UNIVERSITY OF SOUTHERN QUEENSLAND
FACULTY OF HEALTH ENGINEERING AND SCIENCES

Real-Time Strategy Game Control for Mobile Robots

A Dissertation Submitted by:

Joshua Muhldorff

In fulfillment of the requirements of:

Courses ENG4111 and ENG4112 – Research Project

Towards the degree of:

Bachelor of Engineering Honours (Mechatronic)

Submitted October 2017

Abstract

As society has progressed, technology has made major advancements. Practical technologies have become more prominent making many laborious roles redundant through remote controlled and autonomous robots, and recreational technologies have seen great success with the gaming industry being worth tens of billions of dollars. Through the exploration of both mobile robots and real-time gaming systems it was identified that the knowledge of real-time gaming control can be of significant aid to the industry of remotely controlled robots.

An Xbox Kinect Sensor 1.0 was employed along with the Unity game engine to provide feedback to a user through a gaming system. To limit the scope of the project, the focus was made to limit the feedback for the user to have minimal data transfer rates and be focusing solely on the user's situational awareness. This task was not able to be successfully completed throughout the duration of the project, however other components of the research identified the potential of this topic. Through the successful integration of the Xbox Kinect Sensor and Unity, sufficient justification for further research in the field was found. This research includes the exploration of control methods, memory mapping through the program development and real-world applications and testing of a mobile robot.

Limitations of use

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Certificate of Dissertation

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Joshua Muhldorff

[redacted]

[redacted]

Signature

Acknowledgements

Firstly, I would like to acknowledge the assistance of my supervisor Dr Tobias Low. Especially for the help with the project initiation, project recourses and conceptualising ideas on how to get underway. Then for the continued support with any questions I had throughout the year. Without your assistance I would not have even known where to start.

I would like to thank my family and friends and also the academic staff at the University of Southern Queensland for their support throughout the course of my studies. To my parents, Stephen and Deanna, thank you for keeping a roof over my head and keeping me fed for the past 22 years. I will never be able to repay you for all you have done for me.

Lastly, I would like to give a special thank you to Mitch and to my girlfriend Saranna. Mitch, the constant messaging every week to check my progress and to tell me to keep doing a little more each week has been greatly appreciated. Saranna, the more immediate support throughout the year, when stress has been high or motivation has been lacking, has been more valuable than you will ever know. Thank you both.

Contents

List of Figures

List of Appendices

The following list includes documents contributing to this dissertation as appendices.

1. Appendix A – Project Specification
2. Appendix B – Project Timeline
3. Appendix C – Program Code

Chapter 1

Introduction

This dissertation aims to explore the potential of Real-time Strategy Game Control for Mobile Robots. This will be accomplished by developing a program to run a visual input (Xbox Kinect Sensor) through an existing game engine (Unity). A testing process will then be undertaken on a number of different operators to give results on the success of the project.

1.1 Objectives

The objective of the project is to explore the potential of how well real-time gaming methods can be developed to control mobile robots. This can be broken down into the more explicit steps as found in the project specification:

1. Explore current real-time gaming systems and methods of mobile robot control
2. Implement a visual input into a game engine
3. Optimise the input to be a real-time system that can be interpreted by an operator
4. Test the usability of the system with a range of operators.

The research is found greatly in step four with the testing of the usability of the program among a series of operators. While the development of the code is the major part of the project the research is only completed with the application at the end.

If time permits these further objectives will be covered:

5. Apply the system to an existing mobile robot platform
6. Test the usability of the mobile robot system with a range of operators

1.1.1 Expected Outcomes

Upon completion of the project it is expected that the objectives have been achieved. This will give results as to whether current methods of real-time gaming can be used for purposes beyond entertainment and have potential benefit to society through mobile robot control.

If successful the outcome could be used in real applications beyond the bounds of this project.

1.2 Project Background

The project background will discuss the thinking behind the project and the major resources that will be used throughout.

1.2.1 Project Initiation

'Real-Time Strategy Game Control for Mobile Robots' is a faculty offered project at the University of Southern Queensland with the intent of filling the gap in literature around navigation of mobile robots in remote and limited access locations with the utilisation of strategy game control. This gap will be further explored in chapter 2.

While technology has made great advances in recent times there is still many applications where mobile robots are required, but are limited in control and situational awareness due to a lack of a reliable wireless networks, with and adequate bandwidth, where mobile robots are frequently needed.

For the benefit of the user, it is necessary that the information about the robot's environment is presented in real-time. Game engines have been successful with this, hosting online tournaments where multiple players can interact in real-time. The project will utilise and expand on the existing success of game engines by adopting a game engine to interpret and present stripped back data from an Xbox Kinect sensor for use with mobile robot control. The data will be kept as limited as possible so that a user is presented with the minimal information needed to navigate the robot through an unfamiliar environment.

With success, this project will then be appropriate for use in many real-life operations where current technology is limited in its capabilities. Such operations may include:

- Military – Such as bomb diffusing.
- Civilian – Search and Rescue missions.
- Workplace – Hazardous workplaces such as volatile mine shafts or locations of chemical and gas spills where humans are at risk of injury, death or health implications.

1.2.2 Unity Game Engine

The use of the unity game engine is to provide the platform for the project. Many real-time gaming systems exist already and run with great success. In the project, the difference with the use of the

game engine will be to learn and build a map with the real-world surroundings. The game engine will then use that as the playing field and learn more as the subject progresses. In successful real-time game methods, the map and all other game objects in the map is already known, with just user input needing to be learnt. This means much more detail can be provided to the user in real time with consoles and computers, with large amounts of RAM and dedicated graphics cards, storing and running the environment with ease. An example of the quality of maps is shown in figure 1.

Figure 1: Unity Game engine - graphics capabilities.

As mentioned the information will need to be simplified for the purpose of the project. The simplified depth data will be used to show a top down map of where free space and where obstacles should be expected in the environment.

1.2.3 Xbox Kinect Sensor

The Xbox Kinect sensor is essentially a 3D scanner utilising a red green blue (RGB) camera and infrared emitter and collector to provide a 3D scan of an environment, usually a room. With the software, this scan can then provide estimates of depth and provide perspective images from other places in the room such as a top down view or side view, of what is the focus of the sensor. This means the sensor is great at being aware of depth and motion, hence why motion is often used as a game control through the sensor. In this project, the sensors 3D abilities will be used to learn new environments to be given as a map to the user through the game engine.

It is noted that Xbox Kinect 1.0 is utilised for the project, as shown in figure 2, which specifies a 320 x 240 pixel depth image. This depth image provides a 58.5 x 46.6 degree field of view or approximately 5 pixels per degree.

Figure 2: Xbox Kinect Sensor 1.0

1.3 Project Resources

The resources required to complete the project will be the hardware and software. The research aspect of the project will be largely associated with existing software and utilising it in new applications.

Extended resource planning is specified in section 3.5, including a list of required components. A further detailed application and development of software can be found the chapter 4.

1.4 Chapter Summary

The purpose of the project has been presented throughout this chapter, acknowledging the gap in literature and how it will be explored with the aid of Unity and an Xbox Kinect sensor. It has also emphasised the objectives and the ways in which the project can benefit industries into the future.

1.5 Report Outline

The contents of the dissertation will comply with the following chapter outlines:

Chapter 1: Introduction

The introductions will present the concept of the project and the background information around it along with objectives and expected outcomes.

Chapter 2: Literature Review

This chapter will explore all available existing literature, within reason, relating to the project. The result of this will be to provide an extensive knowledge base for the project by determining the gap in the literature for real-time game applications as a control method for mobile robots. This will also ensure that time is not wasted rediscovering existing research.

Chapter 3: Methodology

This chapter will cover the considerations necessary to successfully complete a well-executed and structured research project. Included in this chapter is Project methodology, Risk assessment, Intellectual Property considerations and Experiment methodology.

Chapter 4: Unity Game Engine

The unity game engine is a major tool in the project. Related interactions with the Kinect sensor, set up and problems or successes with the program will discussed throughout this chapter to assist understanding towards the final result of the project.

Chapter 5: Code Implementation

Development and implementation of the code will be discussed throughout this chapter. While the writing of the code is not considered to be the focus of the research it is a crucial part in the success of the project. The code will be subject of testing in the following chapter.

Chapter 6: Results and Discussion

Covered in the results and discussion chapter will be reporting of procedures and results discovered from the testing phase of the project in which the majority of the research will be conducted.

Chapter 7: Conclusion

This chapter will discuss and conclude the results found within the testing. Furthermore, discussion will be conducted on how the findings can implemented into real world situations or built upon for further research.

Chapter 8: Reference List

This chapter will cover a list of references acknowledging the authors of cited works.

Chapter 9: Appendices

This chapter will include all appendices relevant to the report that are too extensive or are nonessential in the primary reporting. This will include things such as project specification and coding scripts.

Chapter 2

Literature Review

This chapter will explore all available existing literature, within reason, relating to the project. The result of this will be to provide an extensive knowledge base for the project by determining the gap in the literature for real-time game applications as a control method for mobile robots. This will also ensure that time is not wasted rediscovering existing research.

In particular the exploration of the subtopics of the project to be reviewed in this chapter are; methods of processing images for real-time applications, real-time gaming applications and control applicable to mobile robots and mobile robots themselves. While some topics may overlap in certain parts they will be categorised as best possible.

2.1 Imaging

Throughout the imaging section two major points will be explored through their existing literature, these are image formats and the Xbox Kinect sensor as an awareness sensor. First covered will be image formats to understand the ways image data can be presented and the data resources they consume or require with focus on this data in transmission. The Xbox Kinect sensor will follow to explore project specific data and image format to understand how the data will be presented and how it should then be manipulated.

2.1.1 Image formats

To understand how the data from the Kinect sensor can best be interpreted and presented it is necessary to explore relevant literature to understand this. For each different image format, there will be corresponding algorithms on how to achieve them and different data requirements for their processing and then transmission.

Hansard and Miles describe a camera that takes a depth image as a time of flight camera (2013). These cameras use infrared or other structured light through an emitter that is then collected, measuring the phase delay then provides an accurate depth measurement which can be presented as an image. For application of the data it is presented as a matrix of numbers associating to a depth which will often be displayed as an image with the colour then representing the depth values.

With a standard RGB camera an image can be interpreted to show the most important information. Such techniques are considered as filtering and are favoured to be used in machine vision (Davies, 2012; Parker, 2011). Detection of edges in an image has long been used as a method of locating important data points while being able to disregard data that is not needed for the desired task. Davies and Parker both discuss various algorithms and methods of achieving edge imaging. Other image types that similarly reduce the data set but don't disregard information needed for clear human analysis are greyscale and black and white images.

Arabnia, Hamid, Deligiannidis and Leonidas reaffirm the literature findings with their book 'Emerging trends in Image Processing, Computer Vision, and Pattern Recognition' (2015). The types of images and processing methods are reported extensively, while they are of greatest use for autonomous robots, the algorithms and filters from the literature sources will contribute to this project in limiting the amount of information that needs to be transmitted.

2.1.2 Xbox Kinect Sensor

In the exploration of literature it appears the Xbox Kinect sensor has an undisputed reputation for being a suitable robot sensor (Khoshelham, Elberink, 2012; Isafriade, Osunmakinde, Bagula, 2013; Ghani, Sahari, 2017). The benefits found for the Kinect sensor include the low cost, simplicity of connection with a PC and the performance of the sensor itself. As stated in the background information the Kinect senor utilises a RGB camera and an infrared emitter and collector providing depth images at a resolution of 320 x 240 and colour images at 640 x 480, both at a rate of 30 frames per second (Isafriade et al., 2013). Other features of the sensor are the ability to capture audio through four different microphones, which can interpret sound direction, along with a tilt motor. The latter two features of the sensor are considered not to be of use in the project and therefore are frequently emitted from relevant literature.

Khoshelham et al. through their work acknowledge the potential of the Kinect sensor as a tool for indoor mapping applications, and utilise it for such a task, as required in the project (2012). From their findings it can be expected that errors of 4cm can be expected in the depth map at the maximum range while most errors will be in the range of a few millimetres. These errors are stated to be occurring from a combination of flaws in the infrared collector, the surface of certain objects in the chosen environment or the setup of the device.

Indoor environments are typically considered more controlled than outdoor or underground environments, due to lighting and other uncontrollable factors. Isafriade et al. have found through their research that the Xbox Kinect sensor is suitable terrains such and mines and underground tunnels

with use of entropy and statistical region merging methods (2013). Their research finds suitable paths for autonomous robots to travel in such conditions with the Kinect sensor as the primary sensor.

Conclusions reached from the existing literature is that the Kinect sensor is a very suitable tool for mobile robots, frequently used for development and sensory applications of robots despite the natural intent of a controller for a game console.

2.2 Existing Real-Time Gaming Methods

Existing methods of real time gaming, in a general manner will be explored through this section. Methods explored may be non-relevant to the outcome of this project but essential to the understanding of real time gaming.

As mentioned briefly in section 1.2.2, there are many existing methods of real-time gaming. The three most popular examples of real-time gaming systems that proven majorly successful are PC gaming, the PlayStation by Sony, and the Xbox by Microsoft. Software companies design real-time games and most of these are available across all three platforms.

Huat and Teo present research on the development of real-time strategy games throughout history. Their works indicate that online games spawned from strategy games that have existed long before a computer such as card games or board games. The study proposes that these games were adopted by computer programmers and now technology exists where even the best strategy game players can be beater by a computer through various algorithms. While the study is limited in its scope the strategy algorithms that were found expanded to real-time strategy games with military focuses. While more complex games, with greater variables and possible outcomes have been developed, the games can be very successful and useful for forming strategies.

'The Console Market' by Kirriemuir presents how popular real-time gaming systems have become. Kirriemuir states that gaming companies such as Sega and Nintendo now have budgets for each game that rival blockbuster movies. This paper was published and 2000 and many significant advances have been made since, however this supports how vast and relevant real-time gaming was, and has become in modern society.

The Xbox and PlayStation as mentioned above, lead the industry for gaming consoles. The most recent flagship console from Sony is a variant of their well-known 'PlayStation 4' (PS4), the 'PS4 Pro'. Microsoft is soon to release their competitor for this market, an update to their' Xbox One' named the 'Xbox One X'. The specifications of both consoles are similar, however, Microsoft quotes that the Xbox One X has "40% more power than any other console". The console can run real-time games in 4k Ultra

High Definition quality (3,840 x 2,160 pixels) at a frame rate of 60 frames per second. With such power in this console extensive cooling systems are required to maintain a suitable temperature. A supercharger styled fan is used in conjunction with liquid cooling to prevent over-heating.

2.3 Real-Time Game Control

Project specific control and relating literature to existing methods of presenting real world real-time data through a game engine will be explored under this section. The extent of which this project has been previously covered in literature will be of significant focus.

Throughout the literature it was found that utilising game engines for the use of mobile robots favoured being used for the integration of both human and robots to coexist in a certain application. Both 'The Robot Engine – Making The Unity 3D Game Engine Work For HRI' and 'Imperfect Robot Control in a Mixed Reality Game to Teach Hybrid Human-Robot Team Coordination', display the usability of game engine to coordinate robots with humans or Human-Robot Interaction (HRI) (Bartneck, Soucy, Fleuret, & Sandoval, 2015; Sosa, Keyes-Garcia, Stanton, Gonzalez, Perez, & Toups, 2015). Another similarity found within these two studies is that they used the Unity Game engine, the Robot engine used the program as a base to make a new program purely for the assimilation of humans and robots whereas the other study used Unity to build a small strategy game as a training device so HRI can be achieved.

Highlighted as an advantage of controlling mobile robots through a game engine is that it is possible to create new software and then run robot simulations in real-time as a method of development before real models are required which can be costly (Bartneck et al., 2015). While not directly related to implementation of the project it is supported in this literature of the capabilities of existing game engines to be used for real-time operations.

A paper by Ohashi, Ochiai, & Kato (2014) closely covers topics to be examined in this project however assuming a fixed stable network for data transfer. The paper looks into a mobile robot that maps the real world via a camera and implanted through Unity. While the world is intended to be mapped from the real world, errors are found to occur and need to be considered. The errors are with the robot having a real and an idealistic state, meaning the real state will be subject to slip and other delays where the virtual robot in the game engine may move differently in its perceived world (Ohashi et al., 2014). Ways to compensate for these errors must be considered.

While the paper by Ohashi et al. (2014) is known to cover similar topics such as the same game engine with a similar application, the data is of high intensity and dependant on a fix network, still leaving open the suggested gap in literature. The three-dimensional view given by Ohashi et al. (2014) will

need to be disregarded and rewritten to a less data intense two-dimensional view. Updates also need to be considered so the state of the virtual robot in the game engine is dependent on the location of the real robot, this will eliminate the potential of errors in the two states.

2.4 Mobile Robots

The literature explored for mobile robots will be a general overview to develop understanding of what exists and the requirements of mobile robots, with a particular focus on ground based robots. Basics covered will be with relation to existing methods of navigation and remote sensing and existing control methods not related to game engines.

There a several ways in which a robot can navigate an environment, these can either be fully autonomous or dependant on human intuition to evaluate the information. Methods of navigation are given by use of different sensors or a combination of sensors. Key systems for navigation are inertial navigation systems and the Global Positioning System (GPS) (Cook 2011). Other typical sensors for navigation include, but are not limited to: vision, touch sensors, ultrasonic and infrared emitters and collectors.

Inertial navigation systems utilise gyroscopes to determine torques and angular velocity changes in a robot (Cook 2011), commonly this system will be used to stabilise a robot in flight or also levelling of cameras in both flight and ground based robots. This technology is utilised by the Xbox Kinect Sensor and can level the vision of the sensor to a degree if required.

GPS is a common tool in navigation, relying on a series of 24 satellites that orbit the earth to provide the accurate location of a receiver on the ground (Cook 2011). The receivers can be of limited size and cost making them suitable for many robotic systems requiring navigation. Upscaling of the project would require deeper consideration of GPS navigation.

Previously mentioned as a tool of navigation, remote sensing is a required aspect of mobile robots to assist with situational awareness and other functions that may be required from the robot. Cook discusses the use of cameras and radar as primary sensory devices for remote sensing (2011). These cameras are used in various set ups with, in the literature it is found that cross referencing of data with a stereo set camera can increase awareness and accuracy of data. In the literature information on panning of cameras is provided which is notable and will be addressed should the field of view from the Kinect sensor be considered too narrow.

Control of a robot is essential for a functional design. The term kinematics descries the task required to be completed by the robot to move from point to point with regards to displacements and velocities

or a local or global coordinate system. Methods of control then take into consideration the torques and velocities required to achieve these motions. It is presented by Tzafestas, (2014) that there are five conventional methods of controlling robots. These five methods are:

i. The Lyapunov-Based Method
ii. Affine Systems and Invariant Manifold Methods
iii. Adaptive and Robust Methods
iv. Fuzzy and Neural Methods
v. Vision Based Methods

Each method has further small variations of how they can be achieved to complete the goal of robot control. The first method is the most relevant to the project and will be researched more intently if required. Other methods presented in the literature relate to more complex control, including control of omnidirectional wheels, aeronautical robots and then vision based control. While vision is the primary sensor for the project control through vision is beyond the scope.

As the control of the robot is not the focus of the research, an in-depth exploration of control is not deemed necessary however a general understanding of terms and methods that maybe be encountered must be understood. Further investigation of control is not necessary at this point.

2.5 Chapter Summary

In summary, the presented literature was considered sufficient evidence that the topic complies with the gap in literature. Both the major topics of the real-time game control and mobile robots presented vast resources to provide basis for proceeding without rediscovering existing work. While some of the literature has surpassed the scope of the project it has required an understanding for what is considered in similar operations to highlight important considerations should the literature need to be revisited to assist development of the project. With the foundation provided with existing literature the linking real-time game control and mobile robots will be undertaken to provide robot control and user awareness through a game engine.

Chapter 3

Methodology

This chapter will cover the considerations necessary to successfully complete a well-executed and structured research project. Included in this chapter is Project methodology, Risk assessment, Intellectual Property considerations and Experiment methodology.

3.1 Project Methodology

To complete the project a successfully a methodology will need to be employed. This methodology will involve research and testing criteria to be completed within a set time frame that will be followed throughout the course of the project. Reporting of all steps of the project methodology will be done as each step is completed to provide results as accurate as possible.

The project methodology is as follows:

1. Identify the scope of the Project. This includes determining the desired outcome of the device's function and project timelines as found in appendix B.
2. Research will be done to find useful literature gaining an extensive knowledge on the topic and ensuring previous work is not being repeated. Research will focus on, but not be limited to, Unity game engine and Xbox Kinect sensor along with real time image processing.
3. Refine what the intention of the code is through research results to discover which image forms and sensors should be utilised.
4. Develop a code to translate the depth image to a top down yes/no map for potential robot navigation paths.
5. Refine the code to run in real-time.
6. Testing of the code for correct operation and then for operation with various operators.
7. Discuss the finding of the research and consider future work that can be done to further validate the success of the project
8. If time permits the project will be extended to involve a physical application to a mobile robot base, for more accurate results and testing of the project success.

3.2 Risk Assessment

This project being mostly utilising a computer will have minimal risks involved. However, there will still be testing methods and interaction with other persons in the completion of the project. Due to this, risks must be identified, assessed and managed to reduce the possibility of harm to both personnel and equipment throughout the duration of the project.

3.2.1 Risk Identification

The risks of this project can be identified in two separate segments, the first being in development stage of the project. Secondly risk will also be identified with testing the stage of the project. The risks associated with the development stage include:

1. Prolonged time at the computer – eye/muscle strain.
2. Fatigue or burn out.
3. Damage to the hardware.

Further risks include those associated with testing and the interaction of people with the software and hardware. The identified risks are:

4. Damage to the hardware or software.
5. Damage to the environment – room and surroundings that the program is tested in and around.
6. Injury to the operator.

3.2.2 Risk Evaluation

The risks noted need to be evaluated on likelihood and severity to determine whether risk management measures shall be required to reduce the severity, likelihood or eliminate the risk completely. All of the risks mentioned are proposed as being of moderate likelihood with low - moderate severity if they occur. The chance of the identified risks occurring can easily be avoided or reduced and is likely to not cause significant damage immediately or over the long term. To reduce the likelihood and severity of the risks a risk management plan will be developed.

3.2.3 Risk Management

To manage the identified risks some precautions and steps will be taken. These precautions and steps are as listed:

1. Ensure a clear workspace and that regular breaks are taken to avoid strain and fatigue during the developing stage.
2. If time permits for robot testing, ensure failsafe steps are put into the code to provide correct operation, shut down in the case of error and if in the case of a moving robot; eliminate the possibility of contact with surrounds.
3. Ensure all operators have a clear understanding of what the program will do and what is expected of them.
4. Ensure where any testing is taking place the environment is tidy and clear of obstacles that may cause hazard to operators or the hardware.

Although there is always a chance that accidents may happen, following the listed precautions is expected to keep the likelihood and severity of any possible mishaps to a minimum.

3.2.4 Project Consequential Effects

Upon completion of this project, the findings and designs put forward may be utilised by future students or researchers to build upon or refine the project. In any of the suggested cases it will be my responsibility to ensure that the information provided in this report is safe and ethical, so I am not liable for any damages that may potentially occur in future works.

To remove future liability, it must be ensured that proper documentation of the project is completed in accordance with section 3.4. This will provide clear information for future developers and researchers with what limitations are found in the project. The detail in this reporting must be of a high standard to avoid any misunderstandings in future works.

Completion of the project in this manner will greatly reduce chance of negative consequences arising now and in future works. With a successful completion of the project consequential effects will be expected to positive by providing personal development and insight into Real-time Strategy Game control for mobile robots, among others.

3.3 Project Intellectual Property

Due to time, undergraduate knowledge and resource constraints, useful but not ground-breaking findings are expected to be produced from this project. However, future students may wish to use this project for guidance, or a base for more in depth research. So this project can be built upon by future students, any findings or designs in this report may be used with acknowledgement where information used is from.

3.4 Experiment Methodology

This section will discuss the methodology behind physical experimentations in the project and how they will be run and documented throughout the project.

The project will consist of two experiment sections, the developing stage and the results stage. The developing stage will be the initial process to create a program to conform to the project specification details, this can be implemented for further testing to see if the project has been a success among other users. The steps taken in the experiment process are as listed:

1. Get the Kinect Sensor to input into unity – All drivers and necessary software should be installed. Some sample codes can then be run to find the Kinect sensor works and can communicate with the game engine.
2. Develop the code to view the correct environment – The sample codes will not provide the necessary format to complete the project. The code must be extensively developed and changed to be of use.
3. Refine code to provide a less data intensive operation – The focus of the project is to provide real time data. Once the correct information is being gathered it must be refined to display to the user in real time.
4. Test the code to see if it runs in real time – The code will be designed to respond in a certain way, it then must be tested to see if this is true. If not step 3 must be completed again.
5. Gain approval to use further users – certain documentation must be completed
6. Run the test with multiple users/operators – This will be completed to ultimately provide if the project was successful or not, by getting people to work in an unfamiliar environment with the “game”.
7. Document results – results and progress will be documented throughout the project however particular focus will be put on the reporting of results given from step 6.

Point 6 provided above must be expanded on to provide detail of the second stage of testing. This will discuss direct methods to proceed with the testing of the device to attain feedback on the success of the program and also being the dominant research component. The steps of testing are as follows:

1. Formalise a set day for testing. As testing will only be able to be completed by one user at a time, a series of half hour sessions will be required throughout the day to cover a range of users. The following steps regarding user tests will be required throughout each individual session.

2. Introduce the user to the program and what will be required of them with testing. The user will be provided with a feedback form to formalise the results of testing.
3. The user will be placed in a room separate to the mobile robot where it will be navigated through the environment. The robot will be controlled by appropriate means to complete three different courses with the Kinect sensor input to Unity being the only tool for the user to navigate the course.
4. All three courses that the user runs will be timed and compared to a fixed time that the robot has completed the course being guided with full situational awareness. If the user gets lost in the course or comes into contact with an obstacle the run will be counted as fail.
5. Above testing will be run as appropriate, when the user completes their three tests they will be asked to complete their feedback form on their experience to add to the research on whether the program is appropriate for use or what further developments should be considered.
6. Upon commencement of all tests the results will be compiled and then scored against full situational awareness times, taking into account excess time taken as a percentage and whilst taking serious account for failed runs, if any occur.

3.5 Resource Planning

3.5.1 Materials

Both physical and digital recourses required for the project:

- Xbox Kinect sensor
- Computer – Personal computer
- Unity Game Engine Software
- Mobile robot platform
- Xbox Kinect windows driver
- Permission/application and compliance to perform a public survey

3.5.2 Lab Access

Due to being an external student lab access is not likely to be required. This may be revaluated further into the project if circumstances change

3.5.3 Workshop Facilities

Work shop and manufacturing applications will not be required as the project is predominantly based around coding methods.

3.5.4 Budget

The project will utilise a laptop and Xbox Kinect sensor that will be of no cost as they are already owned by the University or student. The software used will be of no expense as it is free to download for any user, therefore, no expenses are expected in this project, however a $100 budget will be in place as contingency for unexpected items.

3.5.5 Supervisor

Contact with the supervisor will be kept throughout the project to ensure requirements are met.

- Fortnightly meetings will initially be in place to start up the project, these may be reduced to monthly if found suitable
- Email contact will be kept when necessary
- 21 hours are provided as supervisor time including the making time, supervisor time is to be used sparingly and in an appropriate manner.

Chapter 4

Unity Game Engine

The unity game engine is a major tool in the project. Related interactions with the Kinect sensor, set up and problems or successes with the program will discussed throughout this chapter to assist understanding towards the final result of the project.

4.1 Unity Start Up

Upon the start-up of Unity for the project purposes, certain software needed to be acquired for the project to commence. The Unity game engine is the main essential component and can be downloaded free, for personal use from the Unity3D website. Various versions exist for different operating systems, including windows, IOS and Android. Two different versions of the game engine were utilised for the project due to computer changes.

Once successfully installed, Unity is free to start running and being used for development. For this project, more software was required for the integration of Unity and the Xbox Kinect sensor.

Ensuring that the Xbox Kinect sensor has not yet been connected to the machine, it is required that the software development kit (SDK) drivers for the sensor are downloaded and installed. Only after the completion of the SDK driver installations can the Xbox Kinect Sensor be connected to the machine use for development. Upon connection of the Xbox Kinect sensor, the sensor will be automatically integrated to the machine. If the Sensor is connected before the SDK drivers are installed, errors may occur that prevent the sensor from operating correctly.

Once the sensor and game engine had been installed a wrapper package was located that simplified the integration of necessary tools into Unity. The wrapper package titled “Kinect Wrapper Package for Unity3D” can be located, along with installation instructions, through the link at the end of this section. This package demonstrates firmly the intended use of the sensor as being a controller by providing the ability to track a human’s motion. This is discussed more in Chapter 6.1 – Trial 1. Contents of the package are as listed:

- 1 Sample Scene

    - Kinect Sample
- 12 Unity Scripts
    - Kinect Image Controllers
        - Display Colour
        - Display Depth
    - Kinect Model Controllers
        - Kinect Model Controller V2
        - Kinect Point Controller
    - Kinect Wrapper
        - Depth Wrapper
        - Device or Emulator
        - Kinect Emulator
        - Kinect Interop
        - Kinect Recorder
        - Kinect Sensor
        - Skeleton Wrapper
- 2 Prefabricated (Prefabs) Game objects
    - Kinect Point Man
    - Rainbow Man V6
- 1 Sample game material
    - Rainbow Man Texture

To ensure the Xbox Kinect Sensor has been installed correctly it was found beneficial to install the Kinect 1.0 Developers tool kit, which can be located on the Microsoft website under downloads. This tool kit will piggy back a host program, in this case Unity, which runs and utilises the Xbox Kinect Sensor, while the tool kit provides the secondary displays of raw data from the sensor that can be used to compare if the host program is functioning correctly.

A guide to the Unity set up with an Xbox Kinect Sensor, which was used for this project, can be found at: http://wiki.etc.cmu.edu/unity3d/index.php/Microsoft_Kinect_-_Microsoft_SDK

4.2 Unity Features

The capabilities of the Unity game engine far exceed the requirements of the project. The main features of unity shall be discussed along with the specific features that were relevant to this project.

Upon opening unity and creating a new project Unity presents the home window as shown in figure 3. The main features that are notable are the two work spaces. There is the “game” window which is shown, and then the “scene” window. The game window presents what will be seen by the user from the main camera, essentially their perspective view of the world. The scene window is the where game objects are created and placed in the map and represents the world.

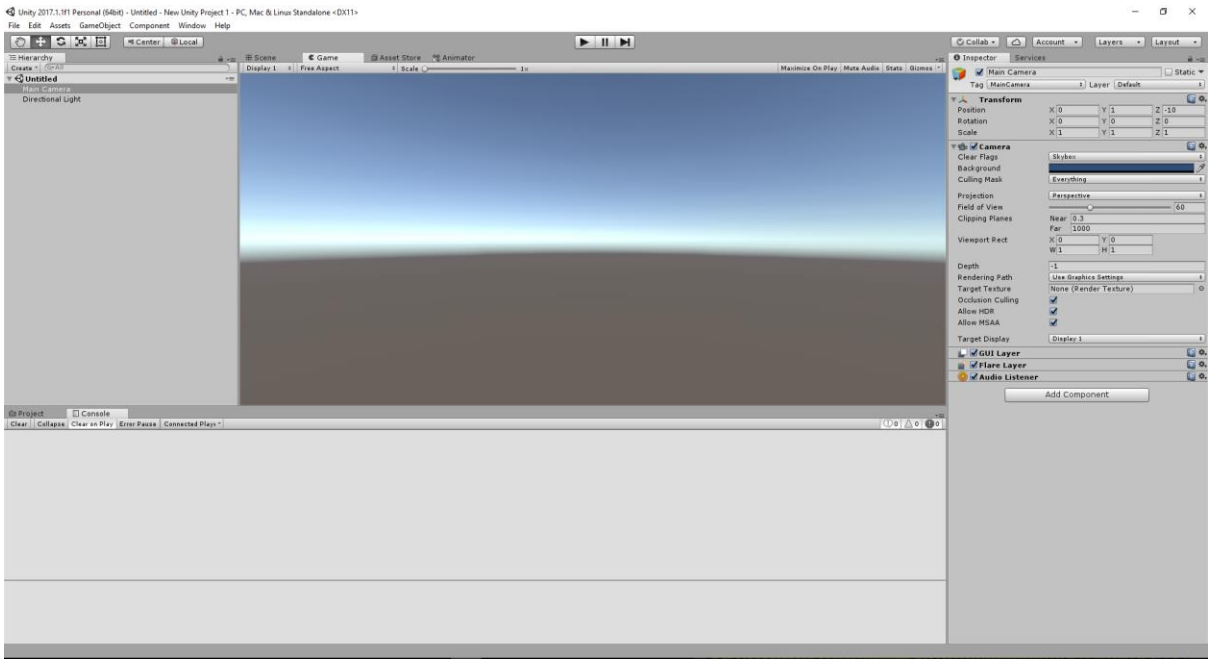


Figure 3: Unity Console Layout

The other two notable features for any project are the “console” and the “inspector”. The console is where errors will be shown and game data and commands can be printed. The inspector on the other hand is the base driver of attaching codes, functions, textures and other required aspects to a game object. Figure 3 demonstrates the default specifications of the main camera in the Inspector window.

For this project, the only game objects that were utilised were two-dimensional planes in a three-dimensional world. The first plane was the depth image imported into unity. The second plane was the top-down depth map. Both planes displayed their information via the use of a texture which was applied to the plane through the inspector window. Textures codes were formed and edited with aid of both Microsoft Visual Studio and Notepad ++ for simplicity.

When creating a plane the object can be specified from the Game Object drop down and is then automatically placed. The tools used to manipulate the plane are then, drag, move, rotate, scale and resize as highlighted in figure 4. Noting the drag function moved the scene relative to the camera and the plane stayed in the same position. Move and resize tools were found the most successful for

manipulation of game objects, while the inspector window was required for final refinement of positioning.

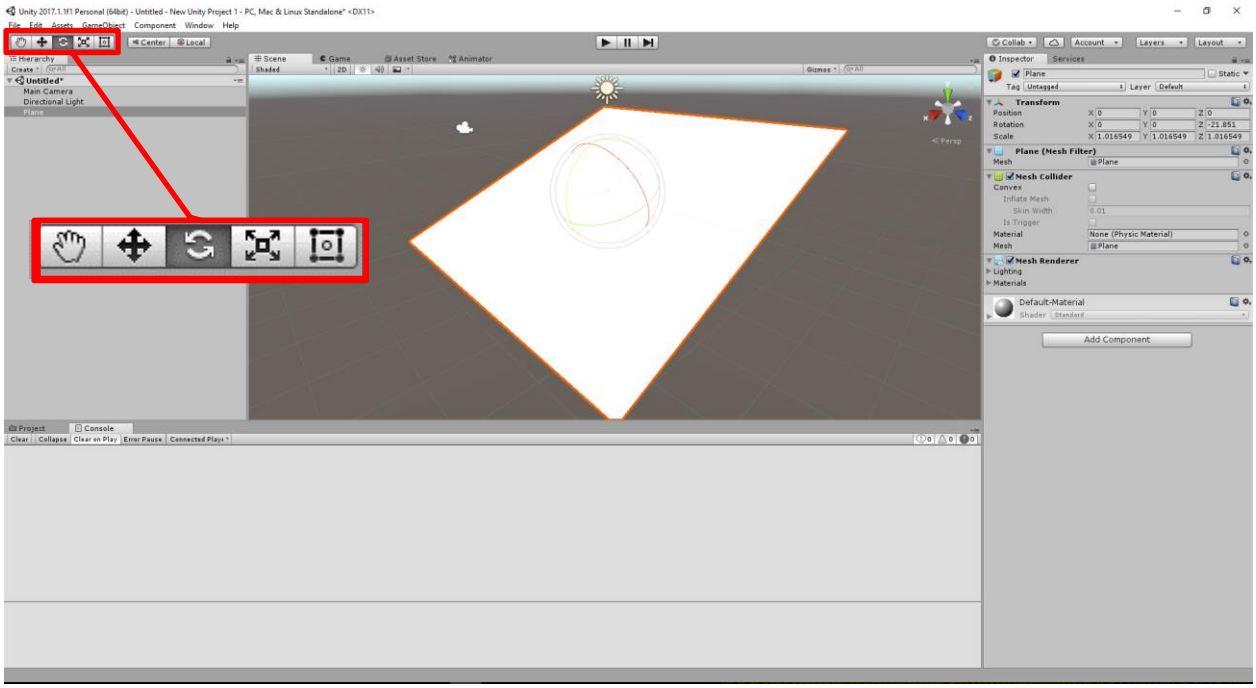


Figure 4: Unity Game Object Creation.

4.3 Findings and Complications

The findings found with relation to the Unity game engine are heavily influenced by the initial run labelled “Trial 1”. Extended findings and complications for the Unity game engine can be found in Chapter 6.1 where trial 1 is documented.

An minor complication that was found after numerous machines had to be used, due to technical difficulties, was the appearance of the error pictured in figure 5. Upon reimporting all game objects when this error occurred, it then appeared again. The error was then ignored after this and had no effect on the result of the program.

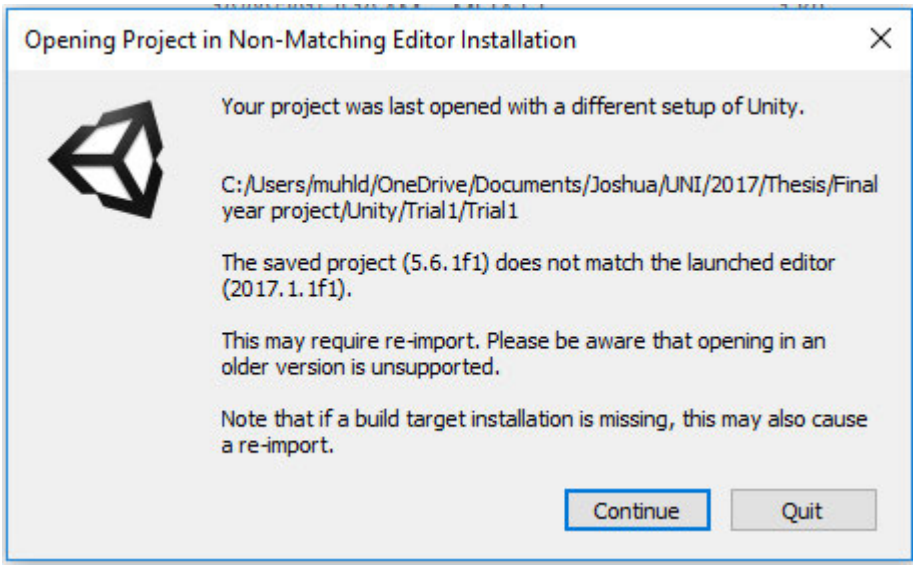


Figure 5: Opening Project in Non-Matching Editor Installation, error.

4.4 Personal Development Recommendations

Learning the skills required to use the Unity game engine within the course of a year is a difficult task. Attaining these skills is a significant attribute of personal development in which increasing coding and gaming design skills can provide knowledge that is transferable to many different tasks.

There can be several different ways to achieve the acquisition of the skills with varying timeframes and levels of success. Trying to achieve a goal through the Unity game engine without a thorough understanding of the coding language and standard conventions of necessary game objects is one of the former methods of learning the game engine and advised against.

Initial methods of learning the game should be through completing the Unity scripting tutorials that are of high standard and cover all beginner aspects of the gaming development. Further project relevant tutorials may be selected as necessary for more complex tasks once a firm base has been made. After this is it recommended that a full guided tutorial on a sample project or game, that is non-specific to the purpose Unity has been employed for, is completed. The roller ball sample is a basic project that can be build, pictured in figure 6, however, it is recommended a sample project of a medium complexity is completed to ensure the knowledge has been acquired.

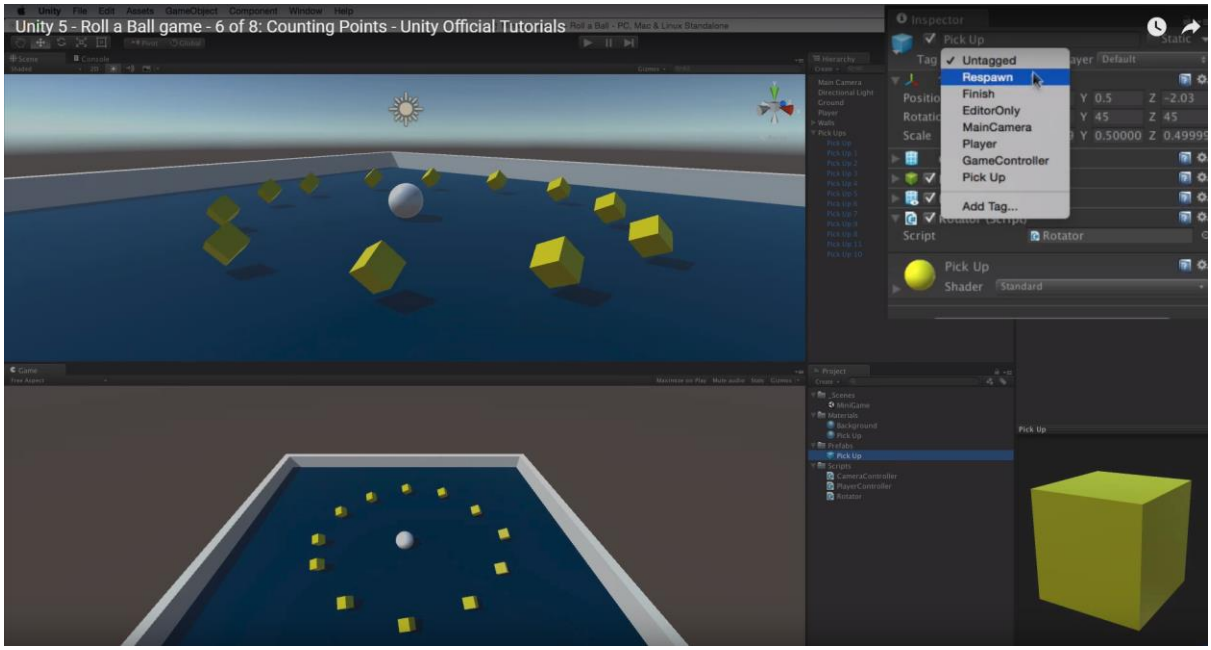


Figure 6: Roller Ball tutorial – as from Unity Official Tutorials

Lastly it should be noted that these tasks are timely and should be completed in the initial phases as time constraints limit the possibility of thorough learning and understanding as the project progresses.

Scripting Tutorials can be found at: https://unity3d.com/learn/tutorials/s/scripting

Project tutorials can be found at: https://unity3d.com/learn/tutorials

Chapter 5

Code Implementation

Development and implementation of the code will be discussed throughout this chapter. While the writing of the code is not considered to be the focus of the research it is a crucial part in the success of the project. The development of the code will be subject of the following chapter.

The intent of the code was to be able to present a user limited information while still providing adequate situational awareness. A greyscale image provides adequate situational awareness to a user where they can see obstacles in an environment almost as well as a full colour image. The depth map provided from the Xbox Kinect sensor runs at 320 x 240 pixels at a frame rate of 30 frames per second. This gives a data transfer rate of approximately 2.3MB/s. To make a new code worthwhile it was devised that the new map would run at the same rate but use some algorithms to simply what the user sees with a final proposed data transfer rate of between 0.9 and 1.0MB/s. The methods proposed to limit the data transfer rate will be discussed in section 5.2.

5.1 Sample Codes

Sample codes were used extensively in this project to initiate the connection between Unity and the Xbox Kinect Sensor. All initial sample codes have been listed in chapter 4, section 4.1, however a list of relevant codes to the final program will be highlighted here, while the code scripts will be shown in Appendix C.

Sample codes used:

- Display depth
- Depth Wrapper
- Device or Emulator
- Kinect Emulator
- Kinect Interop
- Kinect Recorder
- Kinect Sensor

5.2 Main Code

The main code to be developed in this dissertation is in aid of exploring the potential of real-time strategy game control for mobile robots.

It has been devised that the code will aim to maximise the situational awareness of a user controlling a mobile robot with low data intensity real-time feedback. Complex coding will not be required for this task which aids the limitations in time, recourses and pre-existing knowledge.

To provide direction to the code development a flow chart was designed. This flow chart of coding design structure is as shown in figure 7.

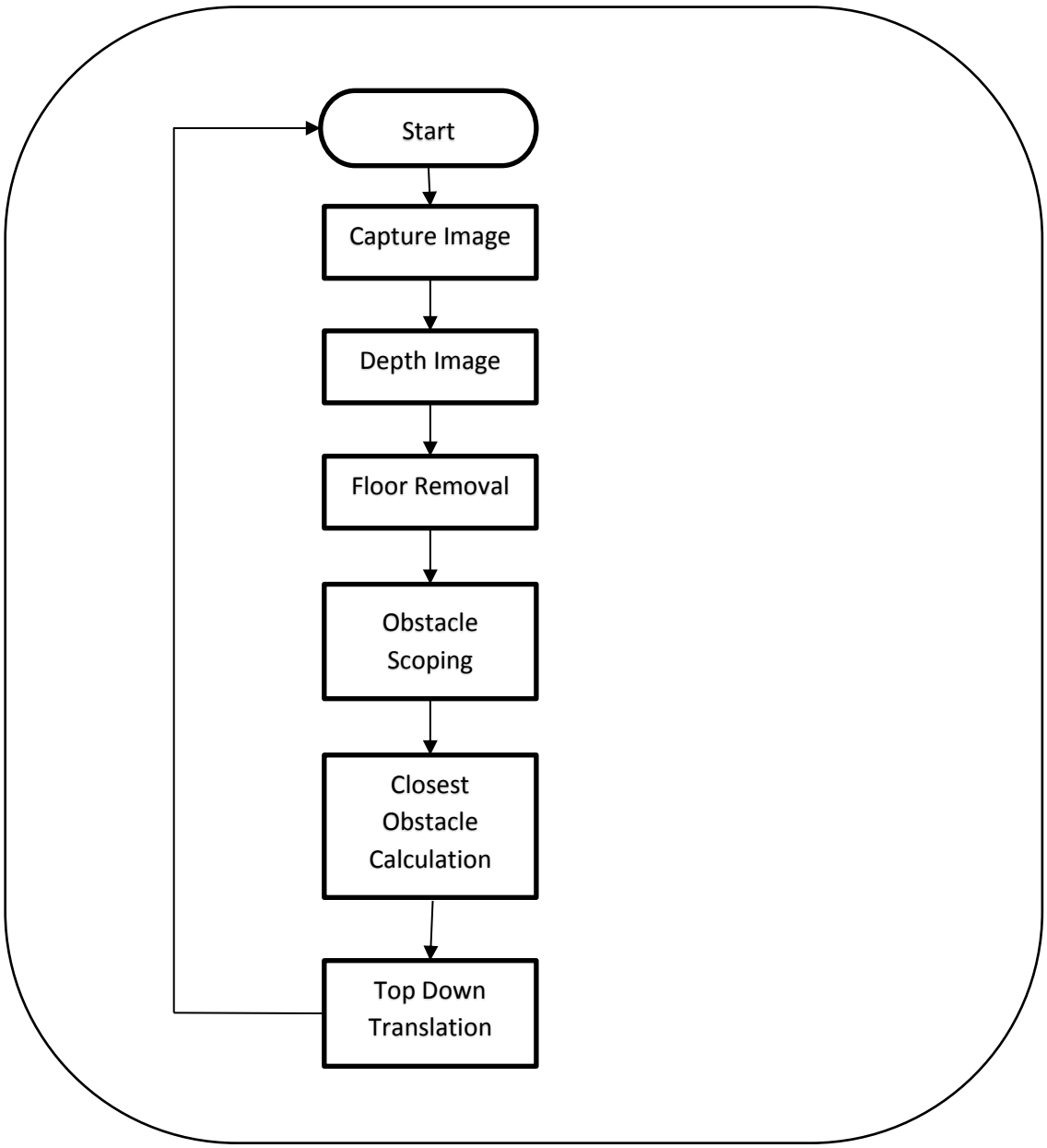


Figure 7: Main code design flow chart

Each process from the main code is further explained in this chapter with the corresponding sections that follow. Results of the development process are shown in Chapter 6.

5.2.1 Depth Isolation

Due to the raw data not being presented in Unity or Kinect Studio it was required that this be isolated from the code in the form of a matrix so the variables in specific scripts and values representing distances could be identified.

5.2.2 Floor Removal

When identifying obstacles from the data, it is intended that the closest object in each direction will be identified and shown as an obstacle, with no other considerations of more distant obstacles being made. This raises an issue with the floor or ground of an environment as the floor will be shown and acknowledged as the closest obstacle, rendering the program as useless. To avoid this, algorithms needed to be adopted to identify the floor and disregard it as an obstacle.

5.2.3 Obstacle Scoping

The Xbox Kinect sensor is intended to sit parallel to a robot base when implanted in future testing. From this stated design intent, it can then be assumed that a ground based robot will not find obstacles in the top half of an image showing the environment ahead of the robot. The lower half of the image will show ground based obstacles that the user will need to identify for the robot to avoid. In this dissertation this will be identified as obstacle scoping, where objects of altitude will be ignored.

This segment of the code will also include the elimination of obstacle shown in the too close range of less than 0.2 metres and those identified too far away at greater than 5 metres.

5.2.4 Closest Obstacle Calculation

Once the obstacles had been scoped it was required that the closest object be identified in each column of pixels, i.e. the smallest value from each column. An algorithm had to be employed for this that would seek out the smallest value for each column and return the desired value to a 320 x 1 array.

5.2.5 Top-Down Image Translation

In the top down image translation, it had to be considered the position of where obstacles would be shown. The 320 x 240 depth image would no longer be the format of the data. Each column of pixels was to be associated to an angle and direction. Meaning closer objects equate to smaller obstacles and distant objects equate to larger obstacles. The grid is intended to be shown as in figure 8, the grid frame will not be shown but the segments will be populated if an obstacle is presented in that space. The grid has a 4.8 metres depth starting at 0.2 metres and ranging to 5 metres from the Xbox Kinect Sensor. This grid will be divided into segments that are 0.183 degrees wide and 10 centimetres deep. While the angle is very small it is representative of one pixel wide. This means that obstacles that are identified are likely to show across several cells of the grid, as obstacles will rarely be 1 pixel wide, this will show better population of the map. The depth is shown at greater intervals as the depth will only be shown as one cell in each direction. The length of an object will not be measured, it is just identified as a distance from the sensor.

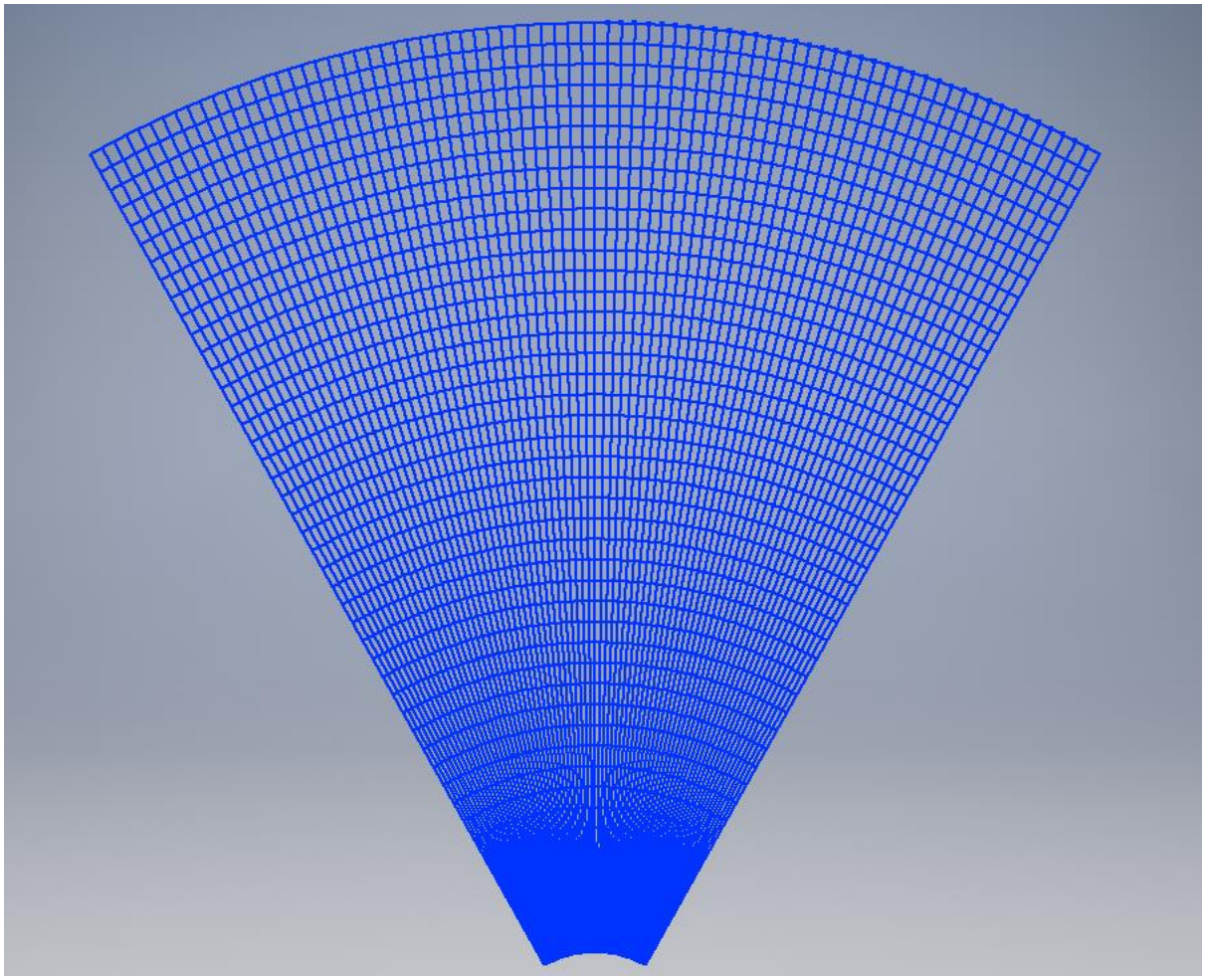


Figure 8: Top down proposed layout

5.2.6 Object Spawning

The final task of the code was to spawn an object in the game scene that represented an object in front in an environment. This is to shown as a “yes/no” map for the user to interpret the best navigation path.

Chapter 6

Results and Discussion

Covered in the results and discussion chapter will be reporting of procedures and results discovered from the testing phase of the project in which the majority of the research will be conducted.

6.1 Trial 1

Trial one was the initial trail to integrate Unity and the Xbox Kinect Sensor. The sample scene, prefabs, textures and codes were utilised from the Microsoft SDK sample package.

Upon start up the game window appeared as shown in figure 9. The main camera showed the Rainbow man which is intended to track a human user in front of the camera and two planes. These two planes were to representative of a colour image and a depth image imported from the Xbox Kinect Sensor.

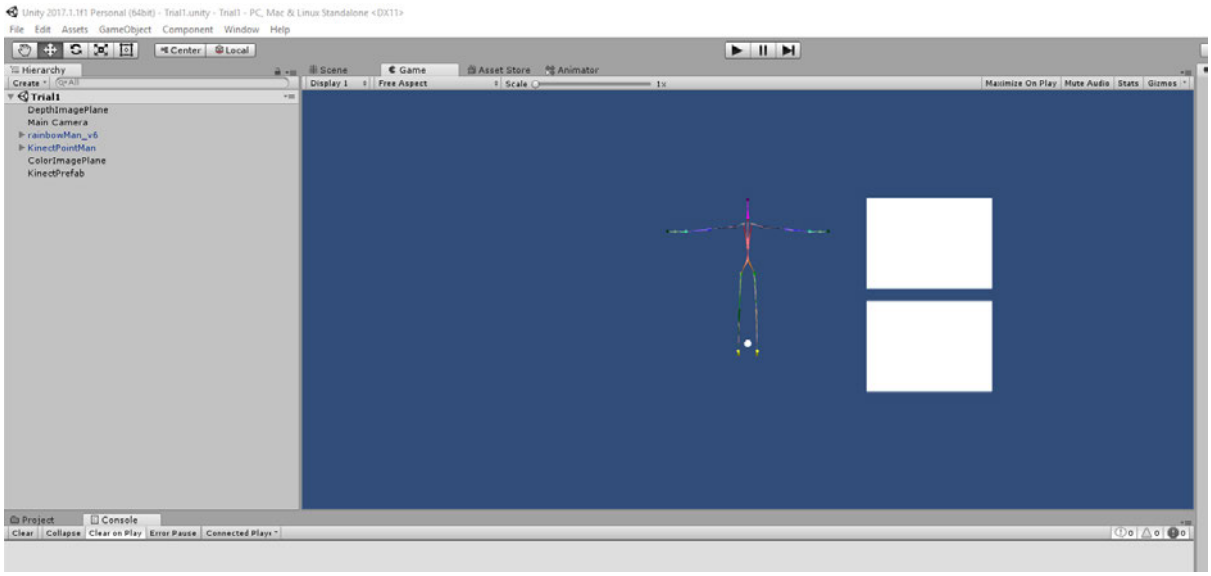


Figure 9: Trial 1 – Start up window

For this run the sample scenes loaded seamlessly into Unity and appeared fine, however, when trying to run the program there was no response from the Xbox Kinect Sensor. After several days of ensuring all required software was installed, installed correctly and compatible, there was still no response from

the sensor. To resolve the bug preventing the program from running all software was uninstalled and the reinstalled, this resolved the problem.

Upon resolving the issues with communication between the Xbox Kinect Sensor and Unity, the program ran as shown in figure 10. It can be seen that the program successfully locates a human and relates data points to the correct body parts, through the rainbow man skeleton, even when the full body of the user cannot be seen. Several bugs were stilled noted with the program, the major one being that Unity required to be closed and reopened to run the program every time it stopped as it could not reset itself. Another bug was that the program seemed to have limited running capacity, usually coming to a halt between 20 and 30 seconds of run time.

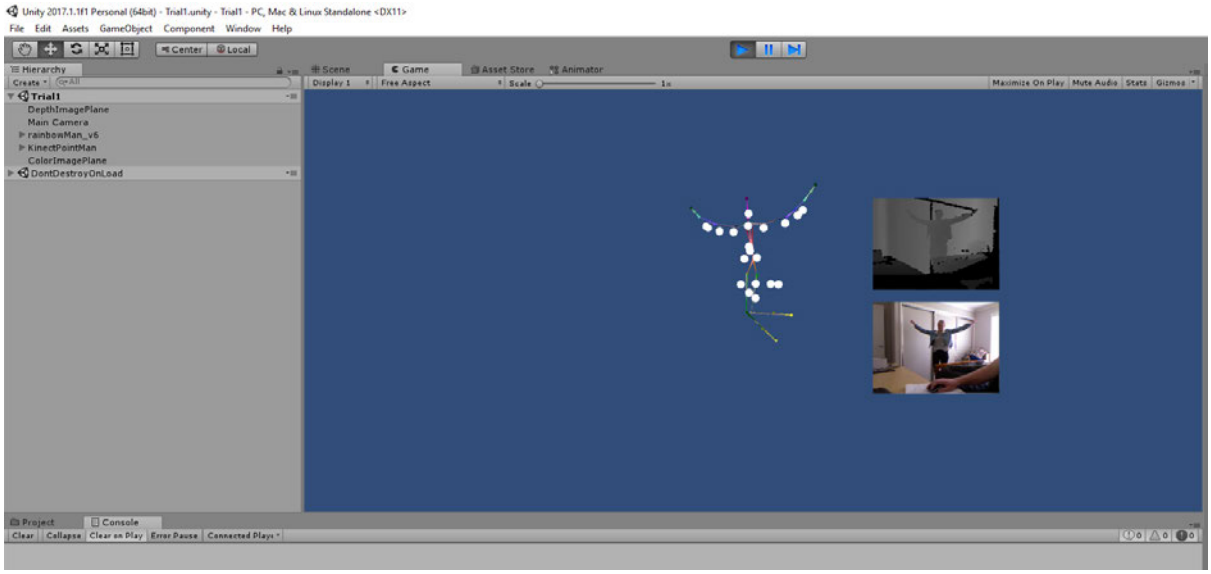


Figure 10: Trial 1 – Running Sample

Once the Xbox Kinect Sensor was communicating with the Unity game engine this trial was considered a success and saved as a working trial. Although there were still some bugs in the program that effected the success of operation, these were not considered detrimental at this stage of the project.

6.2 Trial 2

Trial two had the aim of focusing the depth map and manipulating the scene to be more specific to the project. While learning how to manipulate the scene was less intuitive than other programs, once it was understood how to do so, this was made relatively easy. The start-up window then appeared as shown in figure 11, with one major plane in the game view. This then had the depth image texture attached to examine the input.

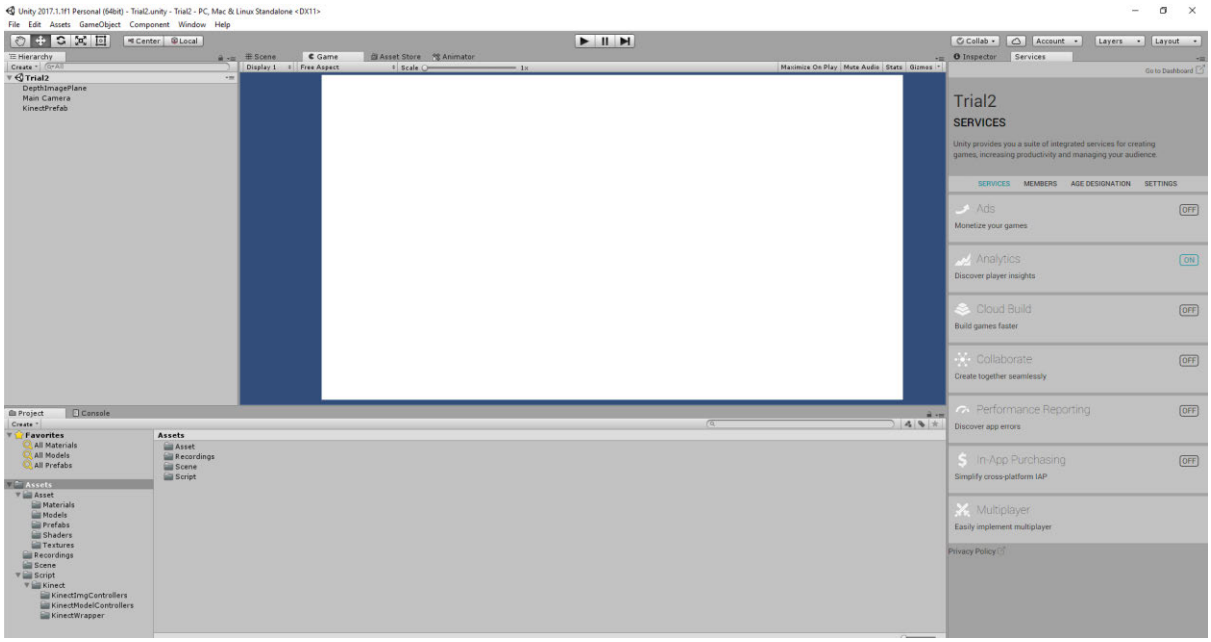


Figure 11: Trial 2 – Start up window

The code was then run as trial 2 to see that the connection between the Xbox Kinect sensor and Unity was still valid with the updated scene. Upon testing the trial was successful and provided images that would be as shown in figure 12.

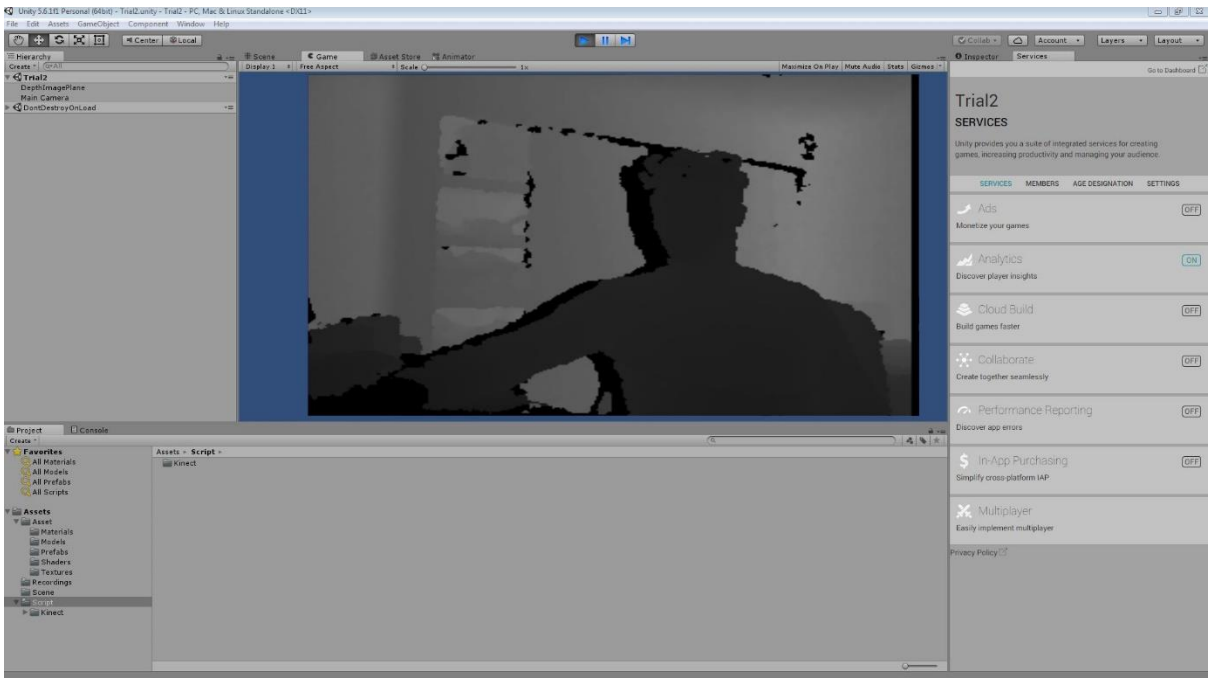


Figure 12: Trial 2 – Running sample

Examining the results found in this trial lead to interesting findings. While the shading of the depth image proved accurate for the most part, there were many errors found in image. The colour scale

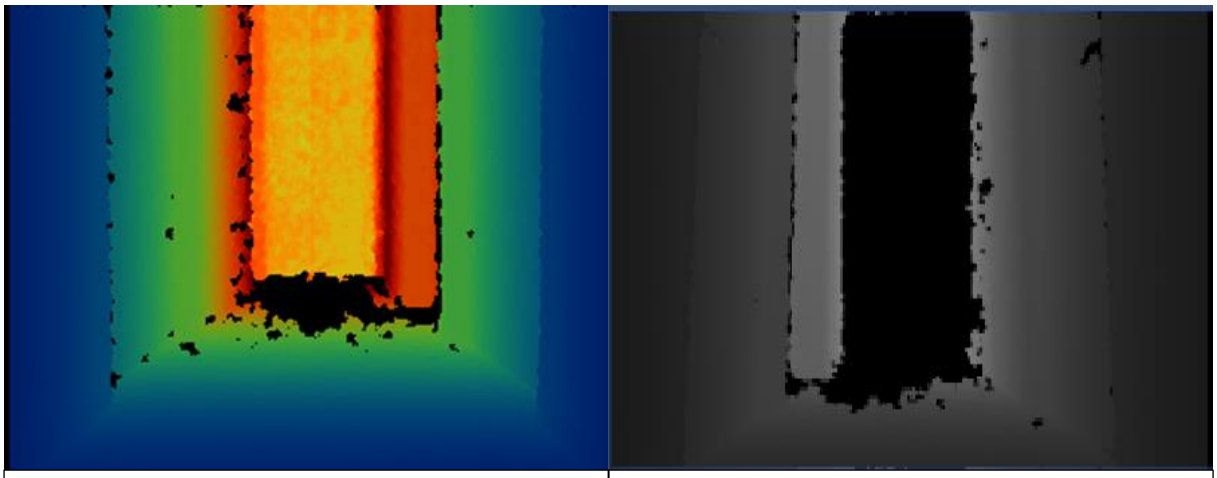
indicates how far away objects in the room are, dark colours being the closest and then gradually getting lighter as the objects become more distant. It is evident in figure 12 that there is a human figure in the foreground and a wall with two edges shown at the back of the image. While these figures are shown at their respected depths, there is significantly large areas shown are black, indicating close obstacles, throughout the image in unexpected places. These are most notable around the human figure and almost appear as a shadow. Then there are also errors shown in the back wall where the most distant objects appear to have some areas highlights as being close obstacles.

These errors were considered significant and explored further in this trial by keeping the trial code the same, but comparing the Xbox Kinect sensor data through another method. The selected method to be used was the Kinect Studio v1.8.0

Comparison through a separate method would indicate whether these errors are Hardware errors or software errors adopted from the source code when transferring the data into the Unity game engine. A hallway, as shown in figure 13, was utilised for this comparison as the length was considered great enough for the purpose of the project and had the walls as a potential obstacle, all the way through to the door, which was 5 metres away from where the Xbox Kinect Sensor was placed.

Figure 13: Kinect Studio – Hallway Colour Image

Acknowledging the colour image as a reference for the environment, this can then be compared to the two methods of depth mapping. Both depth images are shown in figure 14. For the comparison, it is noted that the Unity image is oriented correctly while the Kinect Studio Depth and Colour images show a mirrored image of the reality. The Unity depth image again shows a greyscale spectrum with darker colours representing close objects and light colours indicating a distant object. The Kinect studio depth image uses a colour spectrum with blue indicating close objects and red indicating the most distant objects.



Kinect Studio Depth Image	Unity Depth Image

Figure 14: Kinect Studio and Unity – Hallway Depth Image

The Kinect Studio depth image shown in figure 14, presents some errors and a significant amount of noise in the space of the blacked-out areas. The edges of obstacles appear to have misreading's supporting the findings in the first running sample of trial two. This leads to a hypothesis that the hardware may have some difficulties identifying distances of edges and corners. This however does not explain the phenomena found in figure 14, where large areas in free space showed similar errors. The noise shown at the base of the door may be accounted to the reflection of the tiles not providing an accurate reading.

When inputting the data to Unity, all errors noted in the Kinect Studio depth image translated across. The addition of the major dark spot of the door caused concern with errors as the most distant object in the environment, the door, was appearing as the closest, where no reflections or edges could be noted as a cause.

Rather than explore methods to reduce these errors which would be timely and of little benefit, from this trial it was devised that objects shown closer than 0.2 metres would not need to be considered as

they are likely to be errors. Also, it was devised that the environment will not be considered greater than 5 metres away from the sensor as the errors compounded in Unity at greater distances.

6.3 Trial 3

Listed in Chapter 4, section 4.1 was a list of codes used for trials 1 and 2. Some of these codes were made redundant with the elimination of the rainbow man as the human user tracker, when the prefabs were removed from the scene in trial 2. Trial 3 aimed to delete unnecessary codes and modify other where applicable, to reduce the program while still presenting the depth map.

Examining which codes were necessary and which were not, lead to reducing 11 scripts to 7 to have a functional program. Many of the remaining scripts could also be modified to remove some relation to the rainbow man skeleton. Some scripts were required to keep the rainbow man skeleton references as the prevented the code from flagging “array index is out of range”. This error seemed to appear with no cause, sometimes showing sometimes not, when identical codes were used. Reduction of the code to include no skeleton code was possible, but made this error more prominent and was therefore not used.

Initially when running the trial some warnings flagged in the console, these warnings can be seen in figure 15. These errors were not detrimental to the program and had no effect on operation. They did however point out lines that could be altered in certain scripts.

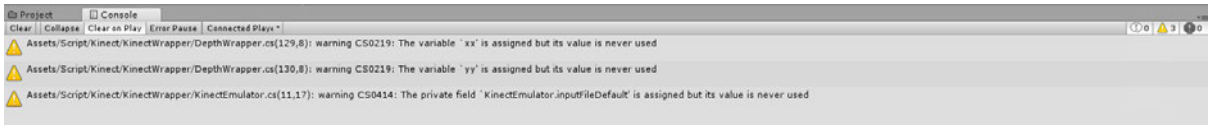


Figure 15: Trial 3 – Initial Code Errors

Further coding adaptations were made until warnings no longer flagged in the console when the program was run.

An unexpected, but beneficial finding was presented in this trial. Upon reducing the number of scripts and their contents, it was found that some of the bugs of the program noted in trial one were eliminated. The program could be run without Unity needing to be restarted after every test, the program could be paused and restarted at any time and could be run for an unlimited duration without stopping on its own accord.

6.4 Trial 4

Trial 4 was the beginning of testing a code as per the stages discussed in section 5.2. For comparison the depth map from the Xbox Kinect sensor would remain and a secondary plane was created to have a new code attached as its texture. This texture is the top down view to be presented to the user for potential navigation of a mobile robot. The layout for trial 4 can be seen in figure 16.

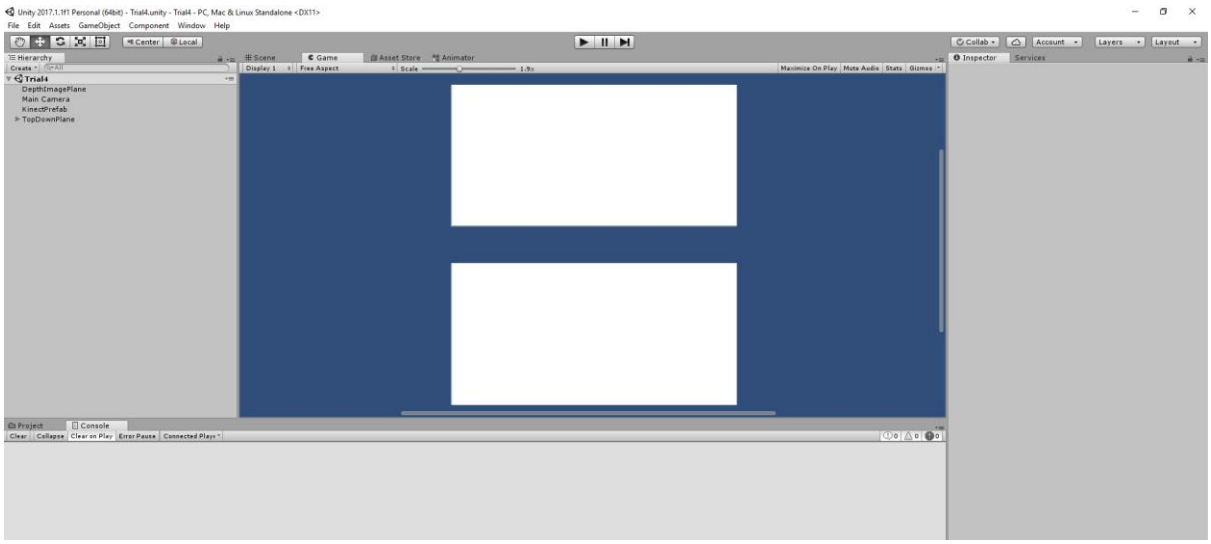


Figure 16: Trial 4 – Start up window

Commencing the code development began with isolating the depth image data. Through various methods it was attempted to highlight the depth data from a variable name attached to a plane, to a readable method for the user to understand. Print to console through the “Debug.Log” command was the first method tried that either presented no data or overloaded the console causing the program not to run.

To combat the lack of success with the print to console method, a print to text file was used from various different sample codes. The text files printed to the correct locations, however they presented unreadable or blank when accessed. This is potentially due to the scripts clearing to no data at times when the program allowed the files to be opened. When attempting to see the results while the program was running the files were unable to be accessed due to already being in use.

Without the matrix data of the depth image being readable, the following processes of the code were difficult to begin. The progress from this trial, although unsuccessful, was saved as a progress point.

6.5 Trial 5

As the depth image data could not be obtained by the expected methods other techniques of collecting the data were briefly explored. These methods were transferring the depth image to a point cloud with a guided tutorial, or rewriting the scripts to be compatible with a three-dimensional shader technique as opposed to the two-dimensional texture on a plane. While these were both increasing the complexity of the data they were explored for one day each. No more time could be offered due to time constraints and the increased work to return the data from that form to the intended project depth map form.

Once the given time for these methods being expended, no success had been found in either method. As the time constraints were still limited other segments of the main code were explored without specific relation to what the data may be from the Unity depth image. These segments are discussed, with details on what was unable to be completed and needs to be addressed for future work, in section 7.3.1.

Chapter 7

Conclusion

This chapter will discuss and conclude the results found throughout the project. Furthermore, discussion will be conducted on how the findings can implemented into real world situations or built upon for further research.

7.1 Key Findings

While there are areas that can improve the completeness of the project, discussed in section 7.3, the findings of the research conducted within this dissertation have been successful. The findings have been aligned in research responses as follows that correspond to each point that was aimed to be covered in the project specification. The project specification can be found in appendix A.

7.1.1 Research Response 1

Explore current real-time gaming systems and methods of mobile robot control.

Through the exploration of systems and mobile robot control it was found that there is extensive information on both real-time gaming systems and methods of mobile robot control, so much so that it could not all be covered within the scope of the project. Methods used are incredibly advanced and would require years of specialising in either real-time gaming or mobile robot control to grasp fully the potential that could be extended to mobile robots.

Although the information in existing literature is extensive it can be concluded that there is significant similarities and potential overlaps in the control methods of virtual and real world are shown to exist. This highlights the potential for strategy game techniques of mapping, control and decision making to be employed in the simplification of mobile robot control of both autonomous and remotely controlled methods.

7.1.2 Research Response 2

Implement a visual input into a game engine.

Through the aid of existing software, the integration between a visual input and a game engine were made. The Unity game engine was utilised for its ease of development along with the Xbox Kinect

sensor 1.0 as it is a well renowned robot sensor. With the success of joining the real world to a virtual world the potential of using gaming techniques for mobile robot control was confirmed. For simplicity and the real world aim the implementation was kept of a low level, in applications with greater network capabilities, point cloud and three-dimensional mapping techniques can be employed for more complex tasks.

7.1.3 Research Response 3

Optimise the input to be a real-time system that can be interpreted by an operator.

Optimisation of the project to be in real to was made simple by the level of the technology used. From the initial trial all the way through to the final trial there was no discernible difference in response of the program and the real-life action. This met the basic criteria of a real-time system.

While the full code development was not successful, the game engine provided a workable format, with the Xbox Kinect Sensor data, from the initial integration onwards. The research that was completed concludes that real-time gaming methods are suitable for mobile robots. Further investigation will prove if a data transfer rate of less than 1MB/s is achievable as a method of navigation and other limitations of use.

7.1.4 Research Response 4

Test the usability of the system with a range of operators.

Due to the time constraints of the project and the limited ability to code the program, this part of the research could not be completed. While this could not provide results for the research the positive findings of the rest of the project demand more research in this area.

7.2 Significance of Findings

The findings of the project were not as clean as expected, however, they still contain a level of significance. Highlighted from the dissertation is that little research currently exists in this area. Given this it is found that the ability to translate a real-world environment into a virtual world where strategy game algorithms and methods can be applied to a mobile robot is possible.

This is of great significance as robots of all forms are becoming not only more common, but a necessity for the existence of metropolitan areas, through transportation and other luxuries, but most specifically food processing. With utilisation of gaming strategies being expanded to robots, this increases the software development budget by billions for robots by allowing the new and innovative solutions for games and simulations to be utilised for real world purposes.

Throughout the exploration of this topic it is has become evident how extensive this field of research is. Various findings of previous literature and project specific work have been provided in this dissertation. From this, areas of further research and potentials have been uncovered with many considerations and possible benefits.

7.3 Further Work and Recommendations

The aforementioned significant findings indicate that there is more work to be done on this topic to find conclusive information and make it more applicable to real world scenarios. Further work includes, integration of the game engine to a mobile robot, expanding the field of view and memory mapping. These will be discussed further throughout section 7.3.

7.3.1 Program Finalisation

Unfortunately, while the project was somewhat successful, not all targets were fully met. The main program that was proposed could not be completed successfully, which has limited some findings of the project. The first future target should be to resolve the issues of this program that has been discussed extensively throughout the dissertation.

It is noted that the main section of the is incomplete due to the depth isolation being unsuccessful. Because of this the code could not be tested but is believed to provide a basis for future work to be completed.

The areas that need to be attended to are the depth isolation, floor removal and obstacles scoping segments of the code, as these are considered most simple when the data to be manipulated is known. Revision of the variables may be necessary to ensure the code can flow through. For the provided code it is all necessary that the top down map consider the angle for each array column as it is currently expected to be shown as a rectangular form and not as the angled map that is desired.

7.3.2 Mobile Robot Integration

While the possibility of using a game engine as a platform for mobile robots was explored in this project, methods of controlling the robot were only looked at throughout existing literature and not through new research. The end goal would be to have a mobile robot fully controlled by the inputs from a game engine in which the robot provides the environmental feedback as per the methods proposed by the completed project work that is discussed in this dissertation.

To achieve this, it is recommended a wireless network be set up with a PC or microprocessor with adequate capabilities to run the main program be mounted to the robot. Initially a Wi-Fi network would be appropriate to communicate to a secondary machine which would have the control inputs of the “game” and utilise the network to receive the map generated from the main program to be

presented to the user. Additional coding will be required to allow for the second machine to interpret the data and allow for input to the game control.

The work done by Ohashi et al. is of great benefit for understanding in this future work. Their works involve the development of control of a mobile robot (Roomba iRobot vacuum cleaner) through the creation of a virtual world with in Unity. Considerations that were highlighted to be involved in future works include acknowledging and attending to the error of having two states. In their study, the robot had a real world and a virtual world state. The real-world state had errors such as wheel slip and motor lag whereas the virtual world robot consisted of an idealistic state where robot response was immediate and free from error. This became an amplified problem on larger trip where the differences in states would become far more significant.

Taking into consideration the findings of Ohashi et al. the study still proves to be a great starting point to develop methods of control for a mobile robot through a game engine. Errors that were found in the Ohashi et al. study are likely to be encountered with the use of environmental feedback methods presented in this project until memory mapping is considered, which will be discussed further in this chapter.

7.3.3 Field of View Expansion

It was found that the field of view when utilising the Xbox Kinect version 1.0 sensor was quite narrow at only 58.5 degrees wide. This was not considered enough to accurately interpret the best possible navigation path. Reiterating the deficiency in field of view, the single Xbox Kinect 1.0 allowed for a minimum turn radius and left 121.5 degrees in the direction of travel unaccounted for. In order to improve this several methods were devised to improve the field of view. These methods are as follows:

7.3.3.1 Xbox Kinect Sensor Upgrade

Upgrading of the visual input sensor to the Xbox Kinect sensor 2.0. The Xbox Kinect sensor 1.0 or version one that was utilised for this project has been superseded by Microsoft and their 'Xbox One' gaming console. The newer version the Xbox Kinect sensor 2.0 has updated harder and software. The greatest benefits that would be seen if implemented in this project are the 640 x 480 pixel depth image (up from 320 x 240 pixels in the version one) still at the frame rate of 30 frames per second. This gives the updated sensor a field of view of 70.6 x 60 degrees, which is improved by 12.1 degrees horizontally and 13.4 degrees in the vertical direction.

7.3.3.2 Additional Xbox Kinect Sensors

A simple way to increase the field of view, of the robot's environment, that the user is presented is to employ a second or potentially even a third Xbox Kinect sensor 1.0. This

method will potentially add 55 degrees (accounting for a minor overlap) to the field view per Kinect sensor added to the array. Limitations and issues with this method include:

1. Non-compatibility with the newer Xbox Kinect sensor 2.0 due to SDK driver limitations.
2. The need for a third-party program to interpret the data from multiple sensors and the relay this data to unity.
3. Majority of the code will require reworking to account for the data form the Kinect sensor being presented from a third-party program and not directly into Unity.
4. Additional sensors transmitting at the same time will increase the required bandwidth to report the environment to the user.

7.3.3.3 Pivoting Xbox Kinect Sensor

As the title of this sections suggests, this method of expanding the field of view is through implementing a swinging motion on the sensor in the lateral direction. This will swing the sensor back and forth with respected to the position of the mobile robot so that the environment in the direction of travel can be fully sighted in one swing. This will continue swinging to update the field of view as the mobile robot progresses. It is recommended that the sensor cover at least 180 degrees (requiring a 121.5 degree swing) which means the sensor is seeing approximately 32.5% of the full field of view at a time. With only this small range of the full field of view being captured at any one moment, this will require a memory of the environment map being stored so the user does not have to recall where the obstacles are from their own memory. This identifies a draw-back to this method as the memory mapping concept, later proposed in section 7.4.3, is potentially complex.

From the three suggested methods of field of view expansion, two different configurations can be recommended due to limitations in hardware and software integration.

The first recommendation is the upgrade to the Xbox Kinect sensor 2.0, in conjunction with a sensor pivoting system. This will allow for the increased field of view being captured by the version two Kinect sensor that will be further amplified and limiting blind spots through the implementation of the pivoting system. Draw-backs of this method are as discussed in section 7.3.2.3.

The second recommendation is to utilise the current Xbox Kinect sensor 1.0 with a second or possibly a third version 1.0 sensor to increase the field of view. A slight binocular vision type would be required between the sensor sets to ensure there is limited blind spots, however monocular vision should be maximised to give the best field of view results. Issues in subsection 7.3.2.2 must be considered when

exploring this potential method of increasing the field of view. The benefit from this is that the field of view will be increased without the potentially complex memory mapping discussed in section 7.3.4.

7.3.4 GPS and Mapping Overlay

The resulting depth map and obstacle locations highlighted from this project give a path of possible navigation for the robot's immediate environment. To increase the real-world potential of this project with mobile robot navigation, it is recommended that a GPS is employed. The addition of a GPS will add to the position of the robot on a global scale, making final targets and navigation way points easier to plan and achieve as opposed to the current system where the robot would seemingly only find possible free paths of navigation.

With the employment of a GPS a location could be found on an existing mapping service such as Google Maps. To overlay the map of immediate environmental obstacles developed in this project over a map such as Google Maps will have increasing benefits to the user for their situational awareness and target locations that they are trying to achieve. The mapping overlay is also likely to increase the reliability of the system by having two methods of mapping reassuring the user of the best path. If implemented only as a control benefit for the user the mapping overlay will not increase the required network bandwidth for user to robot communication.

7.3.5 Memory Mapping

The final recommended method of further project development, memory mapping, is as the name suggests and requires the program to memorise the map that is presented to the user rather than just showing a live image of the immediate environment. The environment will need to be learnt on the user end of the system to limit the amount of data that will need to be transferred.

Benefits of exploring and successfully completing memory mapping include:

1. Recording major obstacles found to simplify possible return path selection.
2. Increased situational awareness for the user.
3. The possibility of upgrading to the Xbox Kinect sensor 2.0 and implementing a pivoting system as mentions in section 7.3.2.

7.3.6 Real World Application

To validate the project in its entirety, a final test and evaluation must be undertaken upon the completion of all further work. While the success of mobile robot applications has been explored and evaluated in this project, the previously mentioned topics in section 7.3 provide completeness to the project and warrant a greater evaluation possible.

Chapter 8

Reference List

Arabnia, H., & Deligiannidis, L. (2015). *Emerging trends in image processing, computer vision, and pattern recognition*. Waltham, MA: Morgan Kaufmann is an imprint of Elsevier.

Bartneck, C., Soucy, M., Fleuret, K., & Sandoval, E. B. (2015). *The robot engine - Making the unity 3D game engine work for HRI*.

Cook. *Mobile Robots: Navigation, Control and Remote Sensing*: John Wiley & Sons Incorporated.

Davies, E. R., & ebrary, I. (2012). *Computer and machine vision: theory, algorithms, practicalities* (4th ed.). Waltham, Mass.

Ghani, M. F. A., & Sahari, K. S. M. (2017). Detecting negative obstacle using Kinect sensor. *International Journal of Advanced Robotic Systems.*

Hansard, M. (2013). *Time-of-flight cameras: principles, methods and applications*. London: Springer.

Hsu, Y.-J., Lin, C.-H., & Shih, J.-L. (2013). *Developing Multi-player Digital Adventure Education Game with Motion Sensing Technologies*.

Huat, CnS & Teo, J 'Online evolution of offensive strategies in real-time strategy gaming', in pp. 1-8.

Isafiade, O., Osunmakinde, I., & Bagula, A. (2013). A Complementary Vision Strategy for Autonomous Robots in Underground Terrains using SRM and Entropy Models. *International Journal of Advanced Robotic Systems.*

Khoshelham, K., & Elberink, S. O. (2012). Accuracy and resolution of Kinect depth data for indoor mapping applications. *Sensors (Basel, Switzerland).*

Kirriemuir, J 2000, 'The console market', Virtual Reality, vol. 5, no. 4, pp. 236-44.

Lin, J., Sun, Q., Li, G., & He, Y. (2014). SnapBlocks: a snapping interface for assembling toy blocks with XBOX Kinect. *Multimedia Tools and Applications.*

Mather, P. M., & Koch, M. (2011). *Computer processing of remotely-sensed images: an introduction* (4th ed.). Hoboken, NJ;Chichester, West Sussex, UK;: Wiley-Blackwell.

Microsoft 2017, XBOX ONE X, Microsoft, <https://www.xbox.com/en-AU/xbox-one-x?xr=shellnav>.

Ohashi, O., Ochiai, E., & Kato, Y. (2014). *A Remote Control Method for Mobile Robots Using Game Engines*.

Parker, J. R., & ebrary, I. (2011). Algorithms for image processing and computer vision (2nd ed.). Indianapolis, Ind: Wiley Publishing, Inc.

Sonka, M., Hlavac, V., & Boyle, R. (1993). *Image processing, analysis, and machine vision* (1st ed.). New York;London;: Chapman & Hall Computing.

Sosa, A., Stanton, R., Perez, S., Keyes-Garcia, C., Gonzalez, S., & Toups, Z. (2015). *Imperfect Robot Control in a Mixed Reality Game to Teach Hybrid Human-Robot Team Coordination*.

Tzafestas, S. G. (2014). *Introduction to mobile robot control* (First ed.). Waltham, MA: Elsevier.

Webb, J., Ashley, J., Dawson, S., & Books24x. (2012). Beginning Kinect programming with the Microsoft Kinect SDK. New York.

Chapter 9

Appendices

Appendix A: Project Specification

ENG4111/4112 Research Project

Project Specification

For: Joshua Muhldorff

Title: Real-time strategy game control for Mobile Robots

Major: Mechatronic Engineering

Supervisor: Tobias Low

Enrolment: ENG4111 – ONC S1, 2017
ENG4112 – ONC S2, 2017

Project Aim: To explore the applications of real-time game strategies as a method to control mobile robots in unfamiliar environments.

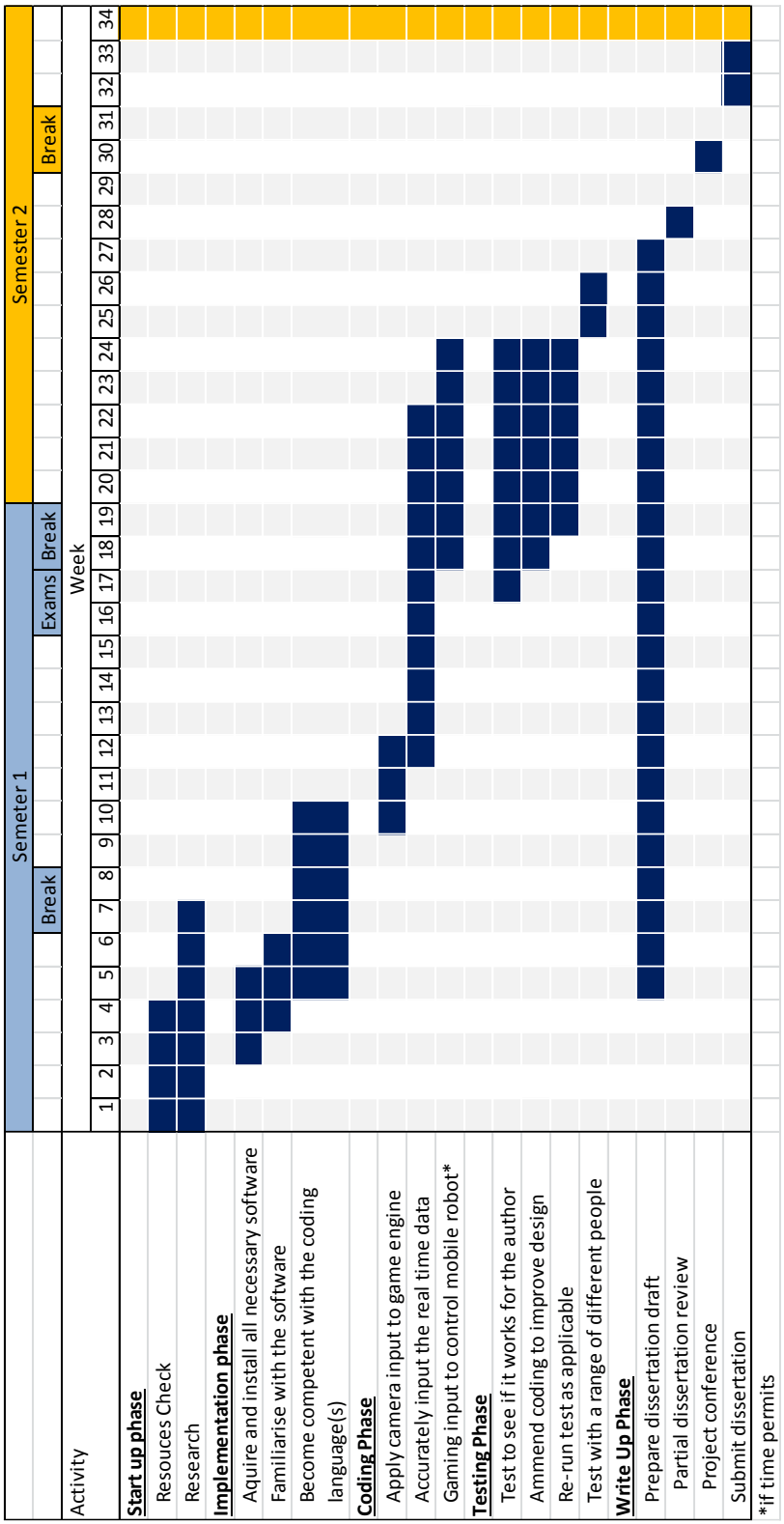
Programme: Issue A, 15th March 2017

1. Explore current real-time gaming systems and methods of mobile robot control
2. Implement a visual input into a game engine
3. Optimise the input to be a real-time system that can be interpreted by an operator
4. Test the usability of the system with a range of operators

If time permits:

5. Apply the system to an existing mobile robot platform
6. Test the usability of the mobile robot system with a range of operators

Appendix B: Project Timeline



Appendix C: Program Code

It is acknowledged that the coding presented in Appendix C (with the exception of the main code) is formed with some minor alterations from existing scripts, found from:

http://wiki.etc.cmu.edu/unity3d/index.php/Microsoft_Kinect_-_Microsoft_SDK

Display Depth

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Renderer))]
public class DisplayDepth : MonoBehaviour {

	public DepthWrapper dw;

	private Texture2D tex;
	// Use this for initialization
	void Start () {
		tex = new Texture2D(640,480,TextureFormat.ARGB32,false);
		GetComponent<Renderer>().material.mainTexture = tex;
	}

	// Update is called once per frame
	void Update () {
		if (dw.pollDepth())
		{
			tex.SetPixels32(convertDepthToColor(dw.depthImg));
			//tex.SetPixels32(convertPlayersToCutout(dw.segmentations));
			tex.Apply(false);
		}
	}

	private Color32[] convertDepthToColor(short[] depthBuf)
	{
		Color32[] img = new Color32[depthBuf.Length];
		for (int pix = 0; pix < depthBuf.Length; pix++)
		{
			img[pix].r = (byte)(depthBuf[pix] / 32);
			img[pix].g = (byte)(depthBuf[pix] / 32);
			img[pix].b = (byte)(depthBuf[pix] / 32);
		}
		return img;
	}

	private Color32[] convertPlayersToCutout(bool[,] players)
	{
		Color32[] img = new Color32[640*480];
		for (int pix = 0; pix < 640*480; pix++)
		{

	if(players[0,pix]||players[1,pix]||players[2,pix]||players[3,pix]||players[4,pix]||p
layers[5,pix])
			{
				img[pix].a = (byte)255;
			} else {
				img[pix].a = (byte)0;
			}
		}
```

```
                return img;

        }
}
```

Depth Wrapper

```
using UnityEngine;
using System;
using System.Collections;

/// <summary>
/// Level of indirection for the depth image,
/// provides:
/// -a frames of depth image (no player information),
/// -an array representing which players are detected,
/// -a segmentation image for each player,
/// -bounds for the segmentation of each player.
/// </summary>
public class DepthWrapper: MonoBehaviour {

	public DeviceOrEmulator devOrEmu;
	private Kinect.KinectInterface kinect;

	private struct frameData
	{
		public short[] depthImg;
		public bool[] players;
		public bool[,] segmentation;
		public int[,] bounds;
	}

	public int storedFrames = 1;

	private bool updatedSeqmentation = false;
	private bool newSeqmentation = false;

	private Queue frameQueue;

	/// <summary>
	/// Depth image for the latest frame
	/// </summary>
	[HideInInspector]
	public short[] depthImg;
	/// <summary>
	/// players[i] true iff i has been detected in the frame
	/// </summary>
	[HideInInspector]
	public bool[] players;
	/// <summary>
	/// Array of segmentation images [player, pixel]
	/// </summary>
	[HideInInspector]
	public bool[,] segmentations;
	/// <summary>
	/// Array of bounding boxes for each player (left, right, top, bottom)
	/// </summary>
	[HideInInspector]
	//right,left,up,down : but the image is fliped horizontally.
	public int[,] bounds;

	// Use this for initialization
	void Start () {
```

```
                kinect = devOrEmu.getKinect();
                //allocate space to store the data of storedFrames frames.
                frameQueue = new Queue(storedFrames);
                for(int ii = 0; ii < storedFrames; ii++){
                        frameData frame = new frameData();
                        frame.depthImg = new short[640 * 480];
                        frame.players = new bool[Kinect.Constants.NuiSkeletonCount];
                        frame.segmentation = new
bool[Kinect.Constants.NuiSkeletonCount,640*480];
                        frame.bounds = new int[Kinect.Constants.NuiSkeletonCount,4];
                        frameQueue.Enqueue(frame);
                }
        }

        // Update is called once per frame
        void Update () {

        }

        void LateUpdate()
        {
                updatedSeqmentation = false;
                newSeqmentation = false;
        }
        /// <summary>
        /// First call per frame checks if there is a new depth image and updates,
        /// returns true if there is new data
        /// Subsequent calls do nothing have the same return as the first call.
        /// </summary>
        /// <returns>
        /// A <see cref="System.Boolean"/>
        /// </returns>
        public bool pollDepth()
        {
                //Debug.Log("" + updatedSeqmentation + " " + newSeqmentation);
                if (!updatedSeqmentation)
                {
                        updatedSeqmentation = true;
                        if (kinect.pollDepth())
                        {
                                newSeqmentation = true;
                                frameData frame = (frameData)frameQueue.Dequeue();
                                depthImg = frame.depthImg;
                                //players = frame.players;
                                segmentations = frame.segmentation;
                                bounds = frame.bounds;
                                frameQueue.Enqueue(frame);
                                processDepth();
                        }
                }
                return newSeqmentation;
        }

        private void processDepth()
        {
                for(int ii = 0; ii < 640 * 480; ii++)
                {
                        //get x and y coords
                        int xx = ii % 640;
                        int yy = ii / 640;
                        //extract the depth and player
                        depthImg[ii] = (short)(kinect.getDepth()[ii] >> 3);
```

```
                        int player = (kinect.getDepth()[ii] & 0x07) - 1;
                        if (player > 0)
                        {
                                if (!players[player])
                                {
                                        players[player] = true;
                                        segmentations[player,ii] = true;
                                        bounds[player,0] = xx;
                                        bounds[player,1] = xx;
                                        bounds[player,2] = yy;
                                        bounds[player,3] = yy;
                                }
                                else
                                {
                                        segmentations[player,ii] = true;
                                        bounds[player,0] = Mathf.Min(bounds[player,0],xx);
                                        bounds[player,1] = Mathf.Max(bounds[player,1],xx);
                                        bounds[player,2] = Mathf.Min(bounds[player,2],yy);
                                        bounds[player,3] = Mathf.Max(bounds[player,3],yy);
                                }
                        }
                }
        }
}
```

Device or Emulator

```
using UnityEngine;
using System.Collections;

public class DeviceOrEmulator : MonoBehaviour {

	public KinectSensor device;
	public KinectEmulator emulator;

	public bool useEmulator = false;

	// Use this for initialization
	void Start () {
		if(useEmulator){
			emulator.enabled = true;
		}
		else {
			device.enabled = true;
		}
	}

	// Update is called once per frame
	void Update () {

	}

	public Kinect.KinectInterface getKinect() {
		if(useEmulator){
			return emulator;
		}
		return device;
	}
}
```

Kinect Emulator

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using Kinect;

public class KinectEmulator : MonoBehaviour, KinectInterface {

	public string inputFile = "Assets/Kinect/Recordings/playback0";
	private string inputFileDefault = "Assets/Kinect/Recordings/playbackDefault";
	private float playbackSpeed = 0.0333f;
	private float timer = 0;
	private bool isDefault = true;

	/// <summary>
	/// how high (in meters) off the ground is the sensor
	/// </summary>
	public float sensorHeight;
	/// <summary>
	/// where (relative to the ground directly under the sensor) should the kinect
register as 0,0,0
	/// </summary>
	public Vector3 kinectCenter;
	/// <summary>
	/// what point (relative to kinectCenter) should the sensor look at
	/// </summary>
	public Vector4 lookAt;

	/// <summary>
	///variables used for updating and accessing depth data
	/// </summary>
	private bool newSkeleton = false;
	private int curFrame = 0;
	private NuiSkeletonFrame[] skeletonFrame;
	/// <summary>

	// Use this for initialization
	void Start () {
		LoadPlaybackFile(inputFile);
	}

	void Update () {
		timer += Time.deltaTime;
		if(Input.GetKeyUp(KeyCode.F12)) {
			if(isDefault) {
				isDefault = false;
				LoadPlaybackFile(inputFile);
			}
			else {
				isDefault = true;
				LoadPlaybackFile(inputFile);
			}
		}
	}

	// Update is called once per frame
	void LateUpdate () {
		newSkeleton = false;
```

```
	}

	void LoadPlaybackFile(string filePath)  {
		FileStream input = new FileStream(@filePath, FileMode.Open);
		BinaryFormatter bf = new BinaryFormatter();
		SerialSkeletonFrame[] serialSkeleton =
(SerialSkeletonFrame[])bf.Deserialize(input);
		skeletonFrame = new NuiSkeletonFrame[serialSkeleton.Length];
		for(int ii = 0; ii < serialSkeleton.Length; ii++){
			skeletonFrame[ii] = serialSkeleton[ii].deserialize();
		}
		input.Close();
		timer = 0;
		Debug.Log("Simulating "+@filePath);
	}

	float KinectInterface.getSensorHeight() {
		return sensorHeight;
	}
	Vector3 KinectInterface.getKinectCenter() {
		return kinectCenter;
	}
	Vector4 KinectInterface.getLookAt() {
		return lookAt;
	}

	bool KinectInterface.pollSkeleton() {
		int frame = Mathf.FloorToInt(Time.realtimeSinceStartup / playbackSpeed);
		if(frame > curFrame){
			curFrame = frame;
			newSkeleton = true;
			return newSkeleton;
		}
		return newSkeleton;
	}

	NuiSkeletonFrame KinectInterface.getSkeleton() {
		return skeletonFrame[curFrame % skeletonFrame.Length];
	}
	/*
	NuiSkeletonBoneOrientation[]
KinectInterface.getBoneOrientations(NuiSkeletonFrame skeleton){
		return null;
	}
	*/
	NuiSkeletonBoneOrientation[]
KinectInterface.getBoneOrientations(NuiSkeletonData skeletonData){
		NuiSkeletonBoneOrientation[] boneOrientations = new
NuiSkeletonBoneOrientation[(int)(NuiSkeletonPositionIndex.Count)];
		NativeMethods.NuiSkeletonCalculateBoneOrientations(ref skeletonData,
boneOrientations);
		return boneOrientations;
	}

	bool KinectInterface.pollColor() {
		return false;
	}

	Color32[] KinectInterface.getColor() {
		return null;
	}
```

```
	bool KinectInterface.pollDepth() {
		return false;
	}

	short[] KinectInterface.getDepth() {
		return null;
	}
}
```

Kinect Interop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.CompilerServices;
using UnityEngine;

namespace Kinect {


	public struct NuiStatusProc
	{

	}

	public interface KinectInterface {

		float getSensorHeight();
		Vector3 getKinectCenter();
		Vector4 getLookAt();

		bool pollSkeleton();
		NuiSkeletonFrame getSkeleton();

		// Get all bone orientation based on skeletondata
		// kinect Sdk 1.7.0, June 2013, Jason Added
		NuiSkeletonBoneOrientation[] getBoneOrientations(NuiSkeletonData
skeletonData);

		bool pollColor();
		Color32[] getColor();
		bool pollDepth();
		short[] getDepth();
	}

	public static class Constants
	{
		public static int NuiSkeletonCount = 6;
	public static int NuiSkeletonMaxTracked = 2;
	public static int NuiSkeletonInvalidTrackingID = 0;

		public static float NuiDepthHorizontalFOV = 58.5f;
		public static float NuiDepthVerticalFOV = 45.6f;
	}

	/// <summary>
	///Structs and constants for interfacing c# with the c++ kinect dll
	/// </summary>

    [Flags]
    public enum NuiInitializeFlags : uint
    {
        UsesDepthAndPlayerIndex = 0x00000001,
        UsesColor = 0x00000002,
        UsesSkeleton = 0x00000008,
        UsesDepth = 0x00000020
    }
```

```
    public enum NuiSkeletonPositionIndex : int
    {
        HipCenter = 0,
        Spine = 1,
        ShoulderCenter = 2,
        Head = 3,
        ShoulderLeft = 4,
        ElbowLeft = 5,
        WristLeft = 6,
        HandLeft = 7,
        ShoulderRight = 8,
        ElbowRight = 9,
        WristRight = 10,
        HandRight = 11,
        HipLeft = 12,
        KneeLeft = 13,
        AnkleLeft = 14,
        FootLeft = 15,
        HipRight = 16,
        KneeRight = 17,
        AnkleRight = 18,
        FootRight = 19,
        Count = 20
    }

    public enum NuiSkeletonPositionTrackingState
    {
        NotTracked = 0,
        Inferred,
        Tracked
    }

    public enum NuiSkeletonTrackingState
    {
        NotTracked = 0,
        PositionOnly,
        SkeletonTracked
    }

       public enum NuiImageType
       {

       DepthAndPlayerIndex = 0,    // USHORT
              Color,                                    // RGB32 data
              ColorYUV,                                 // YUY2 stream from camera h/w,
but converted to RGB32 before user getting it.
              ColorRawYUV,                        // YUY2 stream from camera h/w.
              Depth                                     // USHORT
       }

       public enum NuiImageResolution
       {
              resolutionInvalid = -1,
              resolution80x60 = 0,
              resolution320x240,
              resolution640x480,
              resolution1280x1024                       // for hires color only
       }

       //KK addition - flags for image stream in order to enable near mode
       public enum NuiImageStreamFlags
       {
```

```
            None = 0x00000000,
            SupressNoFrameData = 0x0001000,
            EnableNearMode = 0x00020000,
            TooFarIsNonZero = 0x0004000
        }
        // end of KK addition

    [StructLayoutAttribute(LayoutKind.Sequential)]
    public struct NuiSkeletonData
    {
        public NuiSkeletonTrackingState eTrackingState;
        public uint dwTrackingID;
        public uint dwEnrollmentIndex_NotUsed;
        public uint dwUserIndex;
        public Vector4 Position;
        [MarshalAsAttribute(UnmanagedType.ByValArray, SizeConst = 20, ArraySubType =
UnmanagedType.Struct)]
        public Vector4[] SkeletonPositions;
        [MarshalAsAttribute(UnmanagedType.ByValArray, SizeConst = 20, ArraySubType =
UnmanagedType.Struct)]
        public NuiSkeletonPositionTrackingState[] eSkeletonPositionTrackingState;
        public uint dwQualityFlags;
    }

    public struct NuiSkeletonFrame
    {
        public Int64 liTimeStamp;
        public uint dwFrameNumber;
        public uint dwFlags;
        public Vector4 vFloorClipPlane;
        public Vector4 vNormalToGravity;
        [MarshalAsAttribute(UnmanagedType.ByValArray, SizeConst = 6, ArraySubType =
UnmanagedType.Struct)]
        public NuiSkeletonData[] SkeletonData;
    }

       public struct NuiTransformSmoothParameters
       {
            public float fSmoothing;
            public float fCorrection;
            public float fPrediction;
            public float fJitterRadius;
            public float fMaxDeviationRadius;
       }

       // flag for skeleton tracking mode
       // kinect Sdk 1.7.0, June 2013, Jason Added
       public enum NuiSkeletonFlags
       {
            SUPPRESS_NO_FRAME_DATA = 0x00000001,
            TITLE_SETS_TRACKED_SKELETONS = 0x00000002,
            ENABLE_SEATED_SUPPORT = 0x00000004,
            ENABLE_IN_NEAR_RANGE = 0x00000008
       }

       // the struct based on http://msdn.microsoft.com/en-
us/library/nuiskeleton.nui_skeleton_bone_orientation.aspx
       // kinect Sdk 1.7.0, June 2013, Jason Added
       public struct NuiSkeletonBoneOrientation
       {
            public NuiSkeletonPositionIndex endJoint;
            public NuiSkeletonPositionIndex startJoint;
```

```
		public NuiSkeletonBoneRotation hierarchicalRotation; // local
orientation
		public NuiSkeletonBoneRotation absoluteRotation; // world orientation
	}

	// the struct based on http://msdn.microsoft.com/en-
us/library/nuiskeleton.nui_skeleton_bone_rotation.aspx
	// kinect Sdk 1.7.0, June 2013, Jason Added
	public struct NuiSkeletonBoneRotation {
		public SerialMatrix4 rotationMatrix;
		public SerialVec4 rotationQuaternion;
	}

	// used in NuiSkeletonBoneRotation
	// kinect Sdk 1.7.0, June 2013, Jason Added
	[Serializable]
	public struct SerialVec4 {
		float x,y,z,w;

		public SerialVec4(Vector4 vec){
			this.x = vec.x;
			this.y = vec.y;
			this.z = vec.z;
			this.w = vec.w;
		}
		public Vector4 deserialize() {
			return new Vector4(x,y,z,w);
		}

		public Quaternion GetQuaternion() {
			return new Quaternion(x,y,z,w);
		}
	}

	// used in NuiSkeletonBoneRotation
	// kinect Sdk 1.7.0, June 2013, Jason Added
	[Serializable]
	public struct SerialMatrix4 {
		float m11;
		float m12;
		float m13;
		float m14;
		float m21;
		float m22;
		float m23;
		float m24;
		float m31;
		float m32;
		float m33;
		float m34;
		float m41;
		float m42;
		float m43;
		float m44;

		public SerialMatrix4(Matrix4x4 mat){
			this.m11 = mat.m00;
			this.m12 = mat.m01;
			this.m13 = mat.m02;
			this.m14 = mat.m03;
			this.m21 = mat.m10;
			this.m22 = mat.m11;
```

```
                this.m23 = mat.m12;
                this.m24 = mat.m13;
                this.m31 = mat.m20;
                this.m32 = mat.m21;
                this.m33 = mat.m22;
                this.m34 = mat.m23;
                this.m41 = mat.m30;
                this.m42 = mat.m31;
                this.m43 = mat.m32;
                this.m44 = mat.m33;
        }

        public Matrix4x4 deserialize() {
                Matrix4x4 mat = new Matrix4x4();

                mat.m00 = m11;
                mat.m01 = m12;
                mat.m02 = m13;
                mat.m03 = m14;
                mat.m10 = m21;
                mat.m11 = m22;
                mat.m12 = m23;
                mat.m13 = m24;
                mat.m20 = m31;
                mat.m21 = m32;
                mat.m22 = m33;
                mat.m23 = m34;
                mat.m30 = m41;
                mat.m31 = m42;
                mat.m32 = m43;
                mat.m33 = m44;

                return mat;
        }
}



[Serializable]
public struct SerialSkeletonData {
        public NuiSkeletonTrackingState eTrackingState;
 public uint dwTrackingID;
 public uint dwEnrollmentIndex_NotUsed;
 public uint dwUserIndex;
 public SerialVec4 Position;
 public SerialVec4[] SkeletonPositions;
 public NuiSkeletonPositionTrackingState[] eSkeletonPositionTrackingState;
 public uint dwQualityFlags;

        public SerialSkeletonData (NuiSkeletonData nui) {
                this.eTrackingState = nui.eTrackingState;
  this.dwTrackingID = nui.dwTrackingID;
  this.dwEnrollmentIndex_NotUsed = nui.dwEnrollmentIndex_NotUsed;
  this.dwUserIndex = nui.dwUserIndex;
  this.Position = new SerialVec4(nui.Position);
  this.SkeletonPositions = new SerialVec4[20];
                for(int ii = 0; ii < 20; ii++){
                        this.SkeletonPositions[ii] = new
SerialVec4(nui.SkeletonPositions[ii]);
                }
  this.eSkeletonPositionTrackingState =
nui.eSkeletonPositionTrackingState;
```

```
                this.dwQualityFlags = nui.dwQualityFlags;
            }


            public NuiSkeletonData deserialize() {
                NuiSkeletonData nui = new NuiSkeletonData();
                nui.eTrackingState = this.eTrackingState;
                nui.dwTrackingID = this.dwTrackingID;
                nui.dwEnrollmentIndex_NotUsed = this.dwEnrollmentIndex_NotUsed;
                nui.dwUserIndex = this.dwUserIndex;
                nui.Position = this.Position.deserialize();
                nui.SkeletonPositions = new Vector4[20];
                for(int ii = 0; ii < 20; ii++){
                    nui.SkeletonPositions[ii] =
this.SkeletonPositions[ii].deserialize();
                }
                nui.eSkeletonPositionTrackingState =
this.eSkeletonPositionTrackingState;
                nui.dwQualityFlags = this.dwQualityFlags;
                return nui;
            }
        }

        [Serializable]
        public struct SerialSkeletonFrame
        {
            public Int64 liTimeStamp;
            public uint dwFrameNumber;
            public uint dwFlags;
            public SerialVec4 vFloorClipPlane;
            public SerialVec4 vNormalToGravity;
            public SerialSkeletonData[] SkeletonData;

            public SerialSkeletonFrame (NuiSkeletonFrame nui) {
                this.liTimeStamp = nui.liTimeStamp;
                this.dwFrameNumber = nui.dwFrameNumber;
                this.dwFlags = nui.dwFlags;
                this.vFloorClipPlane = new SerialVec4(nui.vFloorClipPlane);
                this.vNormalToGravity = new SerialVec4(nui.vNormalToGravity);
                this.SkeletonData = new SerialSkeletonData[6];
                for(int ii = 0; ii < 6; ii++){
                    this.SkeletonData[ii] = new
SerialSkeletonData(nui.SkeletonData[ii]);
                }
            }

            public NuiSkeletonFrame deserialize() {
                NuiSkeletonFrame nui = new NuiSkeletonFrame();
                nui.liTimeStamp = this.liTimeStamp;
                nui.dwFrameNumber = this.dwFrameNumber;
                nui.dwFlags = this.dwFlags;
                nui.vFloorClipPlane = this.vFloorClipPlane.deserialize();
                nui.vNormalToGravity = this.vNormalToGravity.deserialize();
                nui.SkeletonData = new NuiSkeletonData[6];
                for(int ii = 0; ii < 6; ii++){
                    nui.SkeletonData[ii] =
this.SkeletonData[ii].deserialize();
                }
                return nui;
            }
        }
```

```
        public struct NuiImageViewArea
        {
            int eDigitalZoom_NotUsed;
            long lCenterX_NotUsed;
            long lCenterY_NotUsed;
        }

        [StructLayout(LayoutKind.Sequential)]
        public class NuiImageBuffer
        {
                public int m_Width;
                public int m_Height;
                public int m_BytesPerPixel;
                public IntPtr m_pBuffer;
        }

        // Reference: http://msdn.microsoft.com/en-
us/library/nuiimagecamera.nui_image_frame.aspx
        [StructLayoutAttribute(LayoutKind.Sequential)]
        public struct NuiImageFrame
        {
                public Int64 liTimeStamp;
                public uint dwFrameNumber;
                public NuiImageType eImageType;
                public NuiImageResolution eResolution;
                //[MarshalAsAttribute(UnmanagedType.LPStruct)]
                public IntPtr pFrameTexture;
                public uint dwFrameFlags_NotUsed;
                public NuiImageViewArea ViewArea_NotUsed;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct ColorCust
        {
                public byte b;
                public byte g;
                public byte r;
                public byte a;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct ColorBuffer
        {
                [MarshalAs(UnmanagedType.ByValArray, SizeConst = 640 * 480, ArraySubType
= UnmanagedType.Struct)]
                public ColorCust[] pixels;
        }
        [StructLayout(LayoutKind.Sequential)]
        public struct DepthBuffer
        {
                [MarshalAs(UnmanagedType.ByValArray, SizeConst = 320 * 240, ArraySubType
= UnmanagedType.I2)]
                public short[] pixels;
        }

        [StructLayoutAttribute(LayoutKind.Sequential)]
        public struct NuiLockedRect
        {
                public int pitch;
                public int size;
                //[MarshalAsAttribute(UnmanagedType.U8)]
                public IntPtr pBits;
```

```
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct rect
        {
                long left;
                long right;
                long top;
                long bottom;
        }

        [StructLayout(LayoutKind.Sequential)]
        public struct NuiSurfaceDesc{
                uint width;
                uint height;
        }

        // to marshal the data from NuiImageFrame to this struct
        // reference: http://msdn.microsoft.com/en-
us/library/nuisensor.inuiframetexture.aspx
        [Guid("13ea17f5-ff2e-4670-9ee5-1297a6e880d1")]
        [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
        [ComImport()]
        public interface INuiFrameTexture
        {
                [MethodImpl (MethodImplOptions.InternalCall,
MethodCodeType=MethodCodeType.Runtime)]
                [PreserveSig]
                int BufferLen();
                [MethodImpl (MethodImplOptions.InternalCall,
MethodCodeType=MethodCodeType.Runtime)]
                [PreserveSig]
                int Pitch();
                [MethodImpl (MethodImplOptions.InternalCall,
MethodCodeType=MethodCodeType.Runtime)]
                [PreserveSig]
                int LockRect(uint Level,ref NuiLockedRect pLockedRect,IntPtr pRect, uint
Flags);
                [MethodImpl (MethodImplOptions.InternalCall,
MethodCodeType=MethodCodeType.Runtime)]
                [PreserveSig]
                int GetLevelDesc(uint Level, ref NuiSurfaceDesc pDesc);
                [MethodImpl (MethodImplOptions.InternalCall,
MethodCodeType=MethodCodeType.Runtime)]
                [PreserveSig]
                int UnlockRect(uint Level);
        }

        public class NativeMethods
        {
                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiSetDeviceStatusCallback")]
            public static extern void NuiSetDeviceStatusCallback(NuiStatusProc
callback, IntPtr pUserData);

            [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiInitialize")]
            public static extern int NuiInitialize(NuiInitializeFlags dwFlags);

               [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiShutdown")]
```

```
            public static extern void NuiShutdown();

                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiCameraElevationSetAngle")]
                public static extern int NuiCameraSetAngle(long angle);

                /*
                 * kinect skeleton functions
                 */

                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiSkeletonTrackingEnable")]
            public static extern int NuiSkeletonTrackingEnable(IntPtr hNextFrameEvent,
NuiSkeletonFlags dwFlags);

            [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiSkeletonGetNextFrame")]
            public static extern int NuiSkeletonGetNextFrame(uint dwMillisecondsToWait,
ref NuiSkeletonFrame pSkeletonFrame);

                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiTransformSmooth")]
            public static extern int NuiTransformSmooth(ref NuiSkeletonFrame
pSkeletonFrame,ref NuiTransformSmoothParameters pSmoothingParams);

                // get bone orientation
                // kinect Sdk 1.7.0, June 2013, Jason Added
                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiSkeletonCalculateBoneOrientations")]
            public static extern UInt32 NuiSkeletonCalculateBoneOrientations([In] ref
NuiSkeletonData pSkeletonData, [Out] NuiSkeletonBoneOrientation[] pBoneOrientations);


                /*
                 * kinect video functions
                 */

                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiImageStreamOpen")]
            public static extern int NuiImageStreamOpen(NuiImageType eImageType,
NuiImageResolution eResolution, uint dwImageFrameFlags_NotUsed, uint dwFrameLimit,
IntPtr hNextFrameEvent, ref IntPtr phStreamHandle);

                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiImageStreamGetNextFrame")]
            public static extern int NuiImageStreamGetNextFrame(IntPtr phStreamHandle,
uint dwMillisecondsToWait, ref IntPtr ppcImageFrame);
                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiImageStreamReleaseFrame")]
            public static extern int NuiImageStreamReleaseFrame(IntPtr phStreamHandle,
IntPtr ppcImageFrame);

                // KK addition - setting image stream flags to enable near mode
                [DllImportAttribute(@"C:\Windows\System32\Kinect10.dll", EntryPoint =
"NuiImageStreamSetImageFrameFlags")]
                public static extern int NuiImageStreamSetImageFrameFlags (IntPtr
phStreamHandle, NuiImageStreamFlags dvImageFrameFlags);
                // end of KK addition
        }

}
```

Kinect Recorder

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using Kinect;

public class KinectRecorder : MonoBehaviour {

	public DeviceOrEmulator devOrEmu;
	private KinectInterface kinect;

	public string outputFile = "Assets/Kinect/Recordings/playback";


	private bool isRecording = false;
	private ArrayList currentData = new ArrayList();


	//add by lxjk
	private int fileCount = 0;
	//end lxjk


	// Use this for initialization
	void Start () {
		kinect = devOrEmu.getKinect();
	}

	// Update is called once per frame
	void Update () {
		if(!isRecording){
			if(Input.GetKeyDown(KeyCode.F10)){
				StartRecord();
			}
		} else {
			if(Input.GetKeyDown(KeyCode.F10)){
				StopRecord();
			}
			if (kinect.pollSkeleton()){
				currentData.Add(kinect.getSkeleton());
			}
		}
	}

	void StartRecord() {
		isRecording = true;
		Debug.Log("start recording");
	}

	void StopRecord() {
		isRecording = false;
		//edit by lxjk
		string filePath = outputFile+fileCount.ToString();
		FileStream output = new FileStream(@filePath,FileMode.Create);
		//end lxjk
		BinaryFormatter bf = new BinaryFormatter();

		SerialSkeletonFrame[] data = new SerialSkeletonFrame[currentData.Count];
		for(int ii = 0; ii < currentData.Count; ii++){
```

```
                    data[ii] = new
SerialSkeletonFrame((NuiSkeletonFrame)currentData[ii]);
                }
                bf.Serialize(output, data);
                output.Close();
                fileCount++;
                Debug.Log("stop recording");
        }
}
```

Kinect Sensor

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using UnityEngine;
using Kinect;

public class KinectSensor : MonoBehaviour, KinectInterface {
       //make KinectSensor a singleton (sort of)
       private static KinectInterface instance;
    public static KinectInterface Instance
    {
        get
        {
            if (instance == null)
                throw new Exception("There needs to be an active instance of the
KinectSensor component.");
            return instance;
        }
        private set
        { instance = value; }
    }

       /// <summary>
       /// how high (in meters) off the ground is the sensor
       /// </summary>
       public float sensorHeight;
       /// <summary>
       /// where (relative to the ground directly under the sensor) should the kinect
register as 0,0,0
       /// </summary>
       public Vector3 kinectCenter;
       /// <summary>
       /// what point (relative to kinectCenter) should the sensor look at
       /// </summary>
       public Vector4 lookAt;

       /// <summary>
       /// Variables used to pass to smoothing function. Values are set to default
based on Action in Motion's Research
       /// </summary>
       public float smoothing =0.5f;
       public float correction=0.5f;
       public float prediction=0.5f;
       public float jitterRadius=0.05f;
       public float maxDeviationRadius=0.04f;
       public bool enableNearMode = false;

       public NuiSkeletonFlags skeltonTrackingMode;


       /// <summary>
       ///variables used for updating and accessing depth data
       /// </summary>
       private bool updatedSkeleton = false;
       private bool newSkeleton = false;
       [HideInInspector]
```

```
        private NuiSkeletonFrame skeletonFrame = new NuiSkeletonFrame() { SkeletonData
= new NuiSkeletonData[6] };
        /// <summary>
        ///variables used for updating and accessing depth data
        /// </summary>
        private bool updatedColor = false;
        private bool newColor = false;
        [HideInInspector]
        private Color32[] colorImage;
        /// <summary>
        ///variables used for updating and accessing depth data
        /// </summary>
        private bool updatedDepth = false;
        private bool newDepth = false;
        [HideInInspector]
        private short[] depthPlayerData;

        //image stream handles for the kinect
        private IntPtr colorStreamHandle;
        private IntPtr depthStreamHandle;
        [HideInInspector]
        private NuiTransformSmoothParameters smoothParameters = new
NuiTransformSmoothParameters();

        float KinectInterface.getSensorHeight() {
                return sensorHeight;
        }
        Vector3 KinectInterface.getKinectCenter() {
                return kinectCenter;
        }
        Vector4 KinectInterface.getLookAt() {
                return lookAt;
        }


        void Awake()
        {
                if (KinectSensor.instance != null)
                {
                        Debug.Log("There should be only one active instance of the
KinectSensor component at at time.");
              throw new Exception("There should be only one active instance of the
KinectSensor component at a time.");
                }
                try
                {
                        // The MSR Kinect DLL (native code) is going to load into the
Unity process and stay resident even between debug runs of the game.
              // So our component must be resilient to starting up on a second run when
the Kinect DLL is already loaded and
              // perhaps even left in a running state.  Kinect does not appear to like
having NuiInitialize called when it is already initialized as
              // it messes up the internal state and stops functioning.  It is resilient
to having Shutdown called right before initializing even if it
              // hasn't been initialized yet.  So calling this first puts us in a good
state on a first or second run.
                        // However, calling NuiShutdown before starting prevents the
image streams from being read, so if you want to use image data
                        // (either depth or RGB), comment this line out.
                        //NuiShutdown();
```

```
                    int hr =
NativeMethods.NuiInitialize(NuiInitializeFlags.UsesDepthAndPlayerIndex |
NuiInitializeFlags.UsesSkeleton | NuiInitializeFlags.UsesColor);
            if (hr != 0)
                    {
              throw new Exception("NuiInitialize Failed.");
                    }

                    hr = NativeMethods.NuiSkeletonTrackingEnable(IntPtr.Zero,
skeltonTrackingMode);
                    if (hr != 0)
                    {
                           throw new Exception("Cannot initialize Skeleton Data.");
                    }

                    depthStreamHandle = IntPtr.Zero;
                    hr =
NativeMethods.NuiImageStreamOpen(NuiImageType.DepthAndPlayerIndex,
NuiImageResolution.resolution320x240, 0, 2, IntPtr.Zero, ref depthStreamHandle);
                    //Debug.Log(depthStreamHandle);
                    if (hr != 0)
                    {
                           throw new Exception("Cannot open depth stream.");
                    }

                    colorStreamHandle = IntPtr.Zero;
                    hr = NativeMethods.NuiImageStreamOpen(NuiImageType.Color,
NuiImageResolution.resolution640x480, 0, 2, IntPtr.Zero, ref colorStreamHandle);
                    //Debug.Log(colorStreamHandle);
                    if (hr != 0)
                    {
                           throw new Exception("Cannot open color stream.");
                    }
                    colorImage = new Color32[640*480];

                    double theta = Mathf.Atan((lookAt.y+kinectCenter.y-sensorHeight)
/ (lookAt.z + kinectCenter.z));
                    long kinectAngle = (long)(theta * (180 / Mathf.PI));
                    NativeMethods.NuiCameraSetAngle(kinectAngle);

                    DontDestroyOnLoad(gameObject);
                    KinectSensor.Instance = this;
                    NativeMethods.NuiSetDeviceStatusCallback(new NuiStatusProc(),
IntPtr.Zero);
             }

             catch (Exception e)
             {
                    Debug.Log(e.Message);
             }
      }


      void LateUpdate()
      {
             updatedSkeleton = false;
             newSkeleton = false;
             updatedColor = false;
             newColor = false;
             updatedDepth = false;
             newDepth = false;
      }
```

```
        /// <summary>
        ///The first time in each frame that it is called, poll the kinect for updated
skeleton data and return
        ///true if there is new data. Subsequent calls do nothing and return the same
value.
        /// </summary>
        /// <returns>
        /// A <see cref="System.Boolean"/> : is there new data this frame
        /// </returns>
        bool KinectInterface.pollSkeleton()
        {
                if (!updatedSkeleton)
                {
                        updatedSkeleton = true;
                        int hr = NativeMethods.NuiSkeletonGetNextFrame(100,ref
skeletonFrame);
                        if(hr == 0)
                        {
                                newSkeleton = true;
                        }
                        smoothParameters.fSmoothing = smoothing;
                        smoothParameters.fCorrection = correction;
                        smoothParameters.fJitterRadius = jitterRadius;
                        smoothParameters.fMaxDeviationRadius = maxDeviationRadius;
                        smoothParameters.fPrediction = prediction;
                        hr = NativeMethods.NuiTransformSmooth(ref skeletonFrame,ref
smoothParameters);
                }
                return newSkeleton;
        }

        NuiSkeletonFrame KinectInterface.getSkeleton(){
                return skeletonFrame;
        }

        /// <summary>
        /// Get all bones orientation based on the skeleton passed in
        /// </summary>
        /// <returns>
        /// Bone Orientation in struct of NuiSkeletonBoneOrientation, quarternion and
matrix
        /// </returns>
        NuiSkeletonBoneOrientation[]
KinectInterface.getBoneOrientations(Kinect.NuiSkeletonData skeletonData){
                NuiSkeletonBoneOrientation[] boneOrientations = new
NuiSkeletonBoneOrientation[(int)(NuiSkeletonPositionIndex.Count)];
                NativeMethods.NuiSkeletonCalculateBoneOrientations(ref skeletonData,
boneOrientations);
                return boneOrientations;
        }

        /// <summary>
        ///The first time in each frame that it is called, poll the kinect for updated
color data and return
        ///true if there is new data. Subsequent calls do nothing and return the same
value.
        /// </summary>
        /// <returns>
        /// A <see cref="System.Boolean"/> : is there new data this frame
        /// </returns>
        bool KinectInterface.pollColor()
        {
```

```
            if (!updatedColor)
            {
                  updatedColor = true;
                  IntPtr imageFramePtr = IntPtr.Zero;

                  int hr =
NativeMethods.NuiImageStreamGetNextFrame(colorStreamHandle, 100, ref imageFramePtr);
                  if (hr == 0){
                        newColor = true;
                        NuiImageFrame imageFrame =
(NuiImageFrame)Marshal.PtrToStructure(imageFramePtr, typeof(NuiImageFrame));

                        INuiFrameTexture frameTexture =
(INuiFrameTexture)Marshal.GetObjectForIUnknown(imageFrame.pFrameTexture);

                        NuiLockedRect lockedRectPtr = new NuiLockedRect();
                        IntPtr r = IntPtr.Zero;

                        frameTexture.LockRect(0,ref lockedRectPtr,r,0);
                        colorImage = extractColorImage(lockedRectPtr);

                        hr =
NativeMethods.NuiImageStreamReleaseFrame(colorStreamHandle, imageFramePtr);
                  }
            }
            return newColor;
      }

      Color32[] KinectInterface.getColor(){
            return colorImage;
      }

      /// <summary>
      ///The first time in each frame that it is called, poll the kinect for updated
depth (and player) data and return
      ///true if there is new data. Subsequent calls do nothing and return the same
value.
      /// </summary>
      /// <returns>
      /// A <see cref="System.Boolean"/> : is there new data this frame
      /// </returns>
      bool KinectInterface.pollDepth()
      {
            if (!updatedDepth)
            {
                  updatedDepth = true;
                  IntPtr imageFramePtr = IntPtr.Zero;

                  /* KK Addition*/
                  /// <summary>
                  /// Sets near mode - move this into the Awake () function
                  /// if you do not need to constantly change between near and far
mode
                  /// current organization is this way to allow for rapid changes
IF they're required
                  /// and to allow for experimentation with the 2 modes
                  /// </summary>
                  if (enableNearMode)
                  {
                        NativeMethods.NuiImageStreamSetImageFrameFlags

      (depthStreamHandle, NuiImageStreamFlags.EnableNearMode);
```

```
                        //test = NativeMethods.NuiImageStreamSetImageFrameFlags
                        //
        (depthStreamHandle, NuiImageStreamFlags.TooFarIsNonZero);
                }

                else
                {
                        NativeMethods.NuiImageStreamSetImageFrameFlags

(depthStreamHandle, NuiImageStreamFlags.None);
                }

                int hr =
NativeMethods.NuiImageStreamGetNextFrame(depthStreamHandle, 100, ref imageFramePtr);
                if (hr == 0){
                        newDepth = true;
                        NuiImageFrame imageFrame;
                        imageFrame =
(NuiImageFrame)Marshal.PtrToStructure(imageFramePtr, typeof(NuiImageFrame));

                        INuiFrameTexture frameTexture =
(INuiFrameTexture)Marshal.GetObjectForIUnknown(imageFrame.pFrameTexture);

                        NuiLockedRect lockedRectPtr = new NuiLockedRect();
                        IntPtr r = IntPtr.Zero;

                        frameTexture.LockRect(0,ref lockedRectPtr,r,0);
                        depthPlayerData = extractDepthImage(lockedRectPtr);

                        frameTexture.UnlockRect(0);
                        hr =
NativeMethods.NuiImageStreamReleaseFrame(depthStreamHandle, imageFramePtr);
                }
        }
        return newDepth;
    }

    short[] KinectInterface.getDepth(){
        return depthPlayerData;
    }

    private Color32[] extractColorImage(NuiLockedRect buf)
    {
        int totalPixels = 640*480;
        Color32[] colorBuf = colorImage;
        ColorBuffer cb =
(ColorBuffer)Marshal.PtrToStructure(buf.pBits,typeof(ColorBuffer));

        for (int pix = 0; pix < totalPixels; pix++)
        {
                colorBuf[pix].r = cb.pixels[pix].r;
                colorBuf[pix].g = cb.pixels[pix].g;
                colorBuf[pix].b = cb.pixels[pix].b;

        }
        return colorBuf;
    }

    private short[] extractDepthImage(NuiLockedRect lockedRect)
    {
        DepthBuffer db =
(DepthBuffer)Marshal.PtrToStructure(lockedRect.pBits,typeof(DepthBuffer));
```

```
                return db.pixels;
        }

        void OnApplicationQuit()
        {
                NativeMethods.NuiShutdown();
        }

}
```

Main Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Runtime.CompilerServices;
using UnityEngine;

//Game Definitions

//Array of objects to spawn
public GameObject[] TopDown;

//Time it takes to spawn Topdown
public float SpawnDelay = 0;
public float theCountdown = 0;

// the range of X
public float xMin;
public float xMax;

// the range of y
public float yMin;
public float yMax;

//////////////////////////////////////////////////////////Depth Isolation




//////////////////////////////////////////////////////////Floor Removal




//////////////////////////////////////////////////////////Obstacle Scoping




//////////////////////////////////////////////////////////Closest Obstacle Calculation


int minint = array[0];                //this code needs to be applied to each array
column of the obstacle scoped matrix
foreach (int value in array) {
  if (value<minint) minint = value;
}


//////////////////////////////////////////////////////////Top Down Image Translation

//Array of obstacle positions
public GameObject[] TopDown;

//Time taken to show objects
[Space(1)]
```

```
public float SpawnDelay = 0;
public float theCountdown = 0;

// the range of X
public float xMin;
public float xMax;

// the range of y
public float yMin;
public float yMax;

public void Update()
{
    // timer to spawn the next Topdown Object
    theCountdown -= Time.deltaTime;
    if (theCountdown <= 0)
    {
        SpawnTopDown();
        theCountdown = SpawnDelay;
    }
}
//This code should be able to be written and addapted seperately without the spawn
function and applied to a plane as a texture for comparison to the depth image


//////////////////////////////////////////////////////////Object Spawning


void SpawnTopDown()
{
    // Defines the min and max ranges for x and y
    Vector2 pos = new Vector2(Range(xMin, xMax), Range(yMin, yMax));

    // Choose a new gridmemeber to spawn from the array
    GameObject gridmemberPrefab = TopDown[Random.Range(0, TopDown.Length)];

    // Creates the random object at the random 2D position.
    Instantiate(gridmemberPrefab, pos, );

       }
}
```