures 6.13 and 6.14 give the plotted results of the two trials using Equation (2.4).

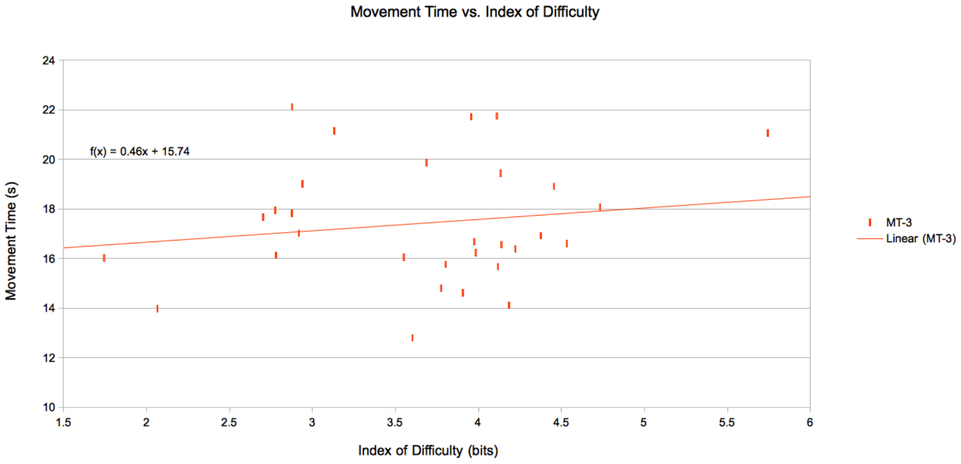


Figure 6.13: Plot showing the relation between *MT* and *ID* for the calculation of *IP* from the gamepad trials

The results chosen for the comparison of *IP* between the two input modalities is taken from the third, most experienced trial for each input device. Comparing the slopes of the two trend lines shown in the figures, the *IP* for each is calculated using the inverse slope of the response. This calculation of *IP* yields scores of 2.17*bps* and 12.5*bps* for the gamepad and LMD trials respectively. The resulting objective measures for *IP* suggests that since the trial completed

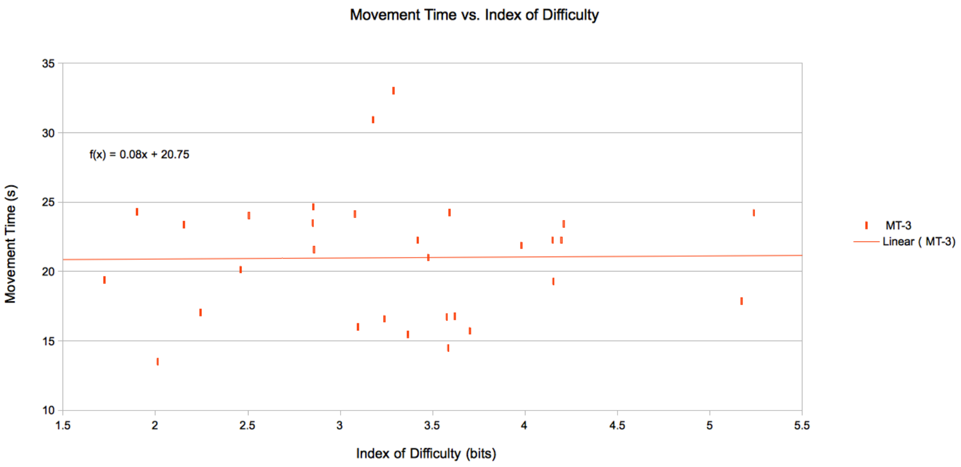


Figure 6.14: Plot showing the relation between *MT* and *ID* for the calculation of *IP* from the LMD trials

using the LMD has a higher throughput at 12.5*bps* as compared with the 2.17*bps* attained using the gamepad; the trial performed using the LMD is better than that of the gamepad controller.

Perhaps more important than the straightforward comparison of *IP* values is the power of Fitts' methodology to predict user performance across a wide assortment of other scenarios, independent of scale or size. A user's *IP* is an objective means for predicting *MT* for an arbitrary task having a known *ID*.

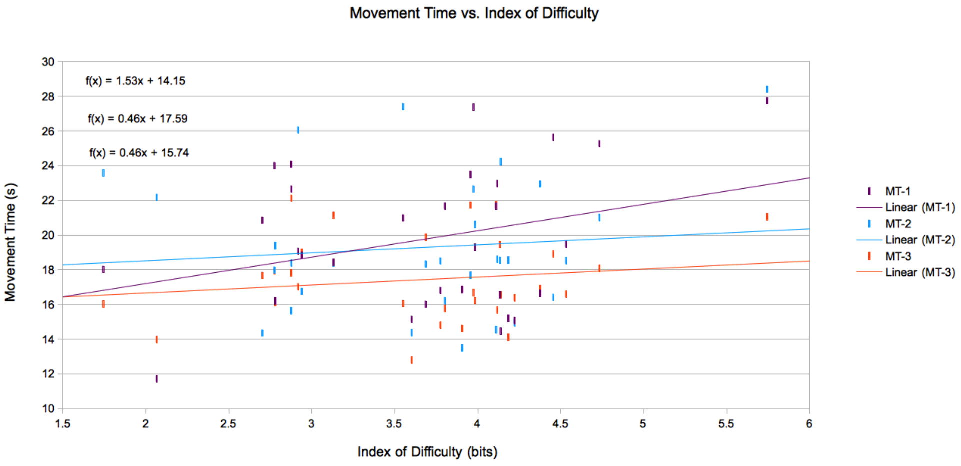


Figure 6.15: Plot showing the improved performance that results from running repetitive trials using the gamepad

The same techniques used for measuring human performance can also be applied to visualize and track improvements and changes in associated skills through the repetitive use of the ETTS system. An example of this phenomenon is observed in Figure 6.15 which shows a clear trend for the improvement in *IP* over the course of three successive trials. This observation is important as it provides validity for the argument that repetitive use of simulators such as the ETTS, will result in improved performance over time.

Chapter 7

Closing Remarks

7.1 Discussion and Conclusion

In conclusion of the research conducted, we begin by reviewing the research objectives of this thesis and identify them as being:

- Analyzing the feasibility of low(er) cost surgical simulators developed using the tools and techniques established by the video game industry.
- Investigate the efficacy of such simulators as a teaching and training add to traditional methods.
- Obtain the means to objectively evaluate user performance in targeting-based tasks.
- Evaluate the impact of HID's in simulator design and performance.

The key objective with this thesis was to develop the tool(s) necessary to answer the important research questions outlined above while still trying to work towards the end goal, which is the development of low(er) cost, widely available surgical simulator with a specific focus on applications in neuro-endoscopic procedures.

Early research quickly demonstrated that through the use of commercially available and well established game engines such as EE, many of the perceived requirements of an endoscopic surgical simulator could be resolved. While the results of this early work reinforced the hypothesis of utilizing existing engines for simulator design and implementation, it also raised key questions that needed to be addressed before further efforts are warranted in the implementation of such a VR simulator.

This realization led to the development of the ETTS system with the intent of addressing some of the immediate shortcomings and obstacles noted in the EE. In particular working with a stripped down, highly focussed platform that catered exactly to the research questions put forth was needed. This allowed complete control and a heightened understanding of the inner workings of the simulator and underlying render engine. The high degree of freedom afforded in the design of the ETTS platform allows for the unrestricted development in the future of new modules as new research questions are put forth.

Through the development of the ETTS system an initial idea of the efficacy of surgical simulators as teaching and training aids can be observed. Additional opportunities have been revealed through the research and development process that further enhance the user's experience (such as the playback and analysis module). Further testing and studies involving users ranging from novices to experts is needed before any such indications can be made definitively however initial results of the limited trials that have been conducted are promising.

A key milestone with the research surrounding the design and implementation of the ETTS system was in the application of Fitts' Law for the objective measure of user performance related to the ID of targeting tasks. This allows us to not only distinguish novices from experts but also provides a means for tracking user performance over time and identify any improvement as a result of using the surgical simulator.

While existing research has already demonstrated a lack of justification for the implementation of expensive haptic interfaces through findings that fail to show any apparent advantages or effects in improving user performance, we were interested in researching the role of readily available and affordable forms of HID's. In particular we compared the results obtained from trials completed using a traditional gamepad style of input to that of a newer, more natural form of input in the Leap Motion. Early results suggest a positive impact in the use of the LMD in targeting tasks, providing a more fluid form of input that is akin to the type of upper limb movement used by clinicians.

7.2 Future Work

Throughout the duration of this thesis a number of interesting research questions have been raised through the observation of initial results and testing of the ETTS system. Of particular interest is the investigation of the role different shaders can play in determining user performance as it relates to targeting-based tasks. Particularly the effects of various lighting cues that aid in depth perception.

The development of the *Playback and Analysis* module inspired a number of new features, some of which have yet to be implemented. Two additional modes would ideally be added in future work, namely a dynamically created heat-map texture and the support for ghosting trial data.

The idea behind the heat-map texture being that targeting approach data would be mapped onto a two dimensional UV texture that could than be visualized in the simulator identifying the most common approaches and indexed against a user's clinical skill ranking (Expert vs Novice). The hypothesis being that such a tool could be used to understand the difference in approach patterns used by novices and experts with the goal of using the data to further enhance training methods.

The inspiration for ghosted trial data comes from similar applications in the video game industry (particularly in racing games) and works to provide a semi transparent overlay of a previous trial's attempt. The result is that while performing a trial session, novices could be guided in realtime by the recorded approach of an expert.

Lastly, and perhaps most ambitious of the desired future work would be the implementation of a pupil tracker that compliments the existing means for capturing trial data. Research into the patterns that allow experienced clinicians to perform at a higher level than novices in tasks such as the identification of the target's longest axis could be used to further improve existing teaching methods. Such an example would be observing the difference in how experts and novices identify the target's longest axis and relative position in the skull using only preoperative images (MRI, CT, X-RAY).

Bibliography

[1] Neurotouch etv. Available at: http://neurosim.mcgill.ca/index.php?page=Simulators.

[2] P. B. Andreatta, D. T. Woodrum, J. D. Birkmeyer, R. K. Yellamanchilli, G. M. Doherty, P. G. Gauger, and R. M. Minter. Laparoscopic skills are improved with lapmentortm training: Results of a randomized, double-blinded study. *Annals of Surgery*, 243(6):854–863, Jun 2006.

[3] P. Backlund and M. Hendrix. Educational games - are they worth the effort? a literature survey of the effectiveness of serious games. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2013 5th International Conference on*, pages 1–8, Sept 2013.

[4] National Research Council Canada. Neurotouch simulators description, 2011. Available at: http://www.neurotouch.ca/eng/simulators.html.

[5] B. Cowan and B. Kapralos. A survey of frameworks and game engines for serious game development. In *Advanced Learning Technologies (ICALT), 2014 IEEE 14th International Conference on*, pages 662–664, July 2014.

[6] Digital Extremes. Evolution engine description and overview, 2014. http://www.digitalextremes.com/about/.

[7] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology*, 47(6):381–391, Jun 1954.

[8] N. Gupta, J. Park, C. Solomon, D. A. Kranz, M. Wrensch, and Y. W. Wu. Long-term outcomes in patients with treated childhood hydrocephalus. *Journal of Neurosurgery: Pediatrics*, 106(5):334–339, May 2007.

[9] R. Guzman, A. V. Pendharkar, M. Zerah, and C. Sainte-Rose. Use of the neuroballoon catheter for endoscopic third ventriculostomy. *Journal of Neurosurgery: Pediatrics*, 11(3):302–306, 2013. PMID: 23259463.

[10] F. A. Haji, A. Dubrowski, J. Drake, and S. de Ribaupierre. Needs assessment for simulation training in neuroendoscopy: a canadian national survey. *Journal of Neurosurgery*, 118(2):250–257, February 2013.

[11] S. Haque and S. Srinivasan. A meta-analysis of the training effectiveness of virtual reality surgical simulators. *Information Technology in Biomedicine, IEEE Transactions on*, 10(1):51–58, Jan 2006.

[12] H. M. Hasson. Simulation training in laparoscopy using a computerized physical reality simulator. *JSLS: Journal of the Society of Laparoendoscopic Surgeons*, 12(4):363–367, Dec 2008.

[13] Ergonomic requirements for office work with visual display terminals (vdts) - part 9: Requirements for non-keyboard input devices. Standard, International Organization for Standardization, August 2015.

[14] G. I. Jallo, K. F. Kothbauer, and I. R. Abbott. Endoscopic third ventriculostomy. *Neurosurgical Focus*, 19(6):1–4, December 2005.

[15] D. D. Limbrick Jr., L. C. Baird, P. Klimo Jr., J. Riva-Cambrin, and A. M. Flannery. Pediatric hydrocephalus: systematic literature review and evidence-based guidelines. part 4: Cerebrospinal fluid shunt or endoscopic third ventriculostomy for the treatment of hydrocephalus in children. *Journal of Neurosurgery: Pediatrics*, 14:30–34, November 2014.

[16] M. A. Kirkman, M. Ahmed, A. F. Albert, M. H. Wilson, D. Nandi, and N. Sevdalis. The use of simulation in neurosurgical education and training. *Journal of Neurosurgery*, 121(2):228–246, 2014. PMID: 24949680.

[17] I. S. MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Hum.-Comput. Interact.*, 7(1):91–139, March 1992.

[18] Microsoft. Xbox 360 controller map, original source, 2014. Xbox 360 Controller Map, Original Source Available at: `http://www.crytek.co.kr/confluence/download/attachments/1146915/cryengine_controls_xb360_01.jpg`.

[19] M. Monteiro, A. Corredoura, M. Candeias, P. Morais, and J. Diniz. Central hospital - master of resuscitation: An immersive learning approach. In *Serious Games and Applications for Health (SeGAH), 2011 IEEE 1st International Conference on*, pages 1–4, Nov 2011.

[20] S. K. Moore. The virtual surgeon [virtual reality trainer]. *Spectrum, IEEE*, 37(7):26–31, Jul 2000.

[21] Leap Motion. Leap motion api documentation, 2014. Leap Motion Controller Figure Available at: `https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Overview.html`.

[22] Oxforddictionaries.com. simulator - definition of simulator in english from the oxford dictionary, 2015.

[23] C. Ribeiro, T. Antunes, M. Monteiro, and J. Pereira. Serious games in formal medical education: An experimental study. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2013 5th International Conference on*, pages 1–8, Sept 2013.

[24] H. W. S. Schroeder, W. R. Niendorf, and M. R. Gaab. Complications of endoscopic third ventriculostomy. *Journal of Neurosurgery*, 96(6):1032–1040, June 2002.

[25] S. L. Da Silva, Y. Jeelani, H. Dang, M. D. Krieger, and J. G. McComb. Risk factors for hydrocephalus and neurological deficit in children born with an encephalocele. *Journal of Neurosurgery: Pediatrics*, 15(4):392–398, April 2015.

[26] Simbionix. Lapmentor description and overview, 2014. Available at: http://simbionix.com/simulators/lap-mentor/.

[27] G. Szkely, C. Brechbhler, J. Dual, R. Enzler, J. Hug, R. Hutter, N. Ironmonger, M. Kauer, V. Meier, P. Niederer, A. Rhomberg, P. Schmid, G. Schweitzer, M. Thaler, V. Vuskovic, G. Trster, U. Haller, and M. Bajka. Virtual reality-based simulation of endoscopic surgery. *Presence*, 9(3):310–333, Jun 2000.

[28] R. J. Teather and W. Stuerzlinger. Pointing at 3d targets in a stereo head-tracked virtual environment. In *3D User Interfaces (3DUI), 2011 IEEE Symposium on*, pages 87–94, March 2011.

[29] J. R. Thompson, A. C. Leonard, C. R. Doarn, M. J. Roesch, and T. J. Broderick. Limited value of haptics in virtual reality laparoscopic cholecystectomy training. *Surgical Endoscopy*, 25(4):1107–1114, Apr 2011.

[30] Unity. Unity game engine description and overview, 2014. https://unity3d.com/.

[31] E. S. Veinott, B. Perleman, E. Polander, J. Leonard, G. Berry, R. Catrambone, E. Whitaker, B. Eby, S. Mayell, K. Teodorescu, T. Hammack, and L. Lemaster. Is more information better? examining the effects of visual and cognitive fidelity on learning in a serious video game. In *Games Media Entertainment (GEM), 2014 IEEE*, pages 1–6, Oct 2014.

[32] T. W. Vogel, B. Bahuleyan, S. Robinson, and A. R. Cohen. The role of endoscopic third ventriculostomy in the treatment of hydrocephalus. *Journal of Neurosurgery: Pediatrics*, 12(1):54–61, July 2013.

[33] J. Wiecha, R. Heyden, E. Sternthal, and M. Merialdi. Learning in a virtual world: Experience with using second life for medical education. *Journal of Medical Internet Research*, 12(1), Jan-Mar 2010.

[34] Q. Wu, J. Yang, and J. Wu. Evaluation of human pointing movement characteristics for improvement of human computer interface. In *Information and Automation (ICIA), 2010 IEEE International Conference on*, pages 799–804, June 2010.

[35] S. Zhai, J. Accot, and R. Woltjer. Human action laws in electronic virtual worlds: An empirical study of path steering performance in vr. *Presence*, 13(2):113–127, April 2004.

Appendix A

Additional Listings

Listing A.1: Serialized Scene File in JSON

```
{
"constraints":{
  "C-0":{
    "gaze":{"+X":25,"-Y":-6,"~Z":false,"~X":false,"+Y":63,"-Z":-13,"~Y":false,"-X":-18,"+Z":29},
    "position":{"+X":155,"-Y":-22,"~Z":false,"~X":false,"+Y":125,"-Z":-153,"~Y":false,"-X":-68,"+Z":142},
    "rotation":{"+X":3,"-Y":-3,"~Z":false,"~X":false,"+Y":4,"-Z":0,"~Y":false,"-X":-3,"+Z":0}
    },
  "Material-7":{
    "Diffuse":{"-Green":0,"-Alpha":1,"+Red":1,"+Green":1,"+Blue":1,"-Red":0,"-Blue":0},
    "Specular":{"+S":1,"-S":0},
    "Ambient":{"-Green":0,"-Alpha":1,"+Red":1,"+Green":1,"+Blue":1,"-Red":0,"-Blue":0},
    "DiffuseMap":true
    },
  },
"sceneData":{
  "Material-7":{
    "Diffuse":{"Green":1,"Blue":1,"Red":1,"Alpha":1},
    "Specular":0,
    "Ambient":{"Green":0.2,"Blue":0.2,"Red":0.2,"Alpha":1},
    "DiffuseMap":0,
    "lowLevel":{
      "diffuseMap":16,
      "diffuseMapSrc":"~/Uv-grid-subgrid.png",
      "shaderProgram":0,
      "specularMapSrc":"",
      "normalMapSrc":"",
      "specularMap":0,
      "normalMap":0,
      "PATH":"materialSrc",
      "cubeMap":15,
      "name":"name"}
      },
  "C-0":{
    "zoom":1,
    "globalCamera":true,
    "gaze":{"X":0,"Y":50,"Z":9},
    "up":{"X":0,"Y":1,"Z":0},
    "position":{"X":66,"Y":66,"Z":70},
    "rotation":{"X":0,"Y":0,"Z":0}
    }
  }
}
```

Listing A.2: Playback slider action function used to update seek position

```
- (IBAction)playbackSliderAction:(id)sender {
    if ([sender isKindOfClass:[NSSlider class]] || [sender isKindOfClass:[OpenGLView class]]) {
        playBackCurrentFrame = [[self playBackSceneFrames] objectAtIndex:[self playBackCurrentFrameIndex]];
        [self setPlayBackCurrentTimeIndex:[NSString stringWithFormat:@”%.02fs”,[[playBackTimeIndices objectAtIndex:[self playBackCurrentFrameIndex]] doubleValue]]];
        [self updateGraphStats];
        NSArray *keys = [playBackCurrentFrame allKeys];
        int objects = 0;
        for (int i=0; i < [sceneObjects count]; i++) {
            if ([[sceneObjects objectAtIndex:i] isKindOfClass:[Entity class]]) {
                if (i < [keys count]){
                    NSString *key = [[(Entity*)[sceneObjects objectAtIndex:i] attributes] objectForKey:@”uid”];
                    if ([keys indexOfObject:key] < [keys count]){
                        NSDictionary *objectAttributes = [playBackCurrentFrame valueForKey:key];
                        Entity *currentObject = (Entity*)[sceneObjects objectAtIndex:i];
                        for (int j=0; j < [[objectAttributes allKeys] count]; j++)
                        {
                            [[currentObject attributes] setObject:[[objectAttributes objectForKey:[[objectAttributes allKeys] objectAtIndex:j]] mutableCopy] forKey:[[objectAttributes
```

```
                                allKeys] objectAtIndex:j]];
                        }
                        [(Entity*)[sceneObjects objectAtIndex:i]
                            updateModelMatrix];
                        objects++;
                    }
                }
            } else if ([[sceneObjects objectAtIndex:i] isKindOfClass:[
                CameraObject class]]) {
                if (i < [keys count]) {
                    if ([keys indexOfObject:@”C–0”] < [keys count]) {
                        NSDictionary *objectAttributes = [
                            playBackCurrentFrame valueForKey:@”C–0”];
                        CameraObject *currentObject = (CameraObject*)[
                            sceneObjects objectAtIndex:i];
                        [currentObject importAttributes:objectAttributes
                            ];
                        [currentObject reloadAttributes];
                    }
                }
            }
        }
    }
}
```

Listing A.3: Excerpt from Xbox 360 Controller Profile

```
"Sticks" : {
  "51" : {
    "Descriptor" : "RX",
    "LogicalMin" : -32768,
    "UsagePage" : 1,
    "LogicalMax" : 32767,
    "Usage" : 51
  },
  "49" : {
    "Descriptor" : "Y",
    "LogicalMin" : -32768,
    "UsagePage" : 1,
    "LogicalMax" : 32767,
    "Usage" : 49
  },
  "48" : {
    "Descriptor" : "X",
    "LogicalMin" : -32768,
    "UsagePage" : 1,
    "LogicalMax" : 32767,
    "Usage" : 48
  },
    "52" : {
      "Descriptor" : "RY",
      "LogicalMin" : -32768,
      "UsagePage" : 1,
      "LogicalMax" : 32767,
      "Usage" : 52
    }
  },
```

Curriculum Vitae

Name:	Shaun Carnegie
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2005 - 2013 B.E.Sc.
	University of Western Ontario London, ON 2007 - 2014 B.Sc
	University of Western Ontario London, ON 2013 - 2015 M.E.Sc.
Honours and Awards:	Dean's Honor List May 2013, May 2014
Related Work Experience:	Teaching Assistant The University of Western Ontario 2013 - 2015

Publications:

Carnegie,S, MacKenzie,J, Togeskov,A, Schmalz,M, de Ribaupierre,S, and Eagleson,Roy (2014) ”Endoscopic NeuroSurgery Simulation: Implementation on the Evolution Engine” The 6th IEEE Conference on Games, Entertainment, and Media. October 22-24, Toronto.

MacKenzie,J, Carnegie,S, Schmalz,J, Schmalz,M, de Ribaupierre,S, and Eagleson,Roy (2014) ”Surgical Simulation Workflow Representation using Hierarchical Task Analysis and Statecharts: Implementation on the Evolution Engine” The 6th IEEE Conference on Games, Entertainment, and Media. October 22-24, Toronto.