Remote-less Gaming with Anaglyph 3D Rendering using Kinect

Timothy Costigan
B.A.I Engineering
Final Year Project April 2012
Supervisor: Dr. Rozenn Dahyot

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Abstract

While keyboards and gamepads are the most common way of interacting with a computer, there may be certain tasks where such control methods may feel unnatural for example 3d modelling. In recent years however, motion controllers such as the Microsoft Kinect have become available which could offer a viable alternative.

The aim of this project is demonstrate the implementation of such a control system with a simple game. The game is controlled by tracking the player’s movements, gestures and hand poses with the Kinect and outputs a 3d image using anaglyph 3d rendering.

The final version of the application successfully fulfills the stated goal of the project however the speed and accuracy will need to be improved in future versions before it can be considered for serious use.
Acknowledgements

I would like to thank Dr. Rozenn Dahyot, my project supervisor, for all her help throughout the project. I would also like to thank Jonathan Ruttle for his technical advice.
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Signature

Timothy Casty

Date

2/4/2012
Chapter 1

Introduction

1.1 Introduction

The aim of this project is to create a simple game which can be played without the need for any sort of remote control. To that end, we developed an application which uses the Microsoft Kinect sensor for input and anaglyph 3d for depth. The Kinect is a relatively cheap depth camera, capable of skeletal tracking and is used for capturing the player’s movements. Anaglyph 3d is a technique of overlaying two slightly different images of the same scene to create a 3d image on a 2d display and is used to give the player a sensation of depth so they can more easily judge distances in the game (more detail in ‘Design’ chapter).

1.2 Motivation

While gamepads, keyboards and mice are ubiquitous methods for controlling computers and game consoles, in certain areas like 3d modelling they can be rather unintuitive as the user must press buttons to do something which would feel much more natural with their hands. Motion controllers like the Nintendo Wii-Mote have attempted to bridge this gap but still require the user to hold a physical device [1]. The Microsoft Kinect on the other hand does not require any physical controller and can track the player’s full body using a structured light approach (projected light pattern to determine depth) and has great potential to provide a new natural and intuitive way to interact with our computer devices [2].

1.3 Guide for Readers

This project report consists of a number of sections which cover various aspects of the project from concept to completion. The breakdown of the report is as follows:

Chapter 2 - ‘Review’ details the technology (software and hardware) and research which form the project,

Chapter 3 - ‘Design’ discusses the high level planning of the application,

Chapter 4 - ‘Implementation’ covers the translation of the design into code,
Chapter 5 - ‘Evaluation’ goes through how the performance, accuracy and correctness of the application were assessed and,

Chapter 6 - ‘Conclusion’ completes the report with an overview on what has been achieved.

The basic structure of the application can be seen in figure 1.1.

Figure 1.1: The high-level structure of the application.
Chapter 2

Review

This chapter covers in detail the various components and pieces of research which form the application.

2.1 Technology Used

![Diagram of application components](image)

Figure 2.1: The initial breakdown of features the application would require.

In the initial planning stages of the project, the application’s required components were decided and broken down as in figure 2.1. Using this breakdown, the libraries and tools listed in table 2.1 were chosen. These components will be discussed in greater detail in the ‘Software’ section.
<table>
<thead>
<tr>
<th>Library/Tool</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Programming language</td>
</tr>
<tr>
<td>Kinect sensor</td>
<td>Capturing player input</td>
</tr>
<tr>
<td>Official Kinect SDK beta</td>
<td>Controlling the Kinect and skeletal tracking</td>
</tr>
<tr>
<td>Coding4Fun Kinect Toolkit</td>
<td>Depth data to bitmap conversion</td>
</tr>
<tr>
<td>Kinect Toolbox</td>
<td>Gesture detection</td>
</tr>
<tr>
<td>EMGU</td>
<td>Hand pose recognition</td>
</tr>
<tr>
<td>Mogre</td>
<td>Game graphics</td>
</tr>
<tr>
<td>IZ3D</td>
<td>Anaglyph 3d rendering</td>
</tr>
<tr>
<td>BulletSharp</td>
<td>Physics</td>
</tr>
<tr>
<td>MogreFreeSL</td>
<td>Game sound</td>
</tr>
<tr>
<td>Glue Editor Lite</td>
<td>Map creation</td>
</tr>
<tr>
<td>DotSceneLoader</td>
<td>Loading map files</td>
</tr>
<tr>
<td>Autodesk 3D Studio Max 2012</td>
<td>Model creation</td>
</tr>
<tr>
<td>Ogre 3ds2mesh</td>
<td>Converting models to Mogre format</td>
</tr>
</tbody>
</table>

Table 2.1: The libraries and tools chosen.

2.1.1 Hardware

Kinect

The project application makes use of the Microsoft Kinect sensor (fig 2.2) for tracking the player’s skeleton and retrieving the raw depth image of the scene. The Kinect is able to track up to six people, 4 passively (no skeleton returned) and 2 actively although this project only makes use of player one’s skeleton [3]. It is capable of recording both colour and depth images but only depth is used by this project as it is unaffected by lighting conditions in the room [3]. Two different versions of the Kinect exist, a 2010 model which was originally designed for the Xbox 360 game console but later received a Windows SDK and the 2012 model designed exclusively for Windows. This project uses the 2010 variant and its Windows SDK as the 2012 version had not been released when work on the application began. The main difference between the two is the range of depth values they can detect, with the original Kinect having a range of 1.2 - 3.5 metres while the new Kinect can operate as close as 40 cm through its new ‘Near Mode’ [4].

Figure 2.2: The Microsoft Kinect Sensor.
2.1.2 Software

The following pieces of software were used by the project either in the final product or in earlier prototypes.

**Official Kinect SDK beta [5]**

The beta version of the official Kinect SDK was used by the project to control the Kinect sensor and is covered by a Non-commercial Use Only Licence [6]. The SDK provides a range of functions, allowing the program to control the motion of the Kinect’s tilt motor, to track up to six individuals with two being actively tracked (returning skeletal joint positions), to extract the raw depth image from the Kinect and to control the Kinect’s microphone array although the microphone feature is not utilised by this project. The official SDK was chosen over community implementations such as OpenKinect [7] because of its superior skeleton tracking, clearer documentation and less convoluted setup (multiple conflicting drivers exist for the community tools) [8]. While a complete version of the SDK was released towards the end of the project, it was designed exclusively for the 2012 version of the Kinect and was therefore not used.

**C# Programming Language**

The project was programmed using Visual Studio 2010 Professional in the C# programming language which is an object oriented and strongly typed language (variable types are explicitly specified) [9]. C# despite being a proprietary programming language is covered by Microsoft’s Community Promise allowing free use of the language for personal or commercial reasons free of charge [10]. C# was chosen instead of other languages such as C++ which were also supported by the SDK because it is a garbage-collected language, meaning it provides automatic memory management which frees the user from allocating and deallocating memory manually and it has good official documentation through MSDN [11]. C++ also lacks native threading support while C# has several different forms of threading such as worker threads which can be used to handle background tasks without forcing the rest of the application to wait [12].

**Coding4Fun Kinect Toolkit [13]**

The Coding4Fun Kinect Toolkit provides a range of extensions to the Kinect SDK which simplifies processes like the conversion of the Kinect’s depth data to a bitmaps and the saving of images. The library is hosted on Microsoft’s open source community website CodePlex and is covered by the Microsoft Public Licence which gives the developer (subject to certain conditions and limitations) the non-exclusive royalty free right to use, reproduce, modify, distribute or sell the toolkit.
Kinect Toolbox [14]

As the Kinect lacks any form of gesture detection by default, the Kinect Toolbox gesture library was used. This library provides functions for the detection of simple hand swipes, body poses and even more complicated gestures however this project only uses the swipe detection feature. The documentation for the library is unfortunately rather poor with there only being examples for some of its functions at the time of writing. The library, like the Coding4Fun Toolkit is hosted on Microsoft’s CodePlex website and is also covered by the same Microsoft Public Licence.

Mogre - Graphics Engine [15]

The graphical rendering for the final version of the game is handled by Mogre (Managed Ogre) (fig 2.3). Mogre is a C# wrapper for the C++ graphics engine Ogre and it was chosen over alternatives like OpenTk (discussed in the next section) as it has many useful features such as a model loader and simplified lighting which OpenTK lacks. Entities such as models, cameras and lights are stored in a tree structure in Mogre, meaning they are all children of some parent node and can have child nodes of their own. The library is covered by the GNU Library or Lesser General Public Licence which allows its use for free in open source or proprietary programs.

OpenTK - Graphics Library [16]

In some early versions of the application, OpenTK, a C# wrapper for OpenGL, OpenCL and OpenAL was used for rendering the graphical portion of the project (fig 2.4). The library is covered by the MIT/X11 licence which gives the developer the right
to use, modify or distribute the library for personal or commercial purposes without limitation. The use of OpenTK in the project was ultimately abandoned for Mogre as it lacked several key features such as a model loader.

Figure 2.4: The first prototype of the project application which used OpenTK. The image on the left is the depth image of the player and the one of the right is the same image in the form of a 3d point cloud.

**EMGU - Computer Vision Library [17]**

The project uses the EMGU library which is a C# wrapper of Intel’s OpenCV computer vision library, to manipulate the depth image (fig 2.5). The library provides a large range of computer vision techniques from simple smoothing algorithms to more complex feature detection systems like SIFT although this project only uses its smoothing, edge detection, contour detection and convex hull methods. EMGU is under a dual licence system which consists of an open source licence and a commercial licence, requiring the purchase of a commercial developers licence if the developer wishes to keep their project closed source.

Figure 2.5: EMGU being used to find the edges and convex hulls of the hands in the left image, with the result in the right image.

**BulletSharp - Physics Engine [18]**

The project uses the BulletSharp library for its physics simulation. BulletSharp is a C# wrapper for the C++ physics engine, BulletPhysics. The library is used to calculate when objects in the game are colliding with each other and how they should respond. BulletSharp was chosen as it was designed with Mogre in mind and could be easily
integrated into the project. The library is covered by the MIT licence (like OpenTK) and is open source.

IZ3D - Anaglyph 3D Drivers [19]

The project uses the IZ3D drivers to give the perception of depth through anaglyph 3d rendering (fig 2.6). IZ3D works with any program that uses DirectX (a group of libraries provided by Microsoft for multimedia applications [20]) for rendering, which Mogre does. The drivers are proprietary and are available in a free package containing just anaglyph drivers and a premium package which costs $39.99 and supports more 3d devices such as head mounted displays and lcd shutter glasses.

Figure 2.6: An example of the game with anaglyph 3d enabled, red/blue glasses are required to see the depth.

MogreFreeSL - Sound Library [21]

The sound for the game is handled by the library MogreFreeSL. This library is a Mogre wrapper for the FreeSL sound library which is in turn a wrapper for OpenAL (a 3D audio API) owned by Creative and provides realistic environmental 3d sound. FreeSL was created by Lukas Heise and can used and modified freely.

Glue Editor Lite [22]

Glue Editor Lite is used to create the scenes for the game (fig 2.7). Maps created using this scene editor are exported to the game in the .scene format which is an XML file specifying the entities, their positions, scales and orientations within the scene. This editor was chosen over others because of its simplicity, as scenes are easily created by dragging and dropping models into the 3d space. The editor however is unfinished as it lacks any terrain manipulation and initially had a severe bug which prevented scenes from
being correctly loaded into the game. The issue was that it incorrectly named different instances of the same model with the model rather than instance name, meaning duplicate names (which are illegal in Mogre) were being used, luckily the program is open source and a bug fix could be written (further details in the ‘Implementation’ chapter).

Figure 2.7: The Glue Editor Lite being used to create one of the game’s scenes.

**DotSceneLoader [23]**

The DotSceneLoader library is used to load the .scene files created by Glue Editor Lite, as Mogre has no scene loading methods. It works by reading in the map file and recreating the models, cameras and lights specified within, in the game world. The library is in the public domain allowing it to be used freely without restriction.

**Autodesk 3D Studio Max 2012 [24]**

The custom models used in the application were creating using the 3d model editor, Autodesk 3D Studio Max (fig 2.8). 3D Studio Max is a proprietary program, with full versions retailing for $3,495 although a free educational version was used for this project. The program was chosen for its clean and consistent interface although it lacks an ability to directly export to Mogre’s .mesh model format so an additional tool ‘Ogre 3ds2mesh’ (discussed in the next section) was used.

**Ogre 3ds2mesh [25]**

In order to convert the .3ds models exported by Autodesk 3D Studio Max 2012 (discussed above) the utility 3ds2Mesh had to be used. 3ds2Mesh is an exporter written by David Geldreich which converts .3ds files to Mogre’s .mesh format. The tool is closed source and no licence is associated with it so the limitations on its use are uncertain.
Nunit - Unit testing tool [26]

As discussed in the ‘Evaluation’ chapter, parts of the program were tested through a process known as ‘unit testing’ where the program is broken down into individual units which are tested in isolation. In order to perform these tests, a unit testing framework like Nunit (fig 2.9) is required which consists of a unit testing library that allows specific testing methods to be written that can be called by an external Nunit testing tool. This tool gives more detailed information on whether or not a test failed or passed and allows the tests to be quickly and easily written and run [27].

2.2 Similar Projects

This section gives an overview of the various papers which were consulted in the course of researching the project. Particular attention was paid to papers which used the Kinect
Global Hand Pose Estimation by Multiple Camera Ellipse Tracking [28]

This paper details a joint project by the University of Nevada and the NASA Ames Research Center, which attempts to track the position and orientation of the user’s hands. This was one of the first papers researched for the project and as such bears little similarity to the final project application. A multi-camera system is used instead of the Kinect (which uses structured light), the user must wear gloves with special ellipse markers to track the orientation of the hands (the project application requires no special clothing although no orientation tracking is implemented) and it does not recognise the player’s hand pose. Nothing was taken from this approach as it provides no hand pose recognition and the Kinect already can track the position of the player’s hands.

Efficient Model-Based 3D Tracking of Hand Articulations using Kinect [29]

This paper from the University of Crete in Greece uses the Kinect sensor to accurately model not just simple poses but rather the actual articulations of the hand. According to the paper, it works through a process known as Particle Swarm Optimisation which is:

“a stochastic, evolutionary algorithm that optimizes an objective function through the evolution of atoms of a population. A population is essentially a set of particles that lie in the parameter space of the objective function to be optimized.” [29]

This approach was considered for the project application but was not used as the method requires computer hardware that was far more powerful than what was available for this project. Even with the required hardware, it could only operate at a frame rate of 15Hz while tracking just one hand, while the project requires full skeletal tracking and recognition of both hands.

Dynamic Hand Pose Recognition using Depth Data [30]

This paper discusses a technique for recognising hand poses developed by Pennsylvania State University and HP Labs Bangalore India, which uses the depth data of the image to create 2d and 3d feature descriptors which are used to identify the pose. An implementation of this method was attempted early on in the project but the low resolution and noisy nature of the Kinect proved difficult and moving closer to the Kinect to take advantage of the increased hand resolution produced a ghosting effect which can be seen in figure 2.10. However, certain aspects of this design were adopted by the project,
Figure 2.10: ‘Ghosting’ effect can be seen as the player gets too close to the Kinect, leaving the operational range.

such as recognising only a fixed number of poses instead of trying to accurately model the hands and the use of the depth image for its illumination invariance (meaning the lighting level of the room would not be important).

**Recognising Hand Gestures with Microsoft Kinect [31]**

This paper from Stanford University uses the Kinect sensor to recognise grasping or dropping gestures, in addition to using full body tracking and a simple game to demonstrate its use. This paper inspired much of the design for the project application although its method of hand recognition differs. The paper’s system uses both the depth and colour data from the Kinect to recognise a sequence of images while the project application uses just the depth data to recognise a single image.

**Fusion4D - Natural and Immersive Interface for Manipulating 3D Objects [32]**

Fusion4D is a student developed application from the University of Sao Paulo, Portugal and shares many similarities with the project application. Fusion4D like the project application, uses the Microsoft Kinect to allow the user to interact with an object in a virtual environment, although it does not feature any hand pose detection and instead uses voice recognition. The main contribution of Fusion4D to the project application, is the idea of using anaglyph 3d rendering to give the player the perception of depth.

**Robot-Manipulator Teleoperation by Markerless Vision-based Hand Arm Tracking [33]**

This paper details a method using two cameras to track the position of the user’s arm and hand in 3d space to control a robotic arm. This paper was the basis for the project application’s hand pose recognition system, using the depth data to produce an edge image of the hand which could be easily processed by a computer vision library.
Chapter 3

Design

This chapter outlines the planned features of the project and discusses how they might be implemented.

3.1 Demonstration Application

Based off the initial plan for the application and the research performed in the ‘Review’ chapter, the project application had the following planned features:

- Mapping of the player’s hands to a virtual set, allowing them to interact with the game’s environment.
- Mapping of the player’s head to the camera so they can peer round objects.
- Simple recognition of whether the hands are open or closed, to allow for more complex control.
- Simple gesture detection for starting and resetting the game.
- Anaglyph 3d rendering to add a sense of depth to the game, increasing the immersion by allowing the player to judge how far they are reaching into the scene.
- Physics simulation to calculate object collisions and their response to those collisions.
- Different scenes or maps to choose from.

In the sections below, we will discuss how these features will work and where appropriate the theory behind them.

3.2 Skeletal Tracking

According to Shotton et al [34] which is the basis for the Kinect’s tracking algorithm, the Kinect sensor tracks the human body using an interesting machine learning approach. The tracking works by associating each pixel of the player in the depth image with a particular segment of the body through the use of a large training set consisting of over 500,000 frames composed of diverse body motions. This approach results in a highly robust and efficient tracking system which is capable of tracking 20 skeletal joints (head, hands, elbows, etc) in real time with many different body types. The skeleton returned
by this method, is used by the project application to map the player’s hands and head to the game.

3.2.1 Hand Tracking

The game’s hand tracking uses the Kinect’s skeleton tracking capability to retrieve the player’s skeleton like in figure 3.1. The position of the player’s hands can be found by simply querying the skeleton for the coordinates of the left and right hands, however the locations returned are initially relative to the edges of the Kinect’s field of view (fig 3.2) which means that the player would be required to stand in the same spot if they wanted to keep the mapping of the hands consistent. The ideal situation would be to make the coordinates of the hands relative to the body instead, so the player could stand wherever they want in the scene. To do this, we find a part of the player’s body which remains more or less stationary when they are moving their hands and use it as the origin for the coordinate system such as the centre of the hips (fig 3.3). We do this by getting the

![Figure 3.1: An example of the skeleton as tracked by the Kinect.](image)

![Figure 3.2: The skeletal coordinates are not initially relative to the body.](image)
Figure 3.3: The hand coordinate system, if the centre hips are taken as the origin.

The position of that point (the hips) and subtracting it from that of the hands (fig 3.4), for example if the centre of the hips is at location:

\[ X_1 = 160, Y_1 = 120 \]

and the hand we are determining the new position of is at:

\[ X_2 = 100, Y_2 = 40 \]

The coordinates of the hand relative to the new origin (centre hips) are calculated like so:

\[ X_3 = X_2 - X_1 = 100 - 160 = -60 \]
\[ Y_3 = Y_2 - Y_1 = 40 - 120 = -80 \]

The player’s hands can then be easily mapped to the screen by associating an arbitrary range, say -80 to +80 in the x direction and -60 to +60 in the y direction to the edges.

Figure 3.4: The coordinates of the player’s hands relative to the new origin.
of the screen, if a hand is at position (-80,-60) it is in the top left hand corner of the screen and if it is at (+80,+60) it is in the bottom right hand corner. There is however a flaw with using a fixed range as the resolution of the hands is affected by their depth, as the hands get closer to the Kinect, the same movement covers a greater distance in the image.

The solution is to scale the range as the hands get closer or further away from the Kinect. We use the depth value to find this new range although, as we want the range to decrease as the hands go further away and increase as they get closer, we measure the depth from the furthest possible point from the Kinect, rather than the Kinect itself. The Kinect can only read a maximum depth of around 4000mm so this will be the cutoff:

\[ \text{Cutoff} = 4000 \]

so the depth can be calculated like so:

\[ \text{Depth} = \text{Cutoff} - \text{Zhand} \]

This value can then be divided by some constant to produce the x range for the current depth of the hand and the y range can be determined by dividing that result by the aspect ratio:

\[ \text{Xrange} = \text{Depth}/\text{Constant} \]
\[ \text{Yrange} = \text{Xrange}/\text{AspectRatio} \]

The constant can be found by choosing a specific depth and range for that depth, say 2000mm and 240pixels and then dividing that depth by that range:

\[ \text{Constant} = (\text{Cutoff} - \text{OptimumDepth})/\text{OptimumRange} \]

which would be:

\[ (4000 - 2000)/240 = 8.3 \]

This constant can then be applied to any depth value to find the appropriate x and y ranges. For example, at 1.8 metres with an aspect ratio of 1.3, the calculation would be:

\[ \text{Zhand} = 1800 \]
\[ \text{Depth} = 4000 - 1800 = 2200 \]
\[ \text{Xrange} = 2200/8.3 = 265 \]
\[ \text{Yrange} = 265/1.3 = 203 \]
so the player’s hand would be in the top left corner of the screen at (-132.5, -101.5) and bottom right at (132.5, 101.5). This calculation is performed on both hands so that a hand at 1.8 metres would be judged on a different range than one at 2 metres. One issue with this method is that the centre of the screen is at the centre of the hips, so some offset may need to be applied to translate the origin up to the chest as otherwise the player would need to reach towards their knees to get to the bottom of the screen.

3.2.2 Head Tracking

In an attempt to increase the immersion, the application maps the camera to the movement of the player’s head, allowing them to look round the side of objects by tilting to the side. As with the hands, the head’s position is initially relative to the edges of the Kinect’s field of view so the position of the head also needs to be determined relative to the body although only along the x axis in this case as vertical camera movement is ignored (fig 3.5).

\[ X_{\text{relative}} = X_{\text{head}} - X_{\text{hip}} \]

As with the hands, the range is also scaled to keep the amount the camera moves for a given amount of tilting, consistent:

\[ Depth = Cutoff - Z_{\text{hip}} \]

\[ X_{\text{range}} = Depth / \text{Constant} \]

\[ \text{Constant} = (Cutoff - \text{OptimumDepth}) / \text{OptimumRange} \]

so if at a depth of 1.8 metres the desired range is 200 pixels, the constant would be

\[ (4000 - 1800) / 200 = 11 \]

For example, at 1.8 metres, the range would be calculated:

\[ Z_{\text{hip}} = 1800 \]

\[ Depth = 4000 - 1800 = 2200 \]

\[ X_{\text{range}} = 2200 / 11 = 200 \]

so the camera will be at half the width of the screen to the left if the head is at (-100) and to the right if at (100).
Figure 3.5: The position of the head is determined relative to the centre of the body.

3.3 Hand Pose Recognition

As the Kinect does not provide any hand pose recognition, the project application had to implement its own system. While many techniques were discussed in the ‘Review’ chapter, no single system could be decided and many aspects from the various methods were assimilated into the final version although J.Kofman et al [33] probably had the most influence.

The final version uses the Kinect’s depth data to isolate the players hands as in figure 3.6, which it then converts into an edge image before applying several methods discussed in this section to identify the hands. As the depth data is calculated using a structured light pattern projected by the Kinect, an added benefit of using just the depth imagery is that the application is illumination invariant and can be used in darkness if needs be.

Figure 3.6: A portion of the same image: left unsegmented depth image with player highlighted green, right: hands isolated.

To isolate the player’s body from the background of the depth image, the project application uses the player index feature of the Kinect, where the 3 least significant bits of every pixel associated with a player contain their player id. The background can be easily removed from the image by simply checking those last 3 bits of every pixel and removing those which do not equal the player id.
The next step is to isolate the player’s hands. Which we do by first, finding the depth value of the hand furthest from the Kinect sensor:

\[ \text{MaxDepth} = Z_{\text{hand}} \]

and filtering the image for all depth values greater than this value, leaving just the hands. If the hands are too close to the player’s body, the body could be included as well so the game should be designed in such a way to discourage the player from pulling their hands that far back.

Once the hands have been isolated, the image is thresholded by the mean to produce a binary image as the depth values are no longer necessary. The image is then smoothed to reduce noise, using gaussian smoothing as it is gentler and better at preserving edges than similar smoothing filters [35]. The edges of the image are then found using an edge detector such as the Canny Edge detector which was developed by John Canny in 1986 [36]. The Canny edge detector is used instead of others like the Robert’s Cross Operator or Sobel as Canny has a “low error rate”, “well localised edge points” and “only one response to a single edge” and in comparison to Sobel, Prewitt and Robert’s edge detectors, it produces better results [37].

The contours, which are the continuous edges of each hand are then found, as the EMGU (computer vision library) contour object contains many methods which can be used to identify the hand. The contour can provide useful information such as the hand’s convex hull which is the smallest convex set (two points can be joined by a straight line without leaving the set) [38] of the hand’s points [39], its convexity defects (points which are not on the edge of the convex hull) and bounding rectangle, which can all be used to help classify the state of the hand. Several contours may be returned for each hand with some corresponding to noise and others to small sections of the hand, so only the biggest one (which should correspond to most of the hand) is used.

Using the hand’s contour, three recognition methods were developed, each an attempt to solve some inherent problem with the previous one. The three methods are:

**Using Convexity Defects**

Inspired by papers like Manresa et al [40], this method works by looking for spread fingers by comparing the hand contour to the convex hull which encloses it. The convexity defects (points which are not on the edge of the convex hull) are measured and if they are large enough, we can assume we have found the gap between the fingers. If enough defects above some threshold (which was found through experimentation) are found, the hand can be classified as open, otherwise it is considered
closed. This was the earliest method tried and unfortunately only works for hands where the fingers are spread.

**Using Bounding Rectangles**

As the method above is only capable of recognising open hands with spread fingers, a new technique was developed to recognise hands when the fingers are close together, using the bounding rectangle as seen in figure 3.8. This method works by comparing the dimensions of the rectangle, based on the observation that an open hand will be taller than it is wide, while a closed hand’s width and height will be similar. This solution works but unfortunately has trouble with spread fingers, as the width and height of a such a hand are quite close.

**Using Convexity Defects and Bounding Rectangles**

In order to handle open hands with both spread and closed fingers, the two methods above were combined. The technique works as follows:

1. Try the bounding rectangle method, if identified as open we return the result, otherwise move to step 2.

2. Apply the convexity defects method, in case the hand is actually spread producing an ambiguous result in step 1, if identified as open, return the result otherwise proceed to step 3.

3. If neither method could identify the hand as open, we label the hand as closed.

This technique unlike the two previous methods is capable of handling open hands with fingers both spread and close together although it can have trouble in some cases which will be discussed in further detail in the ‘Evaluation’ chapter.

![Figure 3.7: The edge image of the hands. The thick line is the convex hull.](image)

![Figure 3.8: The edge image of the hands with their bounding rectangles.](image)
3.4 Gestures

The project application begins in a paused state to give the player time to get into position, requiring a way of indicating to the program that we want the game to start. As the goal of the project is to operate without any form of remote control, this must be achieved entirely through body motions. This project’s solution is to use a gesture detection library to determine when the player has made some unique motion. As only two states are required (start/stop), the gestures recognised are relatively simple, for example the game is started by swiping two closed hands outward and reset by swiping outward with two open hands.

3.5 Physics

In order to allow the player to interact with the objects in the game, a physics engine is used to calculate the collisions between objects and their responses to those collisions. For every model in the game map that we wish to interact with, we create a physics object known as a rigid body which is added to the physics world. All the rigid bodies in the physics engine interact with each other as part of the physics simulation and their positions and orientations are used to update those of their associated model as in figure 3.9. The rigid body can be designated as either static (cannot move, ideal for walls) or dynamic (can move, ideal for a ball).

![Figure 3.9: Each model we wish to have a physical response has an associated rigid body in the physics engine.](image)

3.6 Anaglyph 3D

As the game uses a standard 2d display, the player may have great difficulty determining how far they are reaching in the z direction (towards the screen). The solution is to use an optical trick like anaglyph 3d to give the player the perception of depth. Anaglyph rendering is not a new technique, having been first developed by William Rollman in 1853.
This technique takes advantage of our binocular vision system, which allows us to accurately determine depth, up to 6-7 metres away, thanks to the separation between our eyes creating two images at slightly different angles which our brain can use to calculate the depth of the scene.

Anaglyph 3d works the same way, taking two pictures of the scene at slightly different angles (fig 3.11), colouring them with non-overlapping colours and then overlaying them together on the screen. Using cheap filters like in figure 3.10, only one of the two images is sent to either eye, creating a 3D effect. This effect was produced in the project using third-party drivers (IZ3D), though attempts were made to implement the feature using Mogre compositor scripts [43] which can be used to define post-processing effects but ultimately proved fruitless (more detail in ‘Evaluation’).

![Red/blue anaglyph filters which can be clipped on to the player’s glasses.](image)

![Two images from slightly different angles combined to form a 3d anaglyph image.](image)

### 3.7 Scene Design

In early prototypes of the program, the game world was hard coded which meant that its structure was specified at compile-time. This approach was not very flexible as it required the program to be rewritten to alter the scene in any way so instead it was decided to use an external map file. The .scene format was used which is a simple xml
file which specifies what entities are in the scene, alongside their position, orientation, scale and whether they are static or not (static entities cannot be moved by the physics engine). By using a map file, the game world could be created at run-time and be designed using many freely available scene editors, instead of tweaking lines of code. Here is an example of what the .scene file looks like:

```
<?xml version="1.0" encoding="utf-8"?>
<scene formatVersion="">
  <nodes>
    <node name="tudorhouse_017" id="1">
      <position x="250" y="220" z="-90" />  
      <rotation qx="0" qy="0" qz="0" qw="1" />  
      <scale x="0.5" y="0.5" z="0.5" />  
      <entity name="tudorhouse_017" meshFile="tudorhouse.mesh" static="true" />
    </node>
  </nodes>
</scene>
```

In terms of map design, the scenes needed are not particularly complex as the application’s gameplay is quite simple. The game scenes consist of an object the player can interact with, such as a ball or cube, the hand entities and some optional obstacles. The models associated with the hand entities are not actually used in-game, however their scale, position and bounding box are used to set the initial location, size and collision shapes of the hands which are rendered using custom hand models instead. Figure 3.12 shows what a scene may look like, it should also be noted that the scene lacks any terrain, this is a limitation of the Glue Editor Lite program. The terrain is unfortunately hard coded as a result, consisting of a textured plane, with an associated physics plane for collisions.
Figure 3.12: An example of a scene used in the project application in Glue Editor Lite. The cubes representing the ‘Hand Entities’ are replaced with custom hand models when loaded into the game.
Chapter 4

Implementation

This chapter shows how planned features of the project application, discussed in the ‘Design’ chapter were implemented in code.

4.1 Structure of the Project Application

The core of the project application is composed of 5 event listeners (fig 4.1):

- OnFrameRenderingQueued - Called when the graphics engine is waiting for the graphics card to finish rendering. As the graphics engine is doing nothing else, this is where the ‘physicsWorker’ thread is asked to advance the physics and the positions of the various models are updated.

- nui_SkeletonFrameReady - Called when the Kinect is returning a new skeleton frame. This is where the various parts of the program reliant on the positions of the player’s joints are updated.

- nui_DepthFrameReady - Called when the Kinect is returning a new depth image. Here is where the depth image is passed to the ‘depthWorker’ thread to identify hand poses.

- leftSwipeDetector_OnGestureDetected - Called when the Kinect Toolbox library detects a left hand swipe.

- rightSwipeDetector_OnGestureDetected - Called when the Kinect Toolbox library detects a right hand swipe.

Two worker threads are also used:

- depthWorker - Used to process the depth image handed to it by nui_DepthFrameReady and perform the hand recognition methods detailed in the ‘Design’ chapter. As processing the depth image can take longer than the Kinect’s refresh rate, any new images this thread receives while already processing one, are ignored.

- physicsWorker - Used to advance the game’s physics engine. The physics update runs relatively fast (more details in ‘Evaluation’) so a separate thread to handle it is not particularly necessary. This structure was decided early in the design process when it was not certain how expensive a physics update would be.
Figure 4.1: The core structure of the program.

The application on startup, goes through the following steps (based on the Mogre startup tutorial [44]):

1. Ask the user what scene and hand recognition they want.
2. Initialise the Kinect sensor.
3. Look for what resources need to be loaded from the Mogre Media folder.
4. Initialise Mogre based on the configuration settings (fig 4.2).
5. Load all the required resources.
6. Initialise the physics engine and sound engine.
7. Load the .scene file and add its entities to the physics engine.
8. Start listening to input from the Keyboard - in case the player wants to adjust the Kinect.
9. Initialise the event handlers for the Kinect, Mogre and gesture detection.
10. Enter the main game loop.

Once the above steps are complete the game has started and the user can begin to play.

4.2 User Input

In this section, we detail how the input from the Kinect is used to implement the control planned in the ‘Design’ chapter.
4.2.1 Skeleton Tracking

The skeleton returned by the Kinect is key to controlling the game, as it is used to map the player’s hands and head to the virtual world and to find the hands in the depth image for hand pose recognition. To enable skeletal tracking, the option must be specified on initialisation:

```csharp
Runtime nui = Runtime.Kinects[0];
/*UseDepthAndPlayerIndex is used by the hand pose detection to filter the depth image*/
nui.Initialize(RuntimeOptions.UseDepthAndPlayerIndex
               | RuntimeOptions.UseSkeletalTracking);
```

The performance of the tracking can be tweaked using smoothing parameters to adjust prediction, smoothing, correction etc. The values used by this project were chosen through experimentation, carefully adjusting them till the tracking appeared as close as possible to real life:

```csharp
nui.SkeletonEngine.TransformSmooth = true;
//Use to transform and reduce jitter
var parameters = new TransformSmoothParameters
{
    Smoothing = 0.5f,
    Correction = 0.3f,
}
Prediction = 0.4f,
JitterRadius = 1.0f,
MaxDeviationRadius = 0.5f
};
//apply the transform parameters to the skeleton
nui.SkeletonEngine.SmoothParameters = parameters;

The Kinect returns all the skeletons it is tracking in one frame (up to 6). We only need player one’s skeleton which can be retrieved from the skeleton data structure using a LINQ (Language-Integrated Query) statement [45]:

SkeletonFrame allSkeletons = e.SkeletonFrame;
//get the first tracked skeleton
SkeletonData skeleton = (from s in allSkeletons.Skeletons
    where s.TrackingState == SkeletonTrackingState.Tracked
    select s).FirstOrDefault();

The player’s joints can be easily retrieved from the skeleton:

Joint rightHand = skeleton.Joints[JointID.HandRight];

and their location can be found using their position data member. In order to locate those joints in the depth image, the following method is called:

/*runtime is the Kinect object, position is the location of the hand returned by LINQ and width/height are the dimensions of the depth image*/
public static Vector DepthCoordinates(Runtime runtime, Vector position, int width, int height)
{
    float x, y;
    Vector result = new Vector();
    //returns x and y as a value between 0-1
    runtime.SkeletonEngine.SkeletonToDepthImage(position, out x, out y);
    result.X = x * (width-1);
    result.Y = y * (height-1);
    result.Z = position.Z;
    return result;
}

An example of its use would be:

//for a depth image of a resolution of 320x240
rightHand.Position = Toolkit.DepthCoordinates(nui,
    rightHand.Position,320,240);
The coordinates returned are used to locate the player’s hands in the depth image (used by hand pose recognition). However, in order to map the player’s movements to the game world, the coordinates must instead be relative to their body. We convert the coordinates by calling a method written for this project:

```
Toolkit.determineHandPosition();
```

which implements the technique discussed in the chapter ‘Design’ to make the position of the hands relative to the body.

We find the position of the hands relative to the virtual world by comparing their location to an arbitrary range with the player at its centre. This range is based on how far we think the player should stretch to reach the edges of the screen, if we wanted the range at 2 metres to be 75% the width of the screen at a resolution of 320, it would be 240 pixels. The pixel value of the range is not fixed, as the player takes up differing amounts of the screen depending on their depth. The range is instead found by finding the depth of the hand relative to some maximum value away from the Kinect (so that the range increases as we get closer to the Kinect) and applying a scaling constant to it:

```
//if the max value is taken to be 4000mm
//range = (4000-2000)/8.3 = 240.9
float range = depth/8.3f;
```

The scaling constant was found by choosing some arbitrary depth value and algebraically finding what constant would produce the desired range.

### 4.2.2 Gesture Detection

The player can start and reset the project application through some simple swipe gestures. The gestures are detected using the Kinect Toolbox library’s SwipeGestureDetector. Each detector object can only handle one joint, so two are created, one for each hand:

```
SwipeGestureDetector leftSwipeDetector = new SwipeGestureDetector(),
        rightSwipeDector = new SwipeGestureDetector();
//add the joints
leftSwipeDetector.Add(skeleton.Joints[JointID.HandLeft].Position, nui.SkeletonEngine);
rightSwipeDetector.Add(skeleton.Joints[JointID.HandRight].Position, nui.SkeletonEngine);
```
When a swipe is detected, an ‘OnGestureDetected’ event is thrown which is handled by
the two handlers ‘leftSwipeDetector_OnGestureDetected’ and ‘rightSwipeDetector_OnGestureDetected’.
These handlers simply switch a boolean value which is used by the game to decide whether
or not the game should be started or reset:

```csharp
protected static void rightSwipeDetector_OnGestureDetected(string obj)
{
    if (mRightHand.state == STATE.CLOSED)
        swipeRight = true;
    else if (mRightHand.state == STATE.OPEN)
        swipeRight = false;
}
```

### 4.2.3 Hand Pose Recognition

In order to detect what pose the player’s hands are in (used by the gesture detection
event handlers), one of the three methods discussed in the ‘Design’ chapter are used
(chosen on startup). Those methods are implemented through the following steps:

**Step One**

Using the player index feature of the Kinect, where the 3 least significant bits of the
depth value contain the player id, we filter the depth image to remove everything
but the player (fig 4.3).

```csharp
int player = (int)pixel & 7;
```

Any pixel that does not return 1 (for player one) can be filtered out.

Figure 4.3: The raw depth image with the player value encoded into each pixel.
Step Two
The player’s hands are then isolated by finding the depth value of the hand furthest from the Kinect and filtering out any higher values:

```c
//determine max depth threshold
int max_depth;
if (mRightHand.position.z > mLeftHand.position.z)
    /*HAND_ALLOWANCE is added to give a little flexibility as the depth value from the skeleton may not be accurate - value found through experimentation*/
    max_depth = (int)mRightHand.position.z + Toolkit.HAND_ALLOWANCE;
else
    max_depth = (int)mLeftHand.position.z + Toolkit.HAND_ALLOWANCE;
/*retrieve just the player’s hands from the image. Note - Toolkit is a class containing a collection of custom methods written for this project*/
short[] result = Toolkit.processDepthImageWithPlayer(image,max_depth);
```

This should hopefully leave something like figure 4.4.

![Figure 4.4: The player’s hands isolated in the depth image.](image)

Step Three
The depth image is smoothed to reduce noise as it was found to improve detection in early tests:

```c
Image<Gray, Byte> smoothImage = initialImage.SmoothGaussian(3, 3, 0, 0);
```

Step Four
As the depth values are no longer necessary, the image is thresholded by its mean value to produce a binary image which makes it easier to choose the thresholds for the edge detector in the next step:

```csharp
Image<Gray, Byte> binaryImage = smoothImage.ThresholdBinaryInv(mean, new Gray(255));
```

**Step Five**

The edges of the hands are found using the canny edge detector:

```csharp
Image<Gray, Byte> detectionImage = binaryImage.Canny(new Gray(255), new Gray(255));
```

Canny takes two threshold values one for edge points and another for points linking to edges, as we are using a binary image we can just set both thresholds to the value of the non-thresholded pixels in the last step.

**Step Six**

The contours of the hand are found:

```csharp
Contour<Point> contours = detectionImage.FindContours();
```

which we iterate through to find the biggest one which should correspond to the outer edge of the hand. It is at this point that one of the three methods discussed in the ‘Design’ chapter is employed.

**Using the Convexity Defects**

The convexity defects are found:

```csharp
//note it is actually mispelt in the EMGU library as Defacts
var defects = biggestContour.GetConvexityDefacts(null,
    ORIENTATION.CV_CLOCKWISE).ToArray();
```

which are then measured and if enough of them are beyond a certain distance from the convex hull, we can assume they are the gaps between the fingers of an open hand:

```csharp
//filter defects
int actualDefects = 0;
foreach (var defect in defects)
{
    //determine if above threshold - threshold found by experimentation*/
```
if (defect.Depth > DEFECT_THRESHOLD)
    actualDefects++;
}
//identify hand - OPEN_DEFECTS found by experimentation
if (actualDefects >= OPEN_DEFECTS)
{
    pose = STATE.OPEN;
}
else
{
    pose = STATE.CLOSED;
}

Using Bounding Rectangles
The bounding rectangle of the convex hull is found:

var convexHull =
    biggestContour.GetConvexHull(ORIENTATION.CV_CLOCKWISE);
Rectangle rect = convexHull.BoundingRectangle;

and the width and height are compared to identify the pose:

//just in case the hand is sideways - the ratio is checked both ways
float ratio1 = (float)rect.Width / (float)rect.Height;
float ratio2 = (float)rect.Height / (float)rect.Width;
//choose smallest
float ratio = (ratio1 < ratio2) ? ratio1 : ratio2;
//OPEN_RATIO found by experimentation
if (ratio < OPEN_RATIO)
{
    pose = STATE.OPEN;
}
else
{
    pose = STATE.CLOSED;
}

Using Both
The final method, instead uses both methods. It first tries to classify the hand using the bounding rectangle method, moving on to the convexity defects method if it fails.
Step Seven

In order to reduce the number of false pose recognitions, the running average of the results is taken and used to change the state of the hands instead of the direct output of the method. An average of the last 50 results is taken as this was found through user testing to produce a bearable lag in recognition. Readings are also not taken when the skeleton is moving too much, using the BaryCenterHelper feature of the Kinect Toolbox library:

```csharp
//add skeleton to the helper
BarycenterHelper baryHelper = new BarycenterHelper();
baryHelper.Add(skeleton.Position.ToVector3(), skeleton.TrackingID);
//if player is not moving too much
if {baryHelper.IsStable(skeleton.TrackingID)}
{
    //perform hand pose recognition
}
```

4.3 Virtual World

This section details how the game’s environment was created.

4.3.1 Creating the Scene

The player has a choice between two scenes on startup, a ‘space’ scene and a ‘ball’ scene:

- The ‘space’ scene (fig 4.5(a)) is a relatively simple one with a space theme. Gravity is set to zero and a space skybox is used. This was the first map created for the game and features no real gameplay as it was meant to showcase object manipulation.

- The ‘ball’ scene (fig 4.5(b)) is more complex and is composed of a cloudy sky and grass textured plane surrounded by buildings to form a courtyard where a ball is randomly thrown about for the player to interact with. The goal is to keep the ball moving.

Scene Editor

The game’s two scenes were made using the Glue Editor Lite scene editor. The scenes were created by dragging and dropping the various models into place within the editor and then exporting them to a .scene file for the game to load. As the editor is very simple,
it has no support for the manipulation of terrain, planes, skyboxes, cameras or lights so these features had to be specified in code.

In early tests, the game had trouble loading any scene files which contained more than one instance of a particular model despite each one having a unique name. This problem was eventually tracked down to a severe bug in Glue Editor Lite which gave each instance the name of its model rather than the name specified in the properties field (fig 4.6). This meant that two instances of the model ‘object.mesh’ which are named ‘object1’ and ‘object2’, would instead be referred to as ‘object.mesh’, so Mogre would only deal with the one it loaded first and ignore the rest. Thankfully, as Glue Editor Lite is open source, the solution was to alter one line in the editor’s source code from:

```csharp
this.writer.WriteAttributeString("name", instance.Parent.Name);
```

to:

```csharp
this.writer.WriteAttributeString("name", instance.Name);
```
Loading the Scene

The scene files are loaded using the DotSceneLoader library:

```csharp
Helper.DotSceneLoader mSceneLoader = new Helper.DotSceneLoader();
//parse the .scene file
mSceneLoader.ParseDotScene(filename,
    ResourceGroupManager.DEFAULT_RESOURCE_GROUP_NAME, mSceneMgr, node);
```

The ‘ParseDotScene’ method uses the name, model, orientation, position and scale parameters in the file to create entities which are then attached to the parent ‘node’ and displayed in the game. Certain features like skyboxes, planes or collisions planes are not specified in the scene file and must be hard coded:

```csharp
//skybox
mSceneMgr.SetSkyBox(true, "Examples/SpaceSkyBox");
//floor
Entity floor = mSceneMgr.CreateEntity("floor", "ground");
floor.SetMaterialName("Examples/GrassFloor");
.
.
physics.createPlane(new Mogre.Vector3(0, -ROOT_Y, 0) / physics.mScale,
    Mogre.Vector3.UNIT_Y, 0);
```

Adding Entities to the Physics Engine

In order for the player to interact with the objects in the scene, they are loaded into the physics engine through the following steps:

**Step One**

We iterate through all the static (do not move) and dynamic (move) meshes, which are conveniently kept by the DotSceneLoader:

```csharp
foreach (string item in mSceneLoader.DynamicObjects)
{
    //add item to physics engine in step 2
}
foreach (string item in mSceneLoader.StaticObjects)
{
    //add item to physics engine in step 2
}
```

**Step Two**
For each entity, we create a rigid body to represent it in the physics engine. Everything must be scaled down by some factor as in BulletSharp, 1 unit equals 1 metre, meaning unscaled the objects could be hundreds of metres in size:

```csharp
// use bounding box for dimensions as it matches quite well
CollisionShape shape = new BoxShape(
    entity.GetWorldBoundingBox(true).HalfSize / physics.mScale);
Matrix4 transform = new Matrix4();
transform.MakeTrans(entity.ParentNode.Position / physics.mScale);
/* the parameters are mass, friction, restitution (or bouncy-ness), start
   transform and collision shape. A mass of 0 means the rigid body is
   static, otherwise it is dynamic*/
physics.createRigidBody(1.0f, 0, 0.9f, transform, shape);
```

Whenever the physics engine is advanced by some time step, we use the orientations and positions of the rigid bodies to update their associated models:

```csharp
/* index for the collision array, the dynamic objects were added in the same
   order as they are in mSceneLoader.DynamicObjects*/
int i = 0;
foreach (string item in mSceneLoader.DynamicObjects)
{
    // get associated entity
    Entity entity = mSceneMgr.GetEntity(item);
    // retrieve its rigid body
    RigidBody body = (RigidBody)physics.mWorld.CollisionObjectArray[i];
    Matrix4 x = body.MotionState.WorldTransform;
    // set the entity’s position and orientation to that of the rigid body
    Mogre.Vector3 position = x.GetTrans();
    entity.ParentNode.Position = position*physics.mScale;
    Quaternion q = x.ExtractQuaternion();
    entity.ParentNode.Orientation = q;
    i++;
}
```

When updating the models, the values must be scaled up by the same amount they were scaled down.
Adding Sound

The MogreFreeSL sound library is used to play some background sounds and bounce
effects in the ‘ball’ scene. The background sound is ambient, so it plays at a constant
volume throughout the scene:

```cpp
AmbientSound backgroundSound = mSoundMgr.CreateAmbientSound("urban_loop.ogg",
   "urban_loop", true, false);
backgroundSound.Play();
```

The bounce effect is bound to the ball and follows it around the scene producing a 3d
sound effect which increases in volume as the ball gets closer to the camera and decreases
as it gets further away:

```cpp
SoundEntity objectSound = mSoundMgr.CreateSoundEntity("bounce.wav",
   ballEntity, "bounce", false, false);
```

4.3.2 Creating the Meshes

The project application, uses in addition to the meshes bundled with Mogre, the fol-
lowing custom models which were created in Autodesk 3d Studio Max 2012:

- A textured beach ball mesh for use in the ball scene (fig 4.7(a)).
- A closed left hand mesh for when the player’s left hand is closed (fig 4.7(b)).
- A closed right hand mesh for when the player’s right hand is closed (fig 4.7(c)).
- An open hand mesh for when either hand is open (fig 4.7(d)).

![Figure 4.7: The custom models created for the project application.](image)

These models were created through the following steps:

**Step One**

Primitive shapes such as cubes, spheres and cylinders are positioned, scaled, sculpted
and combined to form the shape of the model (fig 4.8).
Step Two

The primitives are grouped into a single editable mesh to which we apply uvw unwrapping. This unfolds the surface of the model, allowing a texture (such as skin) to be applied using the material editor (fig 4.9). This texture can be created in any image editing software such as Paint bundled with Windows.

Step Three

As 3d Studio Max cannot export to Mogre’s .mesh format, the textured model is instead saved as a .3ds file which can be converted to .mesh using the Ogre 3ds2mesh utility, which returns a .mesh file (fig 4.10) we can be used in Mogre directly.
4.3.3 Anaglyph 3d Rendering

The anaglyph 3d rendering was implemented using IZ3D which the player must enable before playing the game. No modification to the application code was required as IZ3D is a set of drivers which grab any DirectX game’s output and convert it into a 3d image. Attempts were made originally to write anaglyph code using compositor scripts (a Mogre feature) which unfortunately appear to be broken in the current version of Mogre (1.7.1).
Chapter 5

Evaluation

The chapter reviews how the program was tested and how its performance was assessed.

5.1 Testing

This section details the ways in which the project application was tested to ensure both its correctness and stability.

5.1.1 Unit Testing

Certain portions of the project were assessed through a method of testing known as ‘unit’ testing. According to R.Osherove [27]:

“A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A unit is a method or function.”

To test a piece of code or ‘unit’, we use a unit testing framework such as NUnit. Visual Studio does offer a unit testing facility but NUnit was used instead as it allowed tests to be run at any time from outside of Visual Studio through the NUnit commandline or graphical testing tool. To write a ‘unit’ test using NUnit, a testing class must be created (with the attribute [TestFixture]) containing the test methods we would like to run. These methods are given the [Test] attribute and contain whatever method calls and assertions we would like to make:

```csharp
using NUnit.Framework;
[TestFixture]
public class TestClass
{
    [Test]
    public void testCode()
    {
        //test against known results
        double[] result = Toolkit.getDimensionsOfPlane(50, 90, 1.5);
        /*is the result returned the same as the result calculated on paper*/
        Assert.AreEqual((int)Math.Ceiling(result[0]), 150);
    }
}  
```
The advantage of these ‘unit’ tests over other methods is that they are more readable and maintainable and can be written easily and run quickly [27]. A limitation of ‘unit’ tests is that they cannot easily be applied to the entire program as they must operate on code which can be run independently and in isolation. Portions of the program which were deeply interconnected with many other components such as the graphics engine could not be tested this way.

The portions of the project which can be run independently are the methods for determining hand position, pose recognition, player ids and plane dimensions among others. For these methods, ‘Unit’ testing proved invaluable especially for hand pose recognition. Rather than having to start up the program and stand in front of the Kinect every time an alteration was made to the code, a ‘unit’ test which would pass some pre-recorded labelled data through the method could be run instead. Tests were also written to pass invalid data to the various methods to ensure that proper exceptions would be thrown.

The accuracy of the hand pose recognition, discussed in the ‘Accuracy’ section below, was also determined through ‘unit’ testing by recording many different images from the Kinect, visually inspecting each frame to find what poses the hands were in and then passing them all through the three recognition methods and comparing the results.

5.1.2 Integration Testing

The parts of the project which could not be isolated where tested through ‘Integration Testing’.

“Integration testing means testing two or more dependent software modules as a group.” [27]

This method involved running the entire application after every new component was integrated to ensure that its responses and results were correct. In comparison to unit testing, this approach was quite tedious as it required far more time dedicated to running the program and incorrect code could not be as easily identified. This method can also be quite prone to ‘regressions’ where code that used to work, is broken due to changes elsewhere in the program [27]. Due to these downsides, every effort was made to make as many methods as possible independent so they could be ‘unit’ tested instead.
5.2 Performance

The program’s performance was assessed in terms of the speed and accuracy of its skeletal tracking and hand pose recognition. The computer used was a modest Dell Latitude E5410 with the specifications listed in table 5.1.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Windows 7 Professional 32 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i3 M370 2.4GHZ</td>
</tr>
<tr>
<td>Graphics Card</td>
<td>Intel HD Graphics</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
</tr>
<tr>
<td>Resolution</td>
<td>1024x768</td>
</tr>
</tbody>
</table>

Table 5.1: Specifications of the testing computer.

5.2.1 Speed

To assess the speed of the project application, key areas of the code such as hand pose recognition and skeletal tracking were measured for execution time and time between calls. The stopwatch feature of C#’s System.Diagnostics library was used to take the measurements which can be found in table 5.2.

<table>
<thead>
<tr>
<th>Code</th>
<th>Execution Time</th>
<th>Delay Between Executions</th>
<th>Frames Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand Pose Recognition</td>
<td>6.1-7ms</td>
<td>41.1-52ms</td>
<td>19-24Hz</td>
</tr>
<tr>
<td>Hand Pose State Change</td>
<td>1.04-1.32s</td>
<td>-</td>
<td>0.76-1Hz</td>
</tr>
<tr>
<td>Skeletal Tracking/Mapping</td>
<td>2.7-4.4ms</td>
<td>32.7ms</td>
<td>30.58Hz</td>
</tr>
<tr>
<td>Physics Simulation</td>
<td>0.0014-0.0017ms</td>
<td>14-16ms</td>
<td>62.5-71.42Hz</td>
</tr>
<tr>
<td>Graphics</td>
<td>-</td>
<td>-</td>
<td>93Hz</td>
</tr>
</tbody>
</table>

Table 5.2: Results with anaglyph 3d rendering enabled in Release mode.

To discuss the results in the table above, we need some benchmark like the speed of the eye to compare them with. However, the eye does not operate exactly like a camera and differs from person to person so this project will instead take 25 frames per second (the speed of PAL television broadcasts) as the real-time benchmark [46].

Based on the benchmark and the results in table 5.2, the program runs acceptably for real-time operation in most areas. Hand pose recognition however, runs under the benchmark at 19-24Hz as the recognition code can only handle one depth image at a time so any images received while already executing are simply discarded. Whether this speed is sufficient for real time use however, is a moot point as the hand state only changes when the running average changes, making the frame rate effectively 0.7 to 1Hertz. Depending on the task, this speed is not too bad as the human hand is unlikely to change much
within a second although there will still be a perceived lag as the hand can take less than a second to change pose.

These results remain more or less the same with anaglyph 3d disabled although the graphics engine and physics engine (as its frame rate is tied to the graphics engine) run twice as fast. The physics simulation remains consistent as a fixed timestep is not used but rather one which is adjusted according to the update rate.

### 5.2.2 Accuracy

#### Hand Pose Recognition

To assess the accuracy of the hand recognition methods, 120 depth images with hands, both open and closed were captured making a total of 240 different hands. The poses in the images were labelled and then they were passed through the three recognition systems (defects, bounding rectangles and both). The results were compared with the labels and the accuracy was found to be:

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Bounding Rectangles</td>
<td>34% (82 out of 240)</td>
</tr>
<tr>
<td>Using Convexity Defects</td>
<td>74.58% (179 out of 240)</td>
</tr>
<tr>
<td>Using Both</td>
<td>75.42% (181 out of 240)</td>
</tr>
</tbody>
</table>

In a gaming environment, the methods show reasonable accuracy (with the exception of the bounding rectangles method) although the hit rate might be considered unacceptable in medical or industrial environments. From visual inspection of the results, recognition was found to fail when:

- The hand is at an angle (all three methods) (fig 5.1(a)).
- The hand is moving too fast (all three methods).
- The fingers are close together but the thumb is outstretched (bounding rectangles and combined methods) (fig 5.1(b)).
- The arm is included in the image (bounding rectangles and combined methods) (fig 5.1(c)).
- The fingers are close together (defects method).

These failures are due to limitations in the three recognition methods. The project application mitigates some of these issues by using the running average and only taking readings when the skeleton is not moving too much.
Skeletal Tracking

The accuracy of the skeletal tracking was assessed through user testing. It was found that the virtual hands were reasonably responsive but that the tracking completely failed when joints were too close together or obscured. The range of the skeletal tracking was found by standing as close as physically possible to the Kinect and slowly moving away from it while testing its responsiveness. It was found that the Kinect was unable to reliably track the skeleton outside of 1.2-3.5 metres and that the hand pose recognition was best at around 1.8 metres as the images which were used in the design of the recognition methods were taken at this depth.

5.3 Problems

Throughout the development of the project application, the following issues were encountered to which no resolution could either be found or implemented in a reasonable time frame:

**Poor skeletal tracking outside operational range**

As the project used the 2010 version of the Kinect, the operational range is only 1.2 to 3.5 metres. Outside this range the Kinect is incapable of tracking the player. If the 2012 Kinect was used, this range could be extended to 0.4-3.5 metres.

**Poor hand tracking in certain conditions**

As outlined in the section on hand pose recognition above, the recognition methods fail in a number of different situations.

**Unpredictable tracking if joints overlapping or obscured**

The Kinect cannot identify correctly any joints which are too close together or obscured as it has trouble distinguishing between them. This issue means the Kinect cannot be used for applications that require the hands to be close together.

**Broken compositor scripts**

In early prototypes of the application, attempts were made to port the Ogre Stereo
Vision Manager addon [47] to Mogre to implement anaglyph 3d. This addon used compositor scripts [43] to perform the effect, which unfortunately appear to be broken in the version of Mogre used (1.71) as even the bundled example scripts do not fully work.

**Lack of occlusion culling**

Mogre does not feature any form of occlusion culling which is a method of increasing performance by not rendering objects that are invisible to the camera [48]. The lack of this feature made certain plans such as creating objects out of volumetric pixels (so they could be broken apart) instead of meshes, impractical as the frame rate drop would be too significant.
Chapter 6

Conclusion

By creating a game which the player can interact with, without the use of a physical controller, the stated goal of the project has been achieved. All of the planned features such as hand pose recognition, skeletal tracking and anaglyph 3d have been successfully implemented and the final application is capable of tracking the player, mapping their movements to the virtual world and recognising their gestures and hand poses. The project has also demonstrated the great potential the Kinect has for use as an alternative input method.

However the project implementation is not perfect, as the accuracy and speed of the hand pose recognition could be improved further and certainly would not be acceptable for serious use. The Kinect also might not be ideal for situations where the user’s hands need to work closely together as it has trouble identifying joints which are overlapping or obscured. However, many of the problems encountered were as a result of decisions made in the project’s development before the limitations and nuances of the hardware and software were discovered and could have been avoided.

The project’s silver lining is that much has been learned in the areas of program structure, game design, computer graphics, computer vision and 3d modelling which will prove invaluable in future work. Many new skills have been acquired through the learning of a new programming language, testing techniques, graphics engines, physics engines and so on which will remain long after the project’s completion and the lessons gleaned could be easily used to improve future versions of the application.

In terms of future work on the application, there is much that could be improved. First and foremost, the Kinect could be upgraded to the newer 2012 variant which would allow us to take advantage of potentially higher resolution hand images and therefore more complex hand poses. Multiple sensors instead of a just one could be used to capture the player which would reduce the chances of joints being obscured and could reduce the noise in the depth image. Attempts could also be made to implement the more advanced recognition schemes discussed in the ‘Review’ chapter. Finally, the gameplay could be considerably improved with more levels and objectives or the application could even be extended to feature object deformation.
Source Code

The source code for the project application can be found on the attached cd.
Bibliography


16. The Open Toolkit, OpenTK Homepage, (Date Last Accessed 28/3/2012), http://www.opentk.com/

17. EMGU CV, EMGU Homepage, (Date Last Accessed 28/3/2012), http://www.emgu.com


19. IZ3D, IZ3D Homepage, (Date Last Accessed 28/3/2012), http://www.iz3d.com/


22. Glue Editor Lite, Craftwork Games, (Date Last Accessed 28/3/2012), http://www.craftworkgames.com/blog/glue-editor-a-scene-editor-for-ogre/

23. Mogre dotSceneLoader, (Date Last Accessed 28/3/2012), http://www.ogre3d.org/tikiwiki/MOGRE+dotSceneLoader


26. Nunit 2.6, Nunit Homepage, (Date Last Accessed 28/3/2012), http://www.nunit.org/


[38] T. Watkins, Convex Sets Definition, San Jose State University, (Date Last Accessed 28/3/2012), http://www.sjsu.edu/faculty/watkins/convex.htm


