THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Using Motion Controllers in Virtual Conferencing

Jesse Dean

This thesis is submitted in partial fulfilment of the requirements for the Degree of Master of Science at the University of Waikato

March 2012

Abstract

At the end of 2010 Microsoft released a new controller for the Xbox 360 called Kinect. Unlike ordinary video game controllers, the Kinect works by detecting the positions and movements of a user's entire body using the data from a sophisticated camera that is able to detect the distance between itself and each of the points on the image it is capturing. The Kinect device is essentially a low-cost, widely available motion capture system. Because of this, almost immediately many individuals put the device to use in a wide variety applications beyond video games.

This thesis investigates one such use; specifically the area of virtual meetings. Virtual meetings are a means of holding a meeting between multiple individuals in multiple locations using the internet, akin to teleconferencing or video conferencing. The defining factor of virtual meetings is that they take place in a virtual world rendered with 3D graphics; with each participant in a meeting controlling a virtual representation of them self called an avatar.

Previous research into virtual reality in general has shown that there is the potential for people to feel highly immersed in virtual reality, experiencing a feeling of really 'being there'. However, previous work looking at virtual meetings has found that existing interfaces for users to interact with virtual meeting software can interfere with this experience of 'being there'. The same research has also identified other short comings with existing virtual meeting solutions.

This thesis investigates how the Kinect device can be used to overcome the limitations of exiting virtual meeting software and interfaces. It includes a detailed description of the design and development of a piece of software that was created to demonstrate the possible uses of the Kinect in this area. It also includes discussion of the results of real world testing using that software, evaluating the usefulness of the Kinect when applied to virtual meetings.

Acknowledgements

It would not have been possible to undertake and complete this project without the support and assistance of many individuals.

I would like to thank Bill Rogers and Mark Apperley who formulated the idea for the project, and whose input and assistance were invaluable over its course.

I would also like to thank my family and friends who supported me in countless ways over the course of this project.

Finally I would like to thank the University of Waikato for providing me with the opportunity to undertake this research.

Contents

List of Figures

List of Equations

List of Tables

Chapter 1: Introduction

Using Motion Controllers in Virtual Conferencing is a research project that's purpose was exploring how 3D cameras with the capability to recognise and track the movements of human bodies could be used to enhance meeting in a virtual world. The project came about in response to the release of the Kinect by Microsoft. The Kinect is a device equipped with components that give it the ability to capture video of a person, identify that person's body, and track the movements of their body in 3D space.

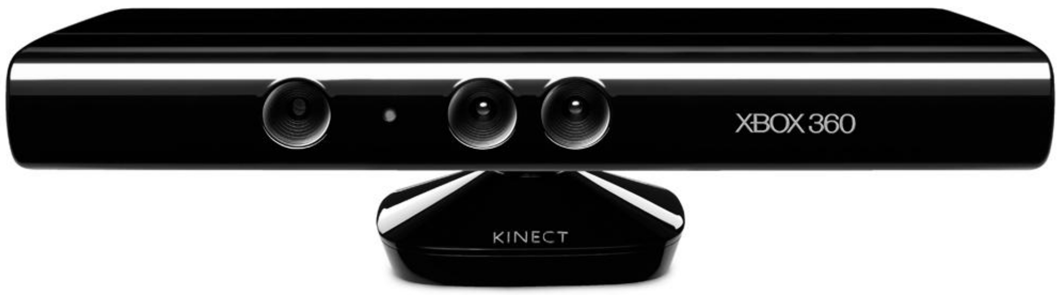


Figure 1: The Kinect Device

The Kinect was produced by Microsoft and first announced under the code name "Project Natal" at the Electronic Entertainment Expo (E3) in June 2009. The device and its associated software have several features, including: full 3D human body tracking, a microphone array backed by sophisticated voice recognition technology, and facial recognition technology. It was designed to allow a person to control a Microsoft Xbox 360 game console using nothing but their body movements and voice (Microsoft, 2009). To achieve all of this, the device is equipped with several sensors.

The first is a colour video camera. It can be seen on the device (see Figure 1) as the central lens on the front. It is capable of capturing video with a resolution of 640 x 480 pixels at 30 frames per second. Aside from providing a video stream for general use, the camera is used as part of the facial recognition system.

The second is a pair of devices that work is tandem: an infrared laser projector (the lens visible on the far left of Figure 1) and an infrared camera (on the right).

The projector emits a particular light pattern across the area in front of the device. This pattern is recognised by the infrared camera, and is used to determine the distance to all of the objects within the camera's field of view. This distance data is then used to separate users from the background in the Kinect's field of view and determine the locations and body positions of those users.

The final sensing device is an array of four microphones to pick up sound. This array can be used to accurately determine the direction from which a sound is coming.

The sensing devices are all mounted across the long bar section of the Kinect. This section is mounted on the Kinect's stand via a motorised joint which is capable of tilting the bar to direct its field of view up and down. This can be used to properly align the camera to get the best view of the scene in front of the device.

This kind of technology has existed for some time; however the Kinect is the first example of this technology that has been mass produced as an affordable consumer device (Zafrulla, Brashear, Starner, Hamilton, & Presti, 2001). It was originally intended to be used exclusively a controller for the Xbox 360, but later its use was expanded to applications on personal computers. It is the new found accessibility of this technology to the general population that has made it desirable to explore its potential applications in various aspects of everyday life.

There are many solutions available for interacting with the Kinect device using a computer. Along with Microsoft's official Kinect for Windows SDK there are a host of unofficial third party systems, most of which were developed before the release of Microsoft's SDK. Of these third party solutions, the most widely used is a system called the OpenNI framework (Hinchman, 2011). OpenNI was created by a company called PrimeSense who were involved in the development of the 3D sensing technology that is used by the Kinect (Gohring, 2010).

The potential application for the technology that was identified for this project is in the area of virtual meetings. Virtual meetings are a way of holding a meeting with multiple participants, all of whom may be in different locations around the world connected using the Internet. A virtual meeting differs from other forms of remote group communication such as a video conference or teleconference in that it takes place in a virtual environment rendered with 3D graphics, much like a computer game. In a virtual meeting each person present in the meeting is represented by a 3D rendered avatar that exists as part of the virtual environment. The avatar will generally take the shape of a person, and will frequently have its appearance customised by the user that it represents.

When participating in a virtual meeting, a user will control their avatar, issuing commands that will cause the avatar to perform actions in the virtual environment. These actions could be anything from performing a gesture to indicate some reaction from the user (e.g. clapping or laughing) to having the avatar move to a particular location within the virtual world. Typically a user will have no representation to the other participants in the meeting besides their avatar. That is to say, there generally won't be an image or video feed of a user available to other participants in the meeting.

When a user wishes to communicate with other participants in a virtual meeting they will typically do so in one of two ways. One is, as mentioned above, to give their avatar an instruction to exhibit some kind of body language. The other way is through more direct communication either using text chat within the software that manages the meeting or (more likely) through voice chat.

Previous work has been done at the University of Waikato that investigated the advantages and drawbacks of existing virtual meeting software (in particular a program called Second Life (Linden Labs, 2003)). This research found that there were several problems with virtual meetings in Second Life. Of particular note were limitations with the user avatars; these included clumsy and unintuitive controls for avatars and the generally limited range of expression that the provided. Despite these issues the research showed that there was also a wide array of potential advantages of virtual meetings over other kinds of remote meeting; these advantages will be discussed in detail in Chapter 2. This made it desirable to seek ways to overcome the limitations, so that the advantages could be fully realised.

The purpose of the project completed for this thesis was to build on findings of this previous work and investigate how a motion controller (in the form of the

Kinect) could be applied to both further enhance the existing advantages of virtual meetings, and to reduce or negate the drawbacks. To do this it was envisioned that the project would involve the creation of a platform upon which various new motion controller based features could be built and tested by users.

Over the course of the project such a platform has been developed as a standalone piece of software (named VMX) that utilises the technology provided by the Kinect to enable virtual meetings to be carried out using a variety of new motion control dependant features. Chapter 5 will explain in detail what these features are, and how they work.

When using VMX users normally sit at their desks, in front of their computers. However the software also supports a user standing up in front of a display screen near their computer in order to perform a presentation, using the screen to display visual aids (the content on the screen is captured and shown in the virtual environment).

As with ordinary virtual meeting software, VMX renders a virtual meeting room with 3D graphics. Within the virtual environment users are represented by avatars. VMX also includes a networking system, which allows users on multiple computers to connect together and join a single meeting. Each user is able to see each other user's avatar positioned somewhere in the virtual environment. Unlike ordinary virtual meeting software, VMX uses the Kinect to detect the user's body position; this information is used to pose the user's avatar.

The software makes use of the data provided by the Kinect in other ways as well. Through a process of experimentation several other features were developed and refined. Users are able to use a variety of hand gestures to interact with elements in the virtual world. Additionally, the body position data is used in tandem with the video feed from the Kinect's colour camera to capture an image of each user's face, which can be used as the face of their avatar. Another use of the Kinect is to allow the user to control their view of the virtual environment by their body position alone; this allows hands free operation of the software, and frees the keyboard and mouse for other uses.

VMX also includes the facility for users to give presentations to other users. The core of this functionality is a large display screen in the virtual environment on to which a user may put any image they wish. This display screen was enhanced (again through a process of experimentation) with Kinect based features that allow the user who is presenting to use a real world screen as a counter-part to the virtual display screen. This real world screen can be used as a reference for the user, giving them a clear idea of what is currently displayed on the virtual screen, and allowing them to precisely control where on the screen their avatar is pointing.

Over the course of the project the individual features of VMX were refined by informal experimentation. In order to gain more informative feedback on the software, in the final stages of the project VMX was put to use in a usability trial designed to test the new Kinect based features in a real world situation. The experiments involved putting the software in the hands of people who had a need to conduct a meeting. The people involved held a meeting and afterwards gave feedback on their experiences using the software. This feedback was used to evaluate the value of the features that were implemented in VMX, and to suggest future paths of exploration for applying motion tracking technology in this area.

The remainder of this thesis is arranged into seven chapters.

Chapter 2 looks into previous work on the main elements of this project. It includes a brief look at the history of motion controllers. It also reviews another project which was similar to this one; it involved the application of Kinect technology to improve the experience of video conferencing. The chapter also looks at previous work that has been done in the realm of virtual meetings. This includes information about the original work that inspired this project, looking into their advantages and limitations. It also briefly looks at some work done in hand gesture recognition, a feature that was investigated in this project.

Chapter 3 discusses how this project developed and evolved. It looks at the key goal of the project, lays out the steps taken to achieve that goal, and discusses each step in turn. An overview of the purpose of each step is given, along with a description of the way it was carried out over the course of the project. Also discussed are the different approaches that were considered for completing each step, and an explanation of why each approach was either adopted or rejected.

Chapter 4 takes a look at the underlying software technology that supports the main software that was created for this project. This chapter outlines the aspects of that technology that were used in the development of the software. The primary purpose of this chapter is to give a platform upon which Chapter 5 can discuss the main software made in this project.

Chapter 5 gives the details of the implementation of VMX. It discusses in detail the way in which each element of the software is built and how it functions. Alongside the technical details, this chapter also talks about the history of the development of each feature of the software, looking at key technical hurdles that were encountered in their development and how those hurdles were overcome. This chapter gives detailed descriptions of all of the algorithms and systems within VMX.

Chapter 6 talks about the user testing phase for VMX. It outlines the goals of user testing and discusses how user testing was approached in this project. This chapter details the experiment that was designed for user testing. After this, the chapter goes on to give the results that were gathered from running this experiment and discusses those results. This discussion looks at what can be learned from the results and how the software might be improved in response to them.

Chapter 7 is the conclusion of this document. It looks back at what was done and discusses the outcomes of the project. It also evaluates the final result in terms of the original goal of the project.

Chapter 8 considers future paths of research that might arise from this project. It discusses future possibilities for features that could be added to the software to further enhance the process of holding and participating in a virtual meeting. The areas for future work discussed arise from both what was learned from the user testing that was done as part of this project, and from other areas of interest that come as natural progressions on the current software. The rationale for each feature is considered along with problems that might arise along the course of their development and potential solutions to those problems.

Chapter 2: Literature Review

This chapter investigates the background of this project. The chapter starts with a brief look at the history of motion controller technology. It then moves on to give an overview of previous research that was done in the area of virtual meetings in a project called “Virtual Worlds as Meeting Places” from which this project arose. This includes discussion of the advantages and disadvantages of virtual meetings over other ways that people use to hold meetings remotely. The chapter then moves on to look at a project that was undertaken by a group of students at another university that had a similar intention to this project. In that project, a piece of software was developed that aimed to enhance the process of holding a meeting by video conference using the technology provided by the Kinect. Then, the chapter looks at an advanced use of the Kinect’s data in gesture recognition. Finally the chapter will look back at all of this previous work in terms of what it means for this project.

2.1 Motion Controller History

Motion controller technology has existed for several decades. It began with simple systems that analysed ordinary video data to detect and analyse motion. Later very complex systems demonstrated ways to accurately detect movements of a human body using sophisticated sensor arrays and systems. Later still, technology for detecting human movements began to simplify and become more accessible to a wider range of people.

Early work on motion tracking tended to involve simple systems where ordinary cameras would be used and analysis would be performed to detect the motion of objects within the camera’s field of view. “Motion Tracking with an Active Camera” (Murray & Basu, 1994) described two common approaches to motion tracking in the early developmental stages of the technology: motion-based tracking and recognition-based tracking. Motion-based tracking works by detecting motion in a camera’s field of view, it is able to detect any kind of object and infer its motion. Recognition-based tracking is more complex. It works by recognising specific objects in the each image frame from the camera. By detecting the same object over multiple frames, its motion can be inferred. This kind of system has an advantage over motion-based systems in that the orientation of the object can be inferred in addition to its position. Recognition-based tracking may also have the capability of tracking motion in three dimensions. The main disadvantages of recognition-based systems are that they are only capable of detecting the motion of recognisable objects, and that they are computationally expensive compared to motion-based systems. The recognition-based systems have the most in common with the modern system used by the Kinect.

Historically systems for tracking the motion of people with high accuracy have been large, complex, and expensive. They often required controlled environments where the tracked user was alone in an empty, predefined space. One such system, called "Constellation" (Foxlin, Harrington, & Pfeifer, 1998) was demonstrated in 1998. The sole purpose of the system was to track the position and orientation of a user's head (for the purpose of orienting the camera in a virtual environment for a head mounted display the user was wearing).

The system was very complex, involving multiple different sensing devices. A user was required to wear several devices on their head including multiple ultrasonic rangefinder modules, and an inertial sensing instrument. The rangefinder modules would send coded infrared signals to beacons positioned in set places around the user. When a beacon received its particular code, it would emit an ultrasonic pulse. The rangefinder modules would detect this pulse, and use the time from when the infrared signal was sent, to when the pulse was detected to determine the distance from the beacon to the rangefinder. By tracking the distances to different beacons, the rangefinders were able to detect their exact location in 3D space. Even this however was not enough to guarantee a high degree of accuracy; the ultrasonic system was vulnerable to acoustic interference and echoes. To compensate for this, the system used the head mounted inertial sensor to filter out distance readings from the rangefinders that were not consistent with movements that the user was actually making. The resulting system was capable of a high degree of accuracy; but the complexity of the system made it prohibitive for small scale applications such as that looked at in this project.

In addition to systems that could track a particular part of the human body, other systems that could simultaneously track the movements of an entire human body were developed. An example of such a system was described in 2004 in a paper entitled “A Real-Time Articulated Human Motion Tracking Using Tri-Axis Inertial/Magnetic Sensors Package” (Zhu & Zhou, 2004). This system utilised micro-electromechanical accelerometers, rate gyros, and magnetometers (all incorporated into a single device called an “Integrated Sensor Pack”) to provide motion sensing, and involved a sophisticated algorithm for combining all of this data to produce highly accurate results. The system divided the human body into 15 segments, the position and orientation of each segment could be calculated by using multiple Integrated Sensor Packs strategically positioned around the human body. This system is closer in capability to the modern Kinect technology than those previously described in its ability to track separate parts of the human body. However, it is still very complex, requiring a user to wear a network of carefully placed sensors on their body; though it does have the advantage of being able to function in a room that requires no special preparation.

Throughout the development of these large and complex systems, research was being done on far simpler systems for detecting the movements of the human body. In 2000 a paper entitled “Stochastic Tracking of 3D Human Figures Using 2D Image Motion” (Sidenbladh, Black, & Fleet, 2000) demonstrated a system that was able to track the motion of a human body using only the colour video data from an ordinary camera. The system was primitive by the standards of the Kinect. It was unable to automatically detect the presence of a body to track, so it required manual setup of the initial position of different joints and limbs on the video image whenever a user was to be tracked. It also suffered from a problem where it would tend to lose track of its target after enough time had passed, requiring that it was setup again before use could continue.

In more recent times the technology has matured and systems that used depth sensing cameras to track human movements were developed. Depth cameras are useful because they make it easier to separate users from the objects behind them, and make it simple to determine the distance of different parts of a user’s body from the camera. In 2008 a paper entitled “Controlled human pose estimation from depth image streams” (Zhu, Dariush, & Fujimura, 2008) demonstrated a system that could track human body movements using only a depth camera. The system was successfully able to track the upper body of a person with reasonable accuracy. It was also capable of automatically detecting a body to track. This system is very close in functionality to that which was ultimately included in the Kinect.

The Kinect itself was released by Microsoft in late 2010. It was more sophisticated than the system described by Zhu et al. in its ability to track multiple people simultaneously and it ability to do full body tracking. The key part of the Kinect that brings the field of motion controllers up to a level where uses such as the one explored in this project are feasible, is the it offers reasonably accurate and robust tracking, at an affordable price, and in a mass produced, widely available device.

2.2 Virtual Worlds as Meeting Places

The motivation for using motion controllers in virtual conferencing lies in an earlier project that was completed at the University of Waikato in 2010. This project was called Virtual Worlds as Meeting Places (Al Qahtani, 2010). The aim of the project was to investigate a use of the virtual world program called Second Life (Linden Labs, 2003) to hold meetings between people in different places, using the Internet.

Second Life is an application that was created by a company called Linden Labs. It is an online persistent virtual world program. A user of this program will typically connect to a Second Life server using a client program (known as a Second Life Viewer). At any given time thousands of users may be connected to the Second Life servers (Plunkett, 2008). Each user is represented by an avatar within a very large virtual world. They are able to control their avatar and can move it around the virtual world. All users are able to see and interact with each other’s avatars. In Second Life the elements that make up the world (structures, object etc.) are built by users of the program and uploaded to the host server, where they become visible to all users. The capability for many users from anywhere on the planet to congregate and interact in a single place in a customisable virtual space is what makes Second Life suitable for virtual meetings; indeed in 2009 headlines were made when IBM reported saving $320,000 in organisation, travel and productivity costs by holding two conferences in Second Life, instead of the real world (Ashby, 2009).

A critical feature of Second Life that is identified in ‘Virtual Worlds as Meeting Places’ is the ability for avatars to perform ‘gestures’ on a user command. A gesture is a predefined action (such as clapping, waving, or performing a handshake). These gestures allow ways of communicating with other people using body language rather than just talking.

2.2.1 The Advantages of Virtual Meetings

Early on, ‘Virtual Worlds as Meeting Places’ identifies some of the potential advantages of virtual meetings over other forms of remote meetings. The first of these advantages contrasts to voice-only methods of meeting such as conference calls, in that virtual meetings allow for the use of visual aids (e.g. slideshows) within the virtual environment. The second advantage of virtual meetings over conference calls that, if a given participant is not familiar with the voices of some of the other participants then they may still be able to tell who is speaking by visually identifying their avatar (or its label).

The third advantage that is presented is unique to virtual meetings. Because a virtual meeting takes place in a single virtual space shared by all the participants, there is the opportunity for participants to share certain interactions such as a handshake. On the face of it, it may seem like a trivial and unimportant thing for two avatars to share a handshake; however in the field of virtual reality research there is a concept known as presence. Presence is a sense of ‘being there’ felt by a user in a virtual environment. Essentially a user forgets their real world surroundings and enters a mindset where they feel like they are truly inside the virtual environment. This concept is also known as ‘immersion’. Very closely related to the concept of presence is the concept of ‘copresence’; the sense of being there together with another person. This concept refers to the idea that a user can feel like they are truly within the virtual environment with the other users in the same virtual space. This means that when one user interacts with another user within the environment, it can have the same relationship building effects as performing the same interaction in real life. This is the value of being able to perform a handshake in the virtual environment. A detailed discussion of the concepts of presence and copresence can be found in the book *'The Social Life of Avatars'* (Schroeder, 2002).

The fourth advantage of virtual meetings stated in 'Virtual Worlds as Meeting Places' comes down to reduced need for travel. This advantage is shared with all forms of remote meeting. The lack of need to assemble every person involved in a meeting in a single physical location means that much time, effort and expense can be saved. This is especially true when the meeting involves people in different cities or countries, or when a large number of people are involved. The virtual conferences held by IBM that were mentioned previously are a prime example of this.

The fifth and final advantage is an interesting concept that is exclusive to virtual meetings (and real world meetings). The idea is that because a virtual conference takes place in a 3D space, users can use 3D modelled objects within the meeting as visual aids. While video of objects may be shown in a video conference, virtual meetings potentially allow all of the users in the environment to directly interact with objects.

2.2.2 Experimental Meetings

Three experiments were carried out over the course of the research done for 'Virtual Worlds as Meeting Places'. These experiments took the form of actual meetings using real-world participants. Each of the three meetings had a different purpose.

The first meeting was called the 'trial meeting' and was geared towards simply gaining some experience of what it is like to hold a meeting in Second Life. The meeting was informal and involved four people, all of whom already knew each other. After the meeting the participants were given a questionnaire that asked about their experiences during the meeting. The results of this meeting reinforced several of the advantages mentioned above. The participants generally reporting that they found virtual meetings a good way to meet with each other.

The second meeting was called the ‘informal meeting’. The informal meeting took place between seven people, all of whom were staff or students of the Faculty of Computing and Mathematical Science (FCMS) at the University of Waikato. It was in the preparation for this meeting that one of the disadvantages of using second life was made apparent. The only place where the participants were freely able to create the kind of objects one would use in a meeting (tables, chairs etc.) was in a special location in the Second Life world called ‘Sandbox Island’. The problem was that Sandbox Island is an open grass field that absolutely any user in Second Life can access, meaning that there was no privacy and a high chance of someone disrupting the meeting. For this particular experiment the problem was ultimately solved through a third party offering access to their own private, appropriately furnished space for the meeting to take place in. The meeting was conducted as an informal conversation between the participants. As with the trial meeting the participants were given a questionnaire to fill out after the meeting.

The third and final meeting was called the ‘formal meeting’. The formal meeting was intended to look at holding a business meeting or conference within Second Life. Unlike the informal meeting, this meeting required a virtual screen on which a slideshow presentation could be displayed. This meant that a new virtual venue equipped with such capabilities was needed. There are businesses that operate within Second Life, that hire out facilities for these kinds of meetings. One of these facilities was hired for the purposes of this experiment. There was an unfortunate drawback of this method, in that in order to show slides on the virtual screens, a fee must be paid per slide. The process of actually uploading things to this screen was also described as being somewhat difficult by the researcher. The participants for this experiment were again drawn from staff and students at the FCMS. The meeting started with one participant giving a presentation to the other participants in the meeting. After the presentation a follow up discussion was held. As with the other experiments, after the meeting the participants were asked to fill out a questionnaire.

2.2.3 Outcomes

A wide array of advantages, disadvantages and problems with virtual meetings were identified from the results of the experiments conducted for Virtual Worlds as Meeting Places. This section summarises the reported results of that project.

One of the interesting points that was noted in the results was that people found the experience of a virtual meeting more immersive than teleconference. This confirms the hypothesis that was given at the beginning of Virtual Worlds as Meeting Places, and indicates that there is potential for virtual meetings to feel like a real world meetings to their participants.

Several advantages that Virtual meetings hold over real world meetings were identified in the results. This included that there is the opportunity for meeting participants to return to their work faster than with real world meetings. This stems from the fact that it can be expected that a meeting participant will join a virtual meeting from their own computer in their own office, meaning that once the meeting is over they do not need to go anywhere else to resume working. This is especially beneficial in situations where the participants would be required to travel significant distances to get to a physical meeting. A counterpoint to this advantage was seen in the results as well; being at their own desks gave participants much opportunity for distraction. It was easy for participants to find something to do during a meeting that would distract their attention and cause them to lose focus on what was happening. There could be an advantage in this, in that if the meeting moves onto a topic that is not relevant to a given participant, then they can do something productive in the mean time. However, if a participant was distracted while something important was going on in the meeting, the usefulness of the meeting could be reduced.

A closely related idea that was noted in the results was that participants were free to carry out tasks without disrupting others in the meeting. This can be something simple, such as getting a cup of coffee or replying to a text message; or something more important, like dealing with a sudden or urgent situation.

Another advantage that was found from setting up the experiments, is that it is relatively cheap (and potentially free) to get access to large spaces that can house many meeting participants. This also applies to equipment in the virtual world (chairs, tables etc.). The biggest potential use of this advantage is in large scale virtual conferencing; the need to hire out expensive conference venues can be eliminated. Second Life does generally have small costs associated with getting access to virtual venues of a suitable size, but in a specialised virtual meeting application there would be no reason for this to be the case. There would however be costs associated with finding a server that could handle a large number of connected participants and the associated networking costs.

The ability to move around the virtual environment instantly was also identified as an advantage from the results of the experiments. The ability to traverse any distance and take no time doing so reduces wasted time. For example, if someone in a large conference was seated at the back of the virtual room and needed to get to the front, they could simply teleport there instead of walking.

The experiments also revealed several drawbacks in holding virtual meetings in Second Life. The first of these was the lack of certain tools that are generally available in real world meetings. In particular, the lack of a whiteboard for participants to draw and write on was noted as being an inconvenience.

Limitations with the user avatars in Second Life were the cause of several problems that were observed in the experiments. The first is that the avatars provide no means of confirming the identity of the person controlling them. An avatar associated with a particular Second Life account will appear the same regardless of who is using it. However, it is possible to tell the difference between separate avatars, as their appearance can be customised. A second and significant problem with the Second Life avatars was that they required constant input from the user to carry out actions. Users needed to issue keyboard and mouse commands to make their avatar do things in the virtual world. If they did not then their avatar would remain still. This could create an ambiguity in that it was impossible to see if a person was still at their computer and paying attention to meeting or not. An avatar that is receiving no commands from a present user looks exactly the same as an avatar whose user is absent. This made it critical that all users understood how to control their Second Life avatar. There was no passive system to do it for them. A more subtle problem with the avatars is that they provide no indication that their user wishes to speak. The consequence of this is that in the experiments users would often start talking over each other.

2.3 Video Conferences

Virtual meetings are a relatively unexplored field. They are not commonly used in everyday situations by the general public. However video conferences are widely used and share certain similarities with virtual meetings (held over the internet, have a visual component etc.). Because of this, and despite the relative newness of the Kinect device, prior to the start of this project work had already be done in attempting to use the Kinect to improve video conferencing software.

2.3.1 Kinected Conference

Kinected Conference (DeVincenzi, Yao, Ishii, & Raskar, Kinected conference: augmenting video imaging with calibrated depth and audio, 2011) is a piece of software that was developed by a group of students at MIT. Of any work that had been done at the outset of this project, Kinected Conference had the most in common with the software that was to be created for this project. Kinected Conference looked for ways to improve the experience of video conferencing. The features of the software make use of the extra data provided by the Kinect’s depth camera and microphone array to enhance the raw video feed. Kinected Conference essentially aimed to do for video conferencing what this project aimed to do for virtual conferencing.

Unlike the software in this project, Kinected Conference was intended to have more than one user per Kinect device. In fact no limit on the number of people that can use a single Kinect device with this software is discussed, and at times up to three people are actually demonstrated using a single device. This is feasible because Kinected conference does not rely on skeletal tracking of users at any point (skeletal tracking is the most significant limiter on how many people can use a single Kinect device at once). The Kinected conference software is also only designed to support two computer systems with Kinect devices being connected simultaneously, meaning that all of the participants of the conference must be in one of only two places, a clear limitation over potential virtual meeting applications.

Kinected Conference implemented several features that, in particular, took advantage of the ability of the Kinect’s audio array to determine the position of a speaker in the Kinect’s field of view, and the ability of the Kinect’s depth camera to identify the spatial location of objects in the view of the video camera. By using this information the software is able to perform a number of visual enhancements to the video feed that is being sent to the remotely connected participants of the meeting. These included focusing the camera on speakers, freezing parts of the camera image, and overlaying spatially contextual graphics.

One of the features presented with this software is called synthetic focusing. Synthetic focusing involves making different users appear in or out of focus depending on who is talking at any given time. The system is presented as working by determining which person in a scene is currently talking, leaving that person in clear focus on the video stream, and using simulated depth of field to blur the parts of the video stream that show the other participants in the meeting. The creators of the software talk of using the depth information from the Kinect to enable realistic degrees of blur to be applied to people who are sitting at different distances away from the camera and also to allow realistic smooth transitions when changing which users are in focus. The proposed rationale behind doing this blurring at all, is to simulate the real world depth of field effect that would be produced by our eyes when focusing on different people around a meeting table – an effect that is greatly reduced when looking at different people on a flat screen. The blurring of inactive users also reduces the likely hood that those users will cause distractions by carrying out other activities (e.g. checking their email). This is an advantage of virtual meetings that was identified in “Virtual Worlds as Meeting Places”, and is a clear example of the Kinect being used to improve video conferencing.

A second feature that the creators of the software describe takes this concept of enabling users to carry out activities without causing a distraction even further. It allows individual users to freeze the part of the video image that shows them, without affecting the parts of the video image showing other participants in the meeting. This allows participants carry out more distracting tasks and actions (e.g. leaving the table) with minimal disruption to other participants in the meeting. The creators of the program utilise the Kinect's depth stream data when deciding which parts of the image to leave frozen. This allows them to minimize the chance that the frozen part of the video will occlude any part of any other participant in the meeting by only freezing pixels when they would show something in a depth range that matched the location of the user who did not wish to be seen at that point. A particularly important result of this is that should a new participant move into the space between the camera and the location of the frozen participant, the system is able to recognise what parts of the image to unfreeze in order to show the new participant, avoiding having frozen pixels of the participant in the background being on top of the new participant in the foreground. A similar function could be utilised in this project to freeze all or part of the data from the Kinect to freeze a user avatar. The creators of the program even speak of the capability to selectively deleting all audio that originates from a particular location in the scene, meaning that if a frozen participant was doing something noisy, that noise would not be transmitted across the video conference link.

DeVincenzi et al. also talk about augmenting the video feed by drawing additional graphics on top of the video. These additional graphics can have a spatial relationship with objects in the scene (e.g. they could appear in focus when a user they were associated with was in focus). The creators give examples of the capability to show things such as name tags above participants in the meeting; other details about those participants such as files that they may be sharing in the meeting; or even the total amount of time that a particular participant has been speaking for over the course of the meeting. The creators also talk about these graphics having interactive elements, such as the ability to click on a person's name tag to get more information about them. This type of interface enhancement is simple to achieve in virtual meetings, and is already present in Second Life in the form of names above avatars.

In more recent publications on their website (DeVincenzi, Yao, Ishii, & Raskar, Kinected Conference | MIT Media Lab), the creators of the Kinected Conference software talk about and demonstrate additional features of the software.

One new feature is called ‘privacy areas’. Privacy areas are similar in function to the ability to freeze the image of a particular person in the video stream except that they allow all activity in a particular part of the room to be rendered invisible. As an example of how this can be used, the creators propose a situation where somebody wishes to set up a presentation in the background for a later part of the meeting, but does not wish to cause a distraction to the current part of the meeting. The software can also be told to hide things in the video that are beyond a certain distance from the camera. Hiding can take the form of overlaying an image of the room as it appeared beforehand or simply painting a solid colour over parts of the video.

A second new feature that is shown by the creators makes use of augmented reality principles to enhance the use of objects as visual aids in a meeting. This is demonstrated with wooden blocks sitting on the meeting table that participants are seated around. Using the depth data from the Kinect, the distance between the blocks is calculated and displayed as a graphic showing a line running between the blocks, labelled with the distance between those blocks. In another example certain blocks are equipped with data matrix codes that allow the software to recognise specific blocks and perform some kind of graphical enhancement to them. In the example shown, certain blocks have images of buildings overlaid on top of them, giving the appearance that the participants in the meeting are arranging buildings on the meeting table. This bears some tangential similarities with the concept addressed in “Virtual Worlds as Meeting Places” of having virtual objects within the meeting which participants can interact with.

Overall, “Kinected Conference” demonstrates how the Kinect can be successfully used to improve an existing form of remote meeting. Many of the features developed in that project, allow video conferencing to make use of augmented reality to address some of its limitations.

2.4 Gesture Detection

The Kinect SDK does not provide any support for gesture recognition. This meant that in order to make use of the Kinect to recognise gestures, a system would need to be implemented. Recognising simple large arm movements as gestures from the data available from the Kinect is trivial. However, finer details such as finger positions are not as simple.

In the past, attempts to track the location and position of a users hands and fingers from visual data have encountered a number of complicating factors. For example the uniform colour of the human hand, and the tendency for self-occlusion when in ordinary use. Specialised hardware in the form of motion capture technology or visual markers can ease these problems, but also present problems of their own in terms of ease of setup and use (Oikonomidis, Kyriazis, & Antonis, 2011). Kinect, being a relatively cheap and easily set up piece of hardware presents the opportunity to capitalise on the advantages of specialised hardware while limiting the associated negative impacts.

The Kinect SDK provides no built in functionality for detecting fingers. It provides only broad full body skeleton tracking on individuals. The skeleton tracking system does however provide an accurate position of a tracked user's hands when they are visible to the Kinect's depth camera. This information can be used in combination with depth stream data (i.e. raw data from the Kinect's depth sensor) to detect a user's finger positions.

Oikonomidis et al. demonstrate the feasibility of applying Kinect in this area with their own solution to the problem of hand tracking. They present what is described as a "model-based" approach where a 3D model of a hand is used to simulate the Kinect data produced by a particular hand position. This model data is then compared to the actual data being received from the Kinect sensor to determine how similar the current model is to the user's actual hand position. New model data is generated until it is deemed similar enough to the actual data, at which point the current model hands position is taken as the user's true hand position.

This method proved reasonably effective for its purpose; however it does come with notable drawbacks. The primary obstacle is that this algorithm is computationally expensive. Its creators required a powerful, modern system and needed to exploit the GPU to even get close to the real-time speeds. However in their report the creators do touch on an alternative class of algorithms to the model-based one they created. They describe this class as "appearance-based". Algorithms of this class map certain image features to particular hand positions that are specifically defined in the program. These algorithms are described as being well suited for problems where there is a small number of known hand positions that need to be detected.

2.5 Summary

Motion controller technology has existed for many decades. However for much of its history the technology has been too complex, expensive, and inaccessible for it to be used in consumer software. The Kinect changes this.

In “Virtual Worlds as Meeting Places” Al Qahtani outlined several advantages to virtual meetings. These were: the ability to use visual aids in presentations, the ability to visually identify meeting participants, the immersive qualities of having a shared space with other participants, the reduction in time and money costs for holding meetings and conferences, and the ability to collaboratively manipulate the virtual environment. These advantages demonstrate the value of this project pursuing virtual meetings as a means of holding remote meetings.

The work done by Al Qahtani identifies aspects of existing virtual meeting software that limit its usefulness. It identified the avatars in Second Life as being particularly unsuitable for their purpose. Their clumsy controls and lack of expression were key areas where the avatars had problems. These two areas show clear possibilities for improvements using the motion controllers, given these devices can capture a user’s body movement directly. Also, the experimental meetings that were held as part of “Virtual Worlds as Meeting Places” provide a basic model for a usability trial for the software created as a part of this project.

In “Kinected Conference” DeVincenzi et al. show that the application of the Kinect to remote meeting software can successfully lead to new features that can address limitations and create new, compelling advantages. This reinforces the idea that the Kinect can be used to improve virtual meeting software.

The work of Oikonomidis et al. demonstrates that the raw data of from the Kinect can be used in ways beyond what is provided by existing Kinect software. It also lays out potential paths for advanced gesture recognition for the software in this project.

Chapter 3: Project Design

This project began with work previously done at the University of Waikato investigating the advantages and limitations of virtual meetings. A few months before the project began Microsoft had released its Kinect motion controller for the Xbox 360. The core idea behind this project was to investigate ways in which this new technology could be used to address some of the limitations of virtual meetings that were found in the previous study, and to look for new ways to apply this technology to further improve the experience of holding a virtual meeting.

This chapter looks at the goals and history of the project. The first section of this chapter will outline the four key steps that were identified. The subsequent sections will look at each step and talk about the approach taken when attempting to complete those steps. Also discussed in these sections will be how these approaches were chosen, including alternative methods that were considered and the reasons for which they were ultimately rejected.

3.1 Project Outline

From the start of the project, the basic goal was to use the Kinect device to enhance the experience of participating in a virtual meeting. It took some time to clarify exactly how to go about achieving that goal. The project proceeded in four steps, listed below. The approach taken was experimental, ideas were formulated and refined through informal testing, so steps 2 and 3 were iterated several times.

1. Investigate the capabilities of the Kinect device and software.

2. Apply those capabilities to design potential enhancements for virtual meeting software.

3. Build these enhancements into a purpose built piece of software.

4. Evaluate the usefulness of those enhancements by testing that software with real people.

The following sections will discuss each of these steps in detail.

3.2 Kinect's Capabilities

The first step called for an investigation into the capabilities and limits of the Kinect device and the software available to interface with it.

At the time this project started (early in 2011), Microsoft provided no official support uses of the Kinect device outside of companies licenced to develop games for the Xbox 360. Despite this, since the release of the Kinect device in late 2010, third parties had been developing unofficial ways to interface with the Kinect from a PC.

To investigate the Kinect's practical capabilities and limitations, a piece of software was created for this project. This software could be thought of as the predecessor to the software that was to be created as part of the third goal of this project. The software was written in C++ and utilised libraries and drivers provided by a company called PrimeSense. PrimeSense was the company originally responsible for providing the 3D depth sensing technology that was used in the Kinect device. They elected to release their own software for interacting with a Kinect device. Their software is divided into two parts. One is called the OpenNI framework which is an API for creating programs that can make use the 3D camera data that the Kinect provides (the API is intended to be usable with a wider variety of 3D camera hardware than just the Kinect) (OpenNI.org, 2010). The other bit of software provided by PrimeSense is called NITE. NITE is responsible for the analysis of 3D data coming in from the Kinect; it is this software that provides things such as user skeleton tracking and gesture recognition (PrimeSense, 2011). PrimeSense also provided device drivers for using the Kinect with a PC (Joystiq, 2010).

The software that was created for this stage of the project had no 3D graphics and was largely directed at looking at the information that could be acquired from the Kinect, what that information looked like and how accurate it was. During this exploratory phase the software was programmed to do things such as using the Kinect to greet people as they walked through a door, or give them instructions based on what they were doing.

3.3 Potential Enhancements

This section looks at the specific ways that were considered for enhancing the process of holding a virtual meeting. Within each subsection, each individual enhancement is outlined and discussed.

The first of the enhancements that are listed here directly address problems and limitations associated with virtual meetings in the program Second Life that were encountered in the work for “Virtual Worlds as Meeting Places” as discussed in Chapter 2 of this thesis. These enhancements were intended as potential solutions to those problems. The remaining enhancements listed in this section were created to take advantage of the abilities of the Kinect to improve virtual meetings in new ways that had not previously been considered.

3.3.1 Avatar Control

One of the first of the problems to be identified by previous work in Second Life was that there was no way to tell if a user was actually present and paying attention during a meeting (just because their avatar is in the virtual meeting doesn’t mean the user is still at their computer) unless that user constantly issued commands to their avatar to perform actions (like clapping, laughing etc.). Essentially, a user was required to constantly and actively provide some kind of input if they wanted to indicate their continued presence at a meeting. A potential way in which the Kinect technology could be applied to solve this problem was obvious: use the skeleton tracking abilities of the Kinect to enable a user to passively puppeteer their avatar. The idea being that whenever a user moved, their avatar would perform the same movement. This meant that a user could indicate their presence simply by doing the things that one does when sitting down listening to somebody speak (look around, adjust sitting position etc.), without any active effort on their part to issue commands to control their avatar.

3.3.2 Facial Expression

A second problem encountered in the study done in Second Life was the lack of any way to gauge people’s reactions to what was being said in a meeting unless that person was explicit, either stating their reaction verbally, or commanding their avatar to perform some action that reflected their reactions. To an extent this is related to the first problem and is indeed partially solved by the solution to the first problem: if a person’s skeleton is being tracked by a Kinect sensor, and their movements are being reflected by their avatar, then that person’s body language will be visible to the other participants in the meeting. While body language may already be accounted for in this project, this does not entirely solve the problem as a person’s body language is only one way in which they can express their feelings.

A person’s facial expression also provides a way to gauge their feelings. As with body language, there is no automatic way to capture and broadcast a person’s facial expression when they are involved in a meeting in Second Life. Initially the idea of using the data from the Kinect to animate a user avatar’s face in the same way as their body was considered for this project. Ultimately however, it was decided that attempting to implement a system for doing this kind of face tracking would be too large of an undertaking. Furthermore, it would have required the implementation of some kind of facial animation system within the graphics system for this project, which would have taken even more time. Fortunately a simpler way of transmitting a user’s facial expression was available. The use of the skeleton tracking information, along with a pair of transformation provided by the Kinect SDK made it possible to isolate the part of the video feed of the Kinect that contained the image of the user’s face. This meant that it would be possible to simply texture a user’s avatar’s head with a live image of that user’s head.

3.3.3 Head Orientation

The first enhancements that weren’t derived from problems and limitations encountered in earlier work actually arose from a limitation with the Kinect itself. The system does not provide information about the current rotation of a user’s head in its skeleton tracking output. This means that while most of a user’s body movements will be reflected by their avatar, the direction that they are looking in won’t be. Furthermore, even if the Kinect did provide this information it would not really be useful as the user’s head would usually be facing straight ahead towards their computer monitor no matter where in the virtual environment they were looking. This meant that a special system was required for deciding how to animate a user avatar's head, based on where the user's view was directed in the virtual environment.

3.3.4 Presentations and the Display Screen

The act of giving a presentation to an audience in a meeting is naturally enhanced by user avatars being animated by the skeleton tracking capabilities of the Kinect for the reasons given above. It frees the user to behave naturally, rather than issue commands to their avatar to perform certain actions. This translates to performing a presentation in that a user simply needs to do the presentation in front of the Kinect camera in the same way that they would in a real world meeting, and their avatar would mimic their actions. This idea of a single participant performing a presentation to the other participants in a meeting lead the creation of a series of new potential features for the software in this project that would make performing such a presentation easier.

One of these potential new features could occur when a user is using a slideshow or some other similar visual aid as a part of their presentation. It was thought that it might be desirable to give that user a real world screen as a prop to do their presentation with; a corresponding virtual screen would exist in the virtual meeting environment that the user's avatar would stand in front of. The image displayed on the real world screen would be captured and transmitted to all connected clients to be displayed as a texture on the virtual screen in the virtual meeting room. The user would be able to point to things on or perform other gestures to their real world screen and their avatar would match those movements in front of the virtual screen.

3.3.5 Gesture Controls and Simulated Touch Screen

The idea for the linked virtual and real world display screen system lead to a handful of ideas for features that would allow the user who was doing a presentation to control the contents of the screen using gestures recognised by the Kinect. Initially it was planned that these gestures would be simple actions that allow control of the contents of the display screen in basic ways. This included functionality that would allow the user to zoom in on and pan across the image that was currently displayed on the screen using broad hand gestures. Later in the project a new possibility became apparent. It seemed that if the software was aware of the location and dimensions of the screen, then that information could be compared against the location of the users hand to give fine control over areas of the screen; essentially this meant that there was the possibility of creating a rudimentary touch surface out of any screen by using the Kinect.

3.3.6 Automatic Camera Positioning

Another potential feature was that was identified was the capability for the software to intelligently select the position and angle the user needed or wished the camera in the virtual environment to face. The idea behind this was to free the user from having to use the keyboard and mouse to look around the virtual meeting room. Instead the user would be able to sit back and pay attention to what was going on in the meeting, using subtle and natural gestures to change their view if necessary.

3.3.7 The Advantages of Virtual Reality

During the design of this project a common theme was to find ways of utilising the fact that the meeting takes place in a virtual world in order to do things that one could not do in a meeting that takes place in the real world.

An example of this lies with the camera controls. Instead of having their view of the virtual environment limited to the perspective of their own avatar, users have the option of taking control of the virtual camera and positioning it anywhere in the environment without shifting their avatar. This could be used by a user to get a better view of another participant while they spoke or see a close up of the contents of the virtual display screen without disrupting other participants and interfering with the meeting. This can be expanded upon by allowing a user to connect to the meeting with a second instance of the software (without enabling its Kinect related capabilities). The user could position this second instance to see the meeting from a different angle, essentially giving them two different views of the scene. A presenter could use this, for example, to simultaneously get a view of their audience and their self while they performed a presentation. Because the second instance would have no Kinect data, there would not be a duplicate avatar in the scene, meaning no disruption would occur.

The development history and technical details of all of the features listed above can be found in : Development & Implementation.

3.4 Building the Software

Early on there was a great deal of consideration given to how the main software for this project should be created. This included consideration of what platform to build the software on, whether it should be built from the ground up, or be based on some already existing software, and even what programming language would be best used to create the software.

3.4.1 Early Investigations

As this project was based principally on work done using the commercially available virtual world software called Second Life, there was some early consideration given to using Second Life as a platform to build on; the idea being to incorporate Kinect functionality directly into the Second Life application, allowing a direct comparison between the results of this project and the results of the project it is based on. Second Life has some limitations however that ultimately made it undesirable to attempt to follow this course. Among these was that there was no guarantee that it would be possible to incorporate skeleton tracking data from the Kinect to control a Second Life avatar in all the ways that were desirable to investigate. Additionally when it comes to matters of doing presentations in Second Life there are limitations on what can be transmitted between participants (for example, in order to use a slide show, a in game currency must be paid to upload the slides in the show) (Al Qahtani, 2010). Also, the fact that all activity in Second Life takes place on a persistent virtual world could have caused difficulties and disruptions when attempting to test any software. In particular any data that needed to be sent between individuals in the meeting needs to fit within the network protocols of the persistent world servers.

For these reasons it was decided that it would be better to build experimental software as a stand-alone application.

The next decision to be made was how to incorporate 3D graphics into the program if Second Life was not going to be used. Consideration was given to building the software as a mod to an existing video game. This would differ from the implementation using Second Life that was dismissed above in that it would not be dependent on online persistent-world servers populated by other users across the world. Such a game mod could use its own servers which meant that it would be possible limit people on those servers to those directly involved in the project, and that it would be possible to implement a custom network protocol to carry whatever data was necessary. Particular consideration was given to utilising the Source engine, a video game engine created by Valve Software. The idea was to use the graphics and network systems of the engine, and to incorporate Kinect related functionality into these systems. This approach was ultimately rejected due to it not being clear if it would be feasible to implement a system for animating characters from the skeleton tracking data provided by Kinect within the Source graphics system. The Source engine is not open source (Valve Software, 2007), so if there was some kind of underlying obstacle to allowing Kinect data to be used in this way, then it could have been extremely difficult to identify the problem and it may have been impossible to fix it.

3.4.2 Ogre

It was always clear that it would be desirable to base the software for this project on a system that would reduce the time spent programming graphics related code, as the goal of the project was not to investigate how to draw 3D graphics from scratch. With plans to use Second Life or the Source engine rejected, another solution had to be found. One possibility was the Ogre game engine. Ogre differs from the Source engine in that it is open source (Ogre), so if there were obstacles in implementing Kinect functionality it would likely be possible to isolate and fix these problems. The other advantage present in using Ogre rather than building the graphics from scratch is that graphics assets that are compatible with its systems are freely available on the internet(Ogre, 2012), which meant that time would not need to be spent creating graphics to represent users and objects in the virtual meeting space. Another advantage of using Ogre is that there were already demonstrations available on the internet, of characters within its animation system being dynamically animated based on skeleton tracking information from the Kinect (OpenNI, 2011). As a consequence of this some early work was done in looking at the Ogre engine in preparation for using it as the main graphics system for the project.

3.4.3 The Kinect SDK

Not long after the decision to use Ogre was made, Microsoft released its own SDK for developing Windows applications with Kinect support. This opened up a new possibility for software development. The Kinect software from Microsoft did not have as all of the same features as the software provided by PrimeSense, but it did have a few advantages (Hinchman, 2011)(Microsoft Kinect SDK vs PrimeSense OpenNI). The first of these was that its skeleton tracking system was more seamless to a user; when utilising the PrimeSense software, a user would be required to assume a calibration pose before skeleton tracking could begin. The calibration pose involved having the user stand up straight with the arms held out horizontally away from the body, and bent upwards into a vertical position at the elbow. The Microsoft provided software could begin skeleton tracking on a user as long as it could make out the users arms, legs, and head, regardless of what position they were in. Overall, the process of locking onto a user was faster and more reliable with the Kinect SDK. Another significant advantage of Microsoft’s Kinect SDK was that it was designed to be usable with the .NET framework which meant that the software for the project would be able to be written in C# (Microsoft, 2011). The C# libraries for the Microsoft’s SDK were simpler to use than the C++ libraries for both Microsoft’s and PrimeSense’s Kinect software. Using them would result in faster development of the features needed for the project.

3.4.4 XNA

Writing the project's software in C# provided another advantage. It would make it possible to utilise the XNA libraries (provided by Microsoft) to create the graphics. This meant that Ogre could be disregarded, along with any need to learn how to operate it. XNA is not a game engine per se, but a framework for building video games that is designed to handle much of the basic code for establishing 3D graphics (among other things) (Microsoft, 2010). The advantage of this system is that it provides the greatest flexibility to the programmer when creating the graphics for a program of any of the systems considered for this project earlier.

Ultimately it was decided that the final software for this project would be written in C#, utilise the Kinect for Windows SDK from Microsoft for interacting with the Kinect device, and use the XNA framework to handle the game-like aspects of the software (graphics and mouse/keyboard input.

3.4.5 Networking

There was still one large component of the program that needed to be considered. This was component would be responsible for the network communication that the program would need in order to transmit information between all of the participants in a single virtual meeting. It was quickly decided that it would be best to simply build a networking system from scratch, as the requirements of the software were fairly simple. All that needed to be done was to build a system that could exchange information between each participant in the meeting, and ensure that all data reached all participants. Consideration was first given to whether the system should use a distributed peer-to-peer model, where each instance of the software would have a connection to every other instance involved in a given meeting; each program instance would be alone responsible for ensuring that its information reached every other connected instance of the program.

In the end however, it was decided that a client-server model would be used. In this system, one instance of the program functions as the server for a meeting; all other participants function as clients, and must connect to the server to join the meeting. The clients only maintain one connection (to the server) and send and receive all data relevant to them across that connection. Each time a client is ready to send data to all of the other participants in a meeting it will send a copy of that information to the server. The server is responsible for ensuring that all information it receives is forwarded on to all other clients. The server is also responsible for sending its own data to each client.

The client-server model was selected over the distributed model for two reasons. Firstly it simplifies the system by requiring only one connection per client, rather than a connection from every client to every other client. This makes it simpler to establish a virtual meeting as each client only needs to know one IP address (the server’s). This also eliminates the possibility of failed connections between clients causing some clients to have only partial information about who is involved in the meeting. The second reason is that it makes it easier to implement a system where one person is responsible for running and managing the meeting. The server controls the state of the meeting and can issue commands to the clients (say for example, to select which client is currently doing a presentation, and thus controlling what is displayed on the virtual screen). Having a simple system for network communication is desirable because it reduces the amount of time spent creating and debugging it, leaving more time for the Kinect related parts of the software. The client-server model did carry the potential disadvantage of being more prone to lag, as it put a large proportion of the networking responsibilities onto a single computer with a single network connection. It was decided that the advantages of the client-server model outweighed the disadvantages, particularly as in an experimental setting, it would be possible to ensure the server has all of the resources it required.

3.4.6 Audio

In order to hold a meeting, users need to be able to talk to each other. This means that a method of transmitting audio between participants in a meeting would be required. Early in the project thought was given to including this functionality within the experimental software itself. In the end however it was decided that it would be best to use an independent third party program to handle audio communication during the user testing phase of this project.

The reason for this decision came down to the additional complexity that such a system would have added to the software. Thought would have had to been given to such things as how to capture and compress audio for transmission across the network, how to ensure that sound was reassembled from packets into a continuous audio stream reliably, and how to decide when to transmit at all (there is no point transmitting audio from a user when that user isn’t talking). These are problems that have already been addressed in existing software, and the potential audio related enhancements based on the Kinect’s microphone array were too few in number to justify the time that it would take to implement an audio system into the experimental software.

3.4.7 Real and Virtual Environments

In order to make progress with the project, it was necessary to make some decisions about the real and virtual worlds used for the experimental system. The virtual world created by the software, and the real world setting inhabited by the user need to be as closely matched as possible. When the user’s avatar is sitting at the meeting table in the virtual environment, the user should be sitting at their desk in the real world; and when the user’s avatar is giving a presentation in front of a virtual display screen, the user should be standing in front of a real display screen.

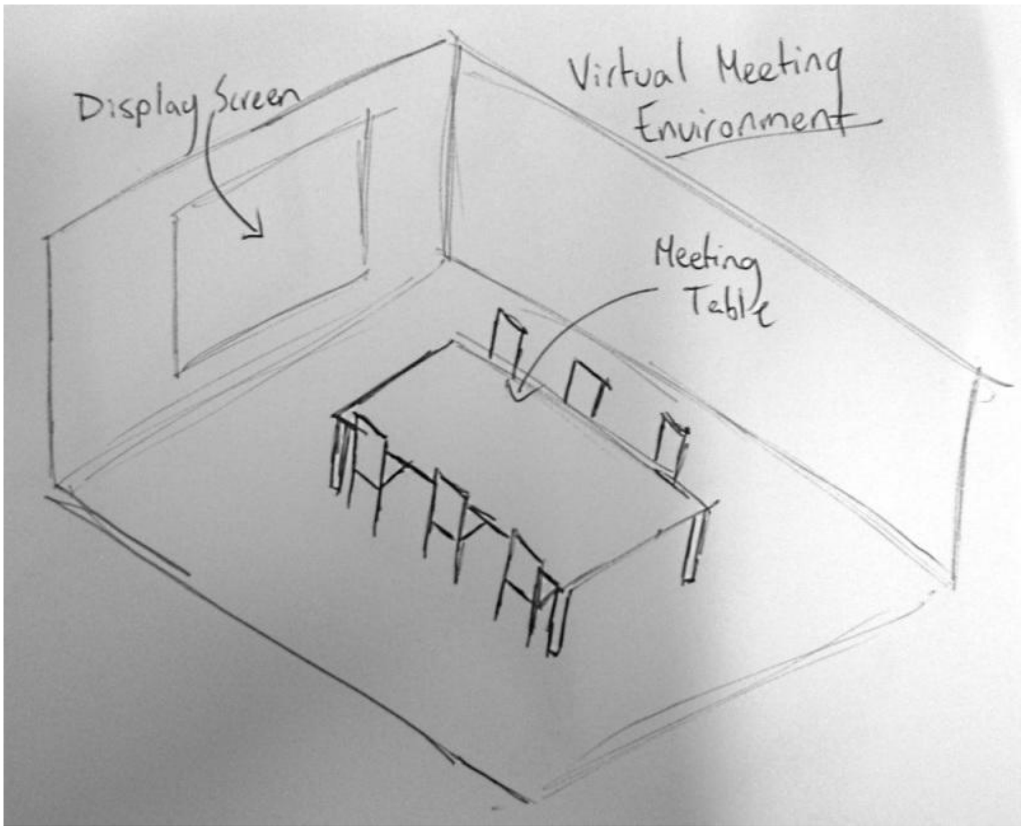


Figure 2: Virtual environment design

Figure 2 shows the design of the virtual environment. It consists of a room containing a rectangular table in the middle, with chairs along two sides. It is similar to the layout used in the Second Life experiment. Participants' avatars normally sit in the chairs during the meeting, and each user's view of the virtual environment is usually from the perspective of their avatar (i.e. a first person camera). At one end of the room there is a large surface that functions as a virtual display screen which can be used by a participant to display images (e.g. slides in a slideshow). There is space in front of the screen and provision is made for an avatar (the presenter) to move from their chair to this space; there they can move about and gesticulate. When a user is presenting, their view of the environment will directed from the screen down the table towards their audience.

Figure 3 shows the real world computer setup for use with the software. An ordinary computer can be seen with a user in front of a monitor with a keyboard and mouse. The user will see their view of the virtual environment on the screen; and be able to control certain aspects of the environment with the keyboard and mouse. A Kinect device is located above and behind the screen; this position ensures that the device’s cameras have as clear a view of the user as possible while still ensuring that the user is far enough away from the device to be beyond its minimum range for detecting them. When the user is sitting in a chair as shown in Figure 3 their avatar would be sitting at the virtual meeting table.

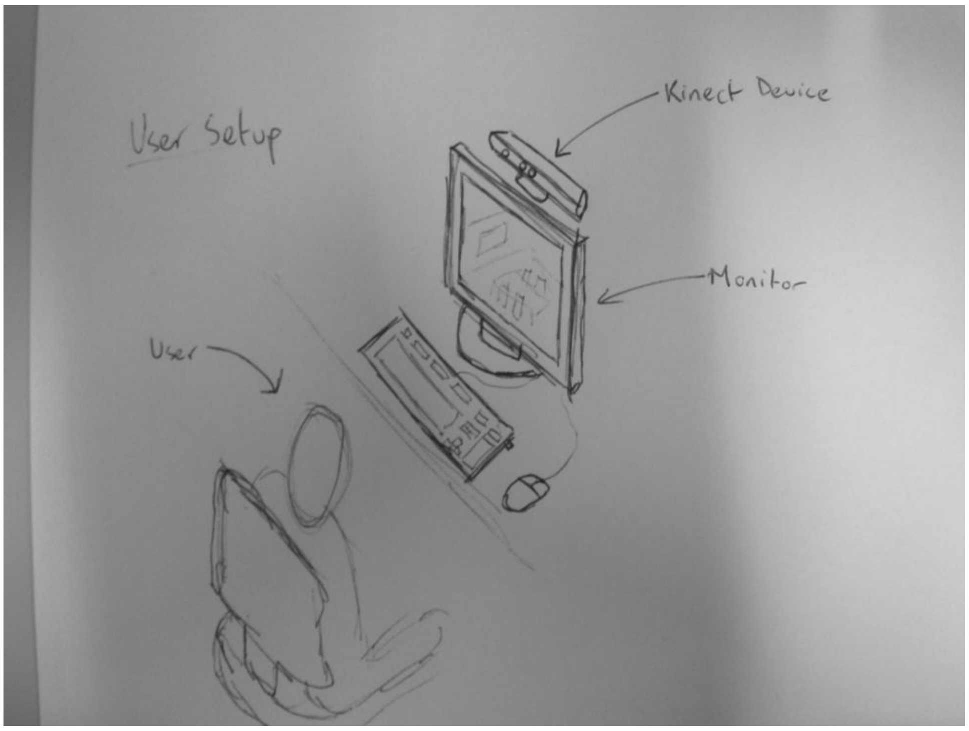


Figure 3: Real world user environment

Figure 4 shows the real world setup for a user doing a presentation. The setup is similar what is shown in Figure 3; the key differences being that the user is further away from the device and is not sitting down, and that there is a large display screen behind the user. The display screen corresponds to the screen in the virtual meeting room. When the user interacts with this screen, those interactions will be reproduced between the user’s avatar and the virtual display screen. The space between the real world display screen and the Kinect device corresponds (loosely) to the space between the virtual display screen and the virtual meeting table, so the presenter may move freely through real world space and their avatar will be able to do the same in the virtual space without intersecting with any virtual objects. The idea is that a user can move between seated and presentation modes simple by standing up and stepping back.

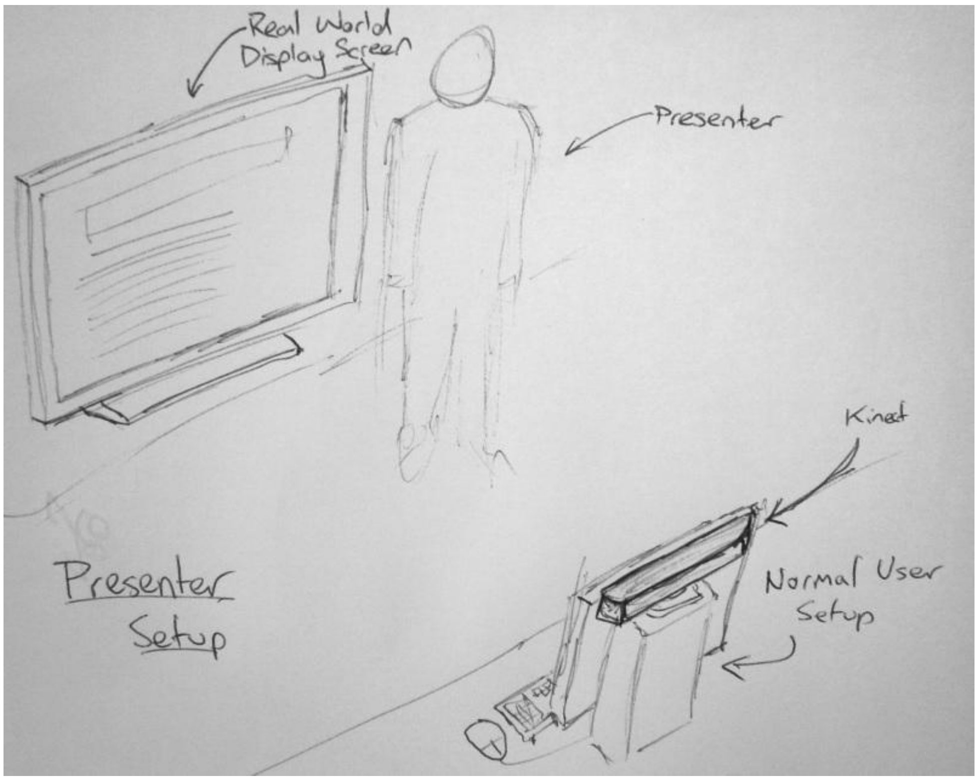


Figure 4: Real world presenter environment

3.5 User Testing

The final stage of the project involved running an experiment to test the software in order to discover how successful the features that were included in it were at serving their purposes in a practical setting. A number of ways of testing the software were considered before a final design for an experiment was decided upon.

There were two main formats for an experiment that were given serious consideration for use in this project. The first of these would have actually involved a series of experiments each of which would test one or more features of the program independently with different people at different times. For example, a single experiment might have involved having a single user sit down with the Kinect device and software set up on a provided computer, and that user would have been asked to complete one or more tasks relating to controlling their view of the virtual environment both using the keyboard and mouse, and using the automatic camera positioning system mentioned earlier. After carrying out the task the user would have been asked a series of questions on the experience, the purpose being to find out the usefulness of the automatic camera controls over conventional camera controls, and to search for further ways to improve them. Different experiments in the series would have involved different users and different features being tested.

The second potential experiment design that was considered called for a less rigid approach. Instead of depending on independent tests for all of the different software features, a single large test would be run involving multiple users testing all aspects of the software. The idea was to use the software as it would be used in the real world, with all of the participants involved in a single experiment conducting a meeting with each other. The participants would be instructed about the all of the features of the software and then asked to conduct a full meeting from beginning to end using the software. After they were done, the participants would answer questions about their experience using the software. The intention of this format of experiment is that the meeting that is being held between the participants is a real meeting, i.e. a meeting that would have been held in the real world even if it was not part of an experiment for this project.

The main advantage of the first of these two options was that it would ensure that every feature could be tested in detail, and that the participant in the experiment would be focussed on giving exactly what information was desired by the experimenter. The second option was ultimately chosen however as it provided several different advantages.

The first was that the second format is similar to what was done in “Virtual Worlds as Meeting Places”. This allowed a more direct comparison between the experience of a virtual meeting with and without using Kinect enabled features.

The second advantage is that it provides one piece of additional information about each of the various features that get tested; specifically, the relevance and overall usefulness of that feature. If the first option for testing the software had been chosen then each feature would have been tested explicitly, a single experiment would test a feature in detail, regardless of whether it was actually particularly useful in the context of an actual virtual meeting. In the second possible experiment format, if a feature was not useful would likely go unused. This would give accurate information about whether it was worth having a given feature at all.

The third advantage of the second experimentation method is that it provides information about the experience of a virtual meeting in general. It makes it possible to determine what the advantages and disadvantages of a virtual meeting over other forms of meeting (e.g. real world meeting in person, video conference, teleconference etc.). If any participant in the meeting has been involved in any other type of meeting, then they will be in the position to comment on what they feel are the advantages or disadvantages of holding a virtual meeting. The information gained about the general experience of a virtual meeting could be used in future to come up with way to further improve that experience.

The fourth advantage of this method lies in the fact that the participants are expected to be carrying out a real meeting with a purpose. This increases the likelihood that participants will be thinking about the features and experience of the virtual meeting software in a realistic context (e.g. when attempting to do something they would normally do in a meeting they might feel frustrated that they were unable to do it in a virtual meeting, or contrariwise satisfied with some feature that allowed them to do what they wanted). This means that the opinions received at the end of the experiment are more likely to be representative of individuals who would be using the software out in real world circumstances. The result of this is that the analysis of the results of the experiment will more likely to provide information that could be used to improve the experience of a virtual meeting in a real world situation.

The primary disadvantage of the second method of running an experiment is that it is more time consuming to carry out. Having the participants hold a real meeting may be useful for having them behave naturally when using the software, but it also means that much time is spent with the participants simply carrying out the business of that meeting, and not necessarily thinking about the virtual meeting experience. Ultimately it was decided that the advantages of the second method outweighed the disadvantages, which is why it was selected as the method to use in the user testing experiments for this project.

It is this decision to conduct the user testing for the software by holding actual meetings with multiple participants that necessitated the inclusion of the networking system for the software that was mentioned in the last section.

Chapter 4: Underlying Systems

The software created for this project (VMX) has relied on many software libraries and development kits. Chief among them, and central to the software's function have been: XNA provided by Microsoft, which provides the 3D graphics functions for the software; OpenNI which is an independent open source system for interfacing with the Kinect device, it was used early in the software's development; and the Kinect SDK which is Microsoft's official system for interfacing with the Kinect device. The official Kinect SDK only became available a few months after the beginning of this project, hence the use of OpenNI earlier in the project's development.

Starting with XNA this chapter will look at each of these systems in detail. A brief explanation of each systems origin and intended purpose will be given first; then their purpose within VMX will be discussed, along with details of that usage. The details given in this chapter will underpin the following chapter, which will discuss the implementation of the VMX software in more detail.

4.1 XNA

This project uses XNA primarily to handle the 3D graphics functions of the software. XNA is also used for several other purposes in the program. It plays a role in initialising the program, providing and calling various core methods that must be overridden by VMX. It handles the running of the main program loop. It also provides access to the data provided from the keyboard and mouse attached to the computer. Much of the information in this section is derived from the XNA Game Studio Documentation(Microsoft, 2010).

XNA (**X**NA is **N**ot an **A**cronym) was created by Microsoft in the middle of the last decade. It is a runtime environment designed to facilitate the creation of video games. It was made with the intention of making the game development process easier by providing much of the underlying code and functions that are often used in games, freeing the developer to focus on programming the systems that are specific to their own games (Microsoft, 2004). Another aspect of XNA is that it is designed to make it relatively simple to produce software that is compatible with several Microsoft products including: Windows, Xbox, and Windows Phone (Microsoft).

As XNA is targeted for game development, it may seem odd to use it for developing the software in this project (which is not a game). The reason for using XNA is that while VMX might not be a game, it does share several aspects with games. Specifically, VMX extensively utilises 3D graphics and takes keyboard and mouse inputs from the user to control aspects of this environment. Also, like a game the software is intended to still be actively doing things when not taking any user input, necessitating the use of a ‘game loop’ to control the program’s functions (in this report the game loop is referred to as the ‘program loop’ or ‘main program loop’).

It should be noted that Microsoft states that XNA is not tested for compatibility with the Kinect SDK (Microsoft, 2011). However, throughout the development of the VMX software, which uses both, no issue has arisen. Indeed there have been some issues with the simultaneous use of XNA and the .NET drawing libraries which are also used by VMX. All of these issues concern classes in the frameworks sharing names (for instance both XNA.Framework and System.Drawing contain classes named ‘Color’, ‘Point’, and ‘Rectangle’), this is easily remedied with appropriate ‘using’ statements to rename the offending types however as shown in Figure 5.

```
using DrawColor = System.Drawing.Color;
using XnaColor = Microsoft.Xna.Framework.Color;
using DrawPoint = System.Drawing.Point;
using XnaPoint = Microsoft.Xna.Framework.Point;
using DrawRect = System.Drawing.Rectangle;
using XnaRect = Microsoft.Xna.Framework.Rectangle;
```

Figure 5: Resolving conflicts between libraries

The full XNA system is very comprehensive including all kinds of functions and systems for video game development. VMX utilises only a handful of these systems, those being the: core program framework, the 3D graphics systems, and the keyboard and mouse input systems.

4.1.1 XNA Core

The core of any XNA program is the ‘Game’ class. Programs that use XNA are expected to have their central class inherit from Game. The Game class provides three abstract methods that are important to the function of an XNA program. These methods must be overridden by a programmer using XNA to insert their own program’s code so it can be called by XNA’s underlying framework. These three methods are called ‘Initialise’, ‘Update’, and ‘Draw’. In addition to these three there is another pair of methods that are less critical but work in the same way as the main three; these methods are ‘LoadContent’ and ‘UnloadContent’.

The Initialise method is called by XNA shortly after the program is started, after the Game and Graphics Device classes have been instantiated, but before LoadContent is called. It is expected by XNA that this method will be used for the initialisation of various aspects of the program. In particular this is where it is expected that services used by the program will be initiated, and any non-graphics related content will be loaded. The base Initialise method in the Game class also has the function of calling the Initialise method of any ‘Game Components’ (Game Components are a means XNA uses to allow modular systems that can be loaded and unloaded as their functionality is required to be included in a game), so a call to base.Initialize must be made from the overriding method to maintain XNA’s functionality. The Initialise method is called by XNA before the Draw method; the result of this is that nothing is displayed on screen until initialisation is complete.

The Update method is one of the most important methods in the entire system. This method is where most of the code that must be run on every iteration of the program loop is placed. Under normal settings, XNA does not leave the frequency with which this method is called entirely up to chance. By default XNA uses a variable called TargetEllapsedTime to control how frequently Update is called. When XNA is ready to call Update, the TargetEllapsedTime variable is checked against the actual amount of time that has passed since the last time Update was called. If the actual time passed is lower than the target time, then XNA will wait to call the Update method. Usually after Update is called the Draw method is called. However, if the update method takes so long to complete that by the time it is finished the actual elapsed time exceeds the target elapsed time then Update will be immediately called again. This has the effect of ensuring that Update is called at the required frequency even when ‘catching up’ from a slowdown. The cost of this is that it causes graphics frames to be dropped if the Update method is running too slowly. XNA does allow this system to be disabled, instead having the Update method run whenever the program is ready to do so.

Complementary to the Update method is the Draw method. This is method is usually called after the Update method (except in the cases as mentioned above) and is responsible for handling the updating of on screen graphics to reflect changes to game state made in the update method. Note that where a call to Update is delayed because not enough time has passed since the last call, the Draw method will still be called. Due to the potential discrepancy between the frequency if calls to the Update and Draw methods, it was important to ensure that VMX strictly kept all updates to program state in the Update method, and left the Draw method to deal with things that have no bearing on the progression of the game outside of drawing a single frame.

The LoadContent method is used by XNA to load resources used by the graphics system into the program. It is called from the Game class base Initialisation method. Additionally it is called at any time when graphics content needs to be reloaded (e.g. on a Device Reset event). Because it is called from the Initialise method, when first run at the start of the program it has to complete before the first call to the Draw method is made, meaning that it too will contribute to a delay between the program starting and graphics being drawn for the first time. Due to its function of loading in game content, the Load Content method makes heavy use of the XNA Content Loader, which will be discussed in the ‘XNA Graphics’ section of this document.

The Unload content method serves the opposite purpose to the Load Content method, unloading graphics resources when the call is made to do so.

4.1.2 XNA Input

The XNA framework contains an extensive library for taking user input into a program. A host of different devices are supported by the framework including:

game pads, keyboards, mice, touch surfaces, accelerometers, and microphones (though not Kinect sensors); exactly which of these is supported depends on the platform the program is running on (for instance, the accelerometer and touch surface systems only work on programs that are running on a Windows Phone 7 system). Of interest to this project are the systems for taking input from the keyboard and mouse. These systems both depend on the program polling the XNA libraries to obtain the current state of the mouse or keyboard (i.e. key presses and mouse movements do not fire events that a program can hook on to).

To use input from a keyboard XNA provides a data structure called ‘KeyboardState’. At any time a program may obtain a copy of the current state of the keyboard by calling the GetState method from the Keyboard class that is provided by the XNA framework; this method is static so there is no need to instantiate a Keyboard object. The primary purpose of a KeyboardState object is to provide information about the keys that are currently being pressed. It provides four different ways that can be used to retrieve this information, three of those ways are methods provided directly by KeyboardState: IsKeyUp, IsKeyDown, and GetPressedKeys; and the fourth way allows individual keys to be accessed directly. VMX only uses the IsKeyUp and IsKeyDown methods in its operation.

IsKeyUp is a function that is used to determine if a key is not currently being held down by a user. It works by taking a key’s identifier as a parameter and then returning a Boolean; if the key is being pressed then the method will return false, if it isn’t then it will return true. IsKeyDown functions in the same way as IsKeyUp but returns true if the key is being pressed and false otherwise. Keys are identified through the Keys enumeration (also provided by the XNA framework) that allows them to be easily accessed by name.

Access to the mouse works similarly to the keyboard. The XNA framework provides a Mouse class that has a static method called GetState that can be used to acquire a MouseState object which contains all of the data about the current state of the mouse. Unlike the Keyboard object, the Mouse object has a couple of extra public members besides GetState. These members are WindowHandle and SetPosition. WindowHandle is a property which contains a reference to the window that is currently being used for mouse processing (usually the single ‘game’ window of the program). The most import fact about this window is that coordinates for the mouse’s current position are reported in the MouseState relative to the top left corner of the window (i.e. the mouse cursor coordinates (0, 0) represent the position at the top left corner of that window). The SetPosition method can be used to programmatically reposition the mouse, the new coordinates are provided as parameters, and are also set relative to the top left hand corner of the currently set window.

The MouseState object is somewhat simpler than its Keyboard equivalent. It provides no special methods for retrieving information, only a series of properties that can be accessed. There are essentially two kinds of properties in the MouseState. The first of these are the button properties. These properties are names LeftButton, RightButton, MiddleButton, XButton1 and XButton2. All of these return an object called ButtonState. ButtonState is an enumeration that is almost identical to the KeyState enumeration; it differs only in that the states are named ‘Pressed’ and ‘Released’ instead of ‘Up’ and ‘Down’. The second type of property contains the coordinates of the mouse cursor. There is an X and a Y coordinate property both are provided as integers which give the number of pixels between the cursor and the top left corner of the currently used window. The coordinates will be negative for X when the mouse is to the left of the window, and for Y when the mouse is above the window.

4.1.3 XNA Graphics

The main reason for this project using XNA was for its 3D graphics libraries. XNA provides a suite of tools for accessing and controlling graphics hardware, loading graphics content into VMX, performing graphics related calculations, and drawing graphics on screen. XNA is based on Microsoft’s DirectX 9 but provides a convenient object based interface to that library.

The key components of the XNA graphics system are the GraphicsDeviceManager and GraphicsDevice. These classes are used by VMX to communicate with and control the graphics chipsets in a computer.

The GraphicsDeviceManager handles the configuration and management of graphics cards. It provides access to a GraphicsDevice object for each graphics card on the system. The GraphicsDeviceManager also provides several other services, a few of which are of interest in this project. In particular, it provides control of the size and shape of the back buffer. The back buffer is the ‘surface’ (area of memory on the graphics card) that graphics are rendered onto before being transmitted to the screen. The size of the back buffer is set using two properties in the GraphicsDeviceManager; these are the width and height in pixels.

The GraphicsDevice class itself has a large number of functions in an XNA program. It is responsible for creating graphics resources (textures, for example), creating shaders, and rendering 3D primitives. It also manages additional configuration information for various aspects of the graphics rendering process.

There are many methods provided in GraphicsDevice for drawing 3D graphics on screen. 3D graphics are made up of many graphics primitives which are essentially flat triangles that are oriented in 3D space. The primitives themselves are each made up of 3 vertices; vertices are points in 3D space that represent the corners of the triangles. The different methods handle different ways of providing vertices and connecting them together to form primitives. When drawing some piece of 3D geometry, the device will simply receive a list of all of the vertices that make up the primitives of that geometry. The device must decide how to assign these vertices into groups of three. There are two ways of doing this. The first is to simply read vertices off of the list three at a time and use each triple to draw a single primitive. The second way is slightly more complex; a second list is past in alongside the vertices, this list contains indices into the list of vertices. When drawing a primitive with a list of indices, the device will read three indices off of the list and then access the three vertices stored in the vertex list at the positions given by the three indices. These three vertices will be used as the corners of the primitive. The advantage of this method is that individual vertices in the vertex list can be referenced more than once, meaning that if several primitives have a vertex in the same place, then the data for that vertex for only needs to be included in the vertex list once.

VMX uses indexed primitives when it is drawing graphics. The reason for this is that all of the geometry in VMX is generated programmatically (i.e. not loaded in from external 3D model files). Separating the code for generating the positions of vertices from the code for linking those vertices together makes the algorithms for generating geometry more readable.

A single vertex can contain several pieces of information. Generally speaking, all vertices will contain at least one set of three dimensional coordinates as a vector. These coordinates give the position of the vertex relative to the origin point within the 3D graphics space. In addition to this a vertex may hold other information such as: texture coordinates, which tell the device how to map a texture over a particular primitive; a vertex normal, which is a vector that is most frequently used to decide what way a primitive is facing with respect to a light source; and colour data, which tells the device what colour a primitive should be drawn. When drawing a primitive the graphics card needs to know what information is stored in a vertex and what to do with it. This is done with a structure called a vertex declaration. The vertex declaration is a very important piece of information. It defines that format that the vertex data is provided in. In VMX, vertices contain a position vector, a normal vector and texture coordinates.

It should be noted that all vertex positions and normals are all stored in a XNA provided data structure called Vector3. As the name suggest a Vector3 object holds a three dimensional vector.

A critical part of the process of rendering the graphics in a program is the shader. The shader is responsible for taking all of the graphics data for a scene and actually transforming it into a 2D image that can be displayed on a screen. Shaders themselves are programs that run on graphics hardware, they are typically written independently of the program that uses them by in a specialised programming language (e.g. High Level Shader Language (HLSL))(Li, 2009). Shaders can be loaded into a program like any other graphics resource, however XNA also provides a prebuilt shader that is capable of rendering a scene with fairly ordinary graphics effects. This XNA provided shader is called BasicEffect. The BasicEffect shader is capable of a wide range of standard graphics effects, such as: vertex transformations, alpha blending (transparency), ambient light, diffuse colouring of objects, directional lighting (up to three independent lights), emissive colouring of objects, distance fog, specular lighting, and texturing. The BasicEffect shader object is created in the program by the graphics device; this is done by instantiating a new BasicEffect object, passing the device into the BasicEffect constructor as a parameter. The shader can be used to draw objects by loading it into the device before calling the DrawUserIndexedPrimitives method through the graphics device.

When drawing a piece of geometry one of the most important functions carried out is to transform the position of the geometry from its 3D representation in the vertex list in to 2D coordinates representing pixels on a computer screen. This is done by transforming the position vector (and if present, the normal vector) of each vertex using a series of matrices. There are three matrices used: a world transformation matrix, the purpose of this is to shift geometry that is currently been processed into its proper location in the scene (relative to other geometry); a view matrix which shifts all vertices from being positioned relative to an arbitrary origin point, to being positioned relative to the camera in the scene; and a projection matrix that transforms the 3D space positions of the vertices into 2D screen space coordinates that give the location of the vertex as is will appear on the computer monitor.

XNA provides a Matrix class that is used for all transformations. All matrices used by XNA are 4x4 so a Matrix object contains 16 values. The XNA Matrix class is an extremely powerful tool with a wide array of features, including methods for handling particular instances of a matrix, and static methods for generating standard matrices.

When dealing with an instance of a matrix there is one method of particular interest for VMX. It is the Decompose method. It is intended for use on world transformation matrices. When called it will return the individual components of the transformation, specifically the translation vector, rotation quaternion, and scale vector as separate structures. It particular use in VMX will be looked at in Section 5.3 .

The Matrix class also provides a series of static methods that allow new matrices to be generated easily from parameters. These include methods for creating translation matrices from a vector, several ways of creating rotation matrices many of which are used in this project (including rotations around a cardinal axis by a given angle, rotations around a given axis by a given angle, and rotation matrices generated from a quaternion), there are also methods for generating scale matrices from vectors, and a handful of ways of generating perspective matrices. In addition to these methods for providing new matrices, the Matrix class also provides functions for performing various operations on existing matrices; these include methods for adding matrices together or subtracting them from one another, dividing or multiplying the components of a matrix by scalar values or the components of other matrices; a matrix can be inverted, negated or transposed; and two matrices can be linearly interpolated. All of these functions produce a new matrix as a result.

The Content Loader is a system provided by XNA for managing external graphics content and loading data from external files into a program. It is capable of loading in data from a wide variety of different file types (e.g. textures, 3D models etc.) into an appropriate data structure for its kind of data. XNA calls the data items that are loaded ‘Graphics Resources’. Resources can also be generated programmatically. VMX uses the content loader to load static textures. It loads these textures from ordinary image files (typically PNG format files). Within VMX the Content Loader does one other thing, it loads in a SpriteFont. The SpriteFont is used when drawing text on the screen.

Textures in particular are used extensively in VMX. XNA provides a class for managing textures called Texture2D. Texture2D provides both a data structure to hold texture information, and a tool for manipulating that data. Texture information is represented as a 2D grid of ‘texels’. Texels in a texture are much the same as pixels in an image. Once filled with texture data, a Texture2D object can be passed directly into a shader where it will be used to texture colour in geometry when a 3D scene is being rendered.

VMX utilises several different ways of getting data into a Texture2D object. The first of these is by Texture2D.SetData. This method has a few overloads but only one is used by VMX. The version of the method that VMX makes use of takes a two dimensional array of XNA framework Color objects. The two dimensions of the array directly correspond to the two dimensions of the image, meaning that the location of each Color object in the array corresponds directly to the location of the texel it represents in the texture. This method is used when VMX generates the contents of a texture programmatically (e.g. when converting an image from the Kinect video data stream into a texture). Another method used by VMX is called Texture2D.FromStream; this loads data into the texture from an appropriately formatted image data stream (Kinect data cannot be loaded in this way as it is not provided by the Kinect runtime in an appropriate format). VMX only uses this method when receiving data from a screen capture operation. The final method used by VMX for loading in a texture is by using the Content Loader.

4.2 The Kinect SDK

The first beta version of the Microsoft Kinect SDK 1.0 was released in June of 2011(Peckham, 2011). It is free to use for non-commercial use. It provides the APIs and systems necessary to build software for Microsoft Windows that utilises the Kinect sensor. It was adopted for use in the development of software for this project shortly after release. The structure of the libraries and the means by which the Kinect sensor data is processed and provided differs significantly from the structure used in OpenNI. The Windows compatible version of the Kinect SDK has limited functions compared with the features provided by the Kinect on Xbox. In particular, it lacks the gesture and facial recognition systems.

The second beta version was released in November of 2011 (Clayton, 2011). It provided only a handful of miscellaneous improvements such as a new event that fires when the status of the Kinect device changes and improvements to the skeletal tracking system (Stott, 2011). Few of the improvements had much effect on VMX; nonetheless VMX was updated to function with this version. The final release of version 1.0 of the SDK was released in February of 2012 (Cangeloso, 2012); this included a series of new and updated features but due to the closeness of its release to the completion of this project, VMX was not updated to run on this new version.

Unless otherwise noted the information in this section was acquired from the Kinect for Windows SDK documentation (Microsoft, 2011).

4.2.1 Image Data Streams

First and foremost, the Kinect SDK provides access to the raw data stream from each of the Kinect device’s on board sensors. A colour video stream can be accessed from the RGB camera, a stream of depth data can be obtained from the depth sensing camera system, and an audio stream can be acquired from the microphone array on the device. When first initialised various parameters can be passed into these streams to control aspects of the data they provide.

The Kinect camera itself actually captures a 1280 x 1024 Bayer colour image at 30 frames per second. This data is compressed and converted to an RGB image before being transmitted over USB to the Kinect runtime on the host computer. Without compression it would not be possible to transmit image data across the USB connection fast enough to maintain 30 fps. The trade-off is a reduction in image quality. The colour video stream that is then provided by the runtime can be configured in several ways. The resolution of the video frame can be configured to one of three available settings: 80 x 60, 320 x 240, and 640 x 480. The number of image frames from the camera that can be buffered at once is also customisable. It can be set as high as four. The other main setting that can be changed is the format in which the image data is provided. There are three possible settings available for this. The data can be provided as 32-bit per pixel image formatted in sRGB colour space. Only the lowest order 24-bits of each pixel’s data contain useful information with 8-bits each for the red, green and blue channels. The image can also be formatted as a YUV colour image with 16-bits per pixel; this format saves memory by only taking up half as much data per pixel. When using a YUV format image, additional limitations are placed on the image stream: it must be using a resolution of 640 x 480, and it must be running at 15 frames per second. The third type of image stream available is also RGB but first converted from a YUV version of the image. VMX uses the 32-bit sRGB format as it makes translating the video data into a texture straight-forward.

The depth camera stream also has a series of configurable settings. Like the colour video stream is can be set at three different resolutions (the same three as the colour video). It also shares the configurable frame rate and frame buffer size. The depth data however is provided in different formats from the colour image data. There are two different depth data formats. Both of these formats have 16 bits per pixel.

The first format is the depth data only setting. In this format the low order 12 bits of each pixel give the distance to the nearest object visible at that pixels coordinates in the sensors field of view. The distance is measured in millimetres. The value does not represent the distance from the point where the sensor is located to the point on the object; rather it represents the distance on the z axis between the (x, y) plane that the sensor sits on and the object. This means if the sensor was facing directly at a flat wall, the entire wall would be represented with the same distance values, instead of the distance increasing for pixels that were closer to the edge of the image. The other four bits of the pixel's data are not used.

The second setting for the depth stream format includes additional data in the form of player segmentation data. Player segmentation data is a bitmap where each pixel gives the user ID of the user that appears in that pixel on the depth image. Within the internal workings of the runtime, the player segmentation data is treated as a separate data stream from the depth data, however in practice it is only available from outside of the runtime when integrated into the depth stream. Each pixel in the combined depth/player segmentation data is still represented by 16-bits. The 3 lowest order bits represent the user ID of the user that appears in the depth pixel for that image. If this value is zero then no user appears in that pixel. The high order 13 bits represent the distance to the object in that pixel, in the same way as the plain depth data format does. When using this format, the resolution of the depth image is limited to a maximum of 320 x 240.

It is worth noting that when providing the depth stream data to a program, the runtime provides it as a byte array. Each pixel uses two bytes in the array, but the bytes are arranged so that the byte that has low order bits of the pixel is placed in the array at the position before the byte that has the high order bits. This order needs to be reversed before the bytes can be stored into a different type of variable. Also of note is that if the depth value for a pixel is ever zero, it means that Kinect is unable to determine the distance to the object in the pixel. Typically this is caused by the object that is in that pixel, being either too close to or too far away from the sensor.

The Kinect SDK provides two ways for a program to actually acquire this data from the runtime. The first is the polling method. When using this method a program will call a method in the Kinect libraries that will provide the latest available frame available for the requested data stream. The runtime will never provide the same frame more than once however, so it is up to the programmer to decide how the retrieval method should behave if a new frame is not available. This is done by providing the retrieval methods with a number that represents the maximum number of milliseconds to wait for a new frame before returning (essentially a time-out). By setting this value to zero the retrieval method can be made to immediately return when there is no new data, allowing the thread to carry out other tasks in the meantime. The time out value can also be set to infinity, which causes the retrieval method to block until a new frame is available.

The second way to retrieve data is using events. When using this method the programmer writes their own event handler that is set up to do whatever needs to be done when new data is received from the runtime. This event handler can then be hooked into the runtime’s DepthFrameReady or ImageFrameReady events. The runtime will then trigger the appropriate event when new data becomes available, and pass on that data into the event handler provided by the programmer. This method saves the programmer having to manually poll the runtime for data. Only one of the two ways of retrieving data can be used at any one time, a single program cannot use the polling and event methods simultaneously.

4.2.2 Skeleton Tracking

The Kinect runtime is capable of tracking the position and orientation of two users standing in front of it. The system uses the depth data received from the Kinect to do this. A 20 “joint” skeleton is inferred based on the image of each user as seen by the depth sensor. Each joint in the skeleton has a set of coordinates that give its position in 3D space. The skeleton data can be provided from the Kinect runtime to a program in the same ways that the depth and image frames are provided: either through the program polling the runtime to acquire the latest data, or by an event handler being hooked into the SkeletonFrameReady event in the runtime.

The skeleton data is computed directly from the data in a depth image frame. A new skeleton frame is generated whenever the Skeleton Tracking Engine in the runtime finishes processing a depth frame. This occurs even if a depth frame has no users in it, thus it is possible for a skeleton frame to contain no actual skeleton data. A skeleton frame includes within it the timestamp of the depth frame that the skeleton frame was generated from, allowing the skeleton frame to be easily matched with its corresponding depth frame.

The skeleton tracking engine provides two forms of skeleton tracking: active tracking, and passive tracking. The engine can handle two simultaneous actively tracked skeletons; an actively tracked skeleton contains complete data for the skeleton; this includes all of the joint positions that could be determined, either by looking at the depth data directly or by being inferred from a combination of other joint positions and the depth data. In addition to the actively tracked skeletons, the engine can pick up and provide passive tracking for four additional skeletons. Passively tracked skeletons contain no data for individual joint positions in the skeleton, only a single coordinate representing the centre of mass of the skeleton (a centre of mass coordinate is also provided for actively tracked skeletons).

A single skeleton frame always contains an array of exactly six skeletons. Each of the six can be tracked actively, passively or not at all. The runtime guarantees to always keep a particular user’s skeleton in the same place in the array for as long as that user stays in view of the depth sensor. This means that when accessing the skeleton for a particular user, it is not necessary to check through all of the skeletons to find the one that has the user ID associated with that user; instead the index of the skeleton for that user in the skeleton array simply needs to be remembered.

4.2.3 Coordinate Systems

The Kinect SDK utilises several different coordinate systems. Each defines the location of points in their respective data streams. There are three different systems: Skeleton Space, which is used for skeleton and joint positions; Depth Image Space, which is used to define the location of objects in the depth image; and Colour Image Space which defines the location of points on the colour image stream.

The skeleton space system is a full 3D Cartesian system, with points defined with respect to a single origin. The Kinect SDK places the Kinect sensor array at the origin point when defining the coordinates of objects. The positive Z axis moves outwards directly in front of the sensor array, the positive Y axis goes directly upwards, and the positive X axis moves off to the left when viewed while facing in the same direction as the sensor array. It should be noted that if the sensor array is on a lean, either from being on a non-level surface, or from being tilted on its pivoting stand, then the direction of the Y axis may not be perpendicular to the floor. This can cause users who are standing up straight in the real world to appear on a lean in skeleton space.

The Figure 6 illustrates this:

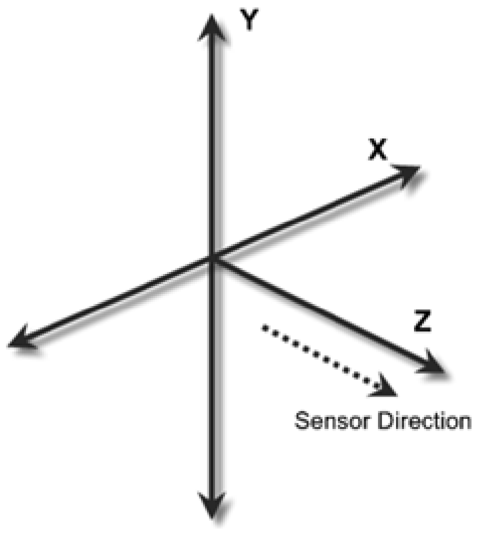


Figure 6: Skeleton Space coordinate orientation(Microsoft, 2011)

The units of the coordinates in skeleton space represent distances in metres in the real world, for instance if a skeletons centre of mass had a Z coordinate if 1.0, then we would know that the user corresponding to that skeleton was standing one metre away from the Kinect sensor array.

Every time the runtime processes a new skeleton frame, it will attempt to provide an estimate of the plane of the real world floor in skeleton space. It provides this estimate as a Vector4 where each value in the vector serves as a coefficient in an equation for a plane in 3D space. The equation is as follows:

$$Ax + By + Cz + D = 0$$

Equation 1: Equation for a plane in 3D space

In this equation the four values of the Vector4 are places into the equation like so: A = Vector4.x, B = Vector4.y, C = Vector4.z, D = Vector4.w. Note that the values in the vector will be scaled so that D represents the distance between the sensor array and the floor, in metres.

Depth Image Space is the coordinate space used for describing the location of objects in the depth image. While still technically three dimensional, the system differs significantly from the skeleton space system in that the units on the equivalent of the x and y axes do not correspond to real world distances. Rather x and y coordinates refer to pixels on the depth image. The equivalent of the z axis does however relate to real world distance in that it gives the number of millimetres between the plane that sits of the x and y axes and the object that appears in a given pixel. Figure 7 illustrates how the distances given in the depth image relate to the Kinect sensor itself:

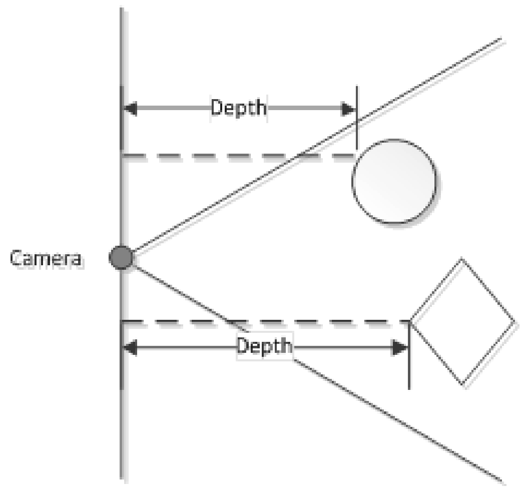


Figure 7: Depth space measurements (Microsoft, 2011)

It is critical to note that the distances given by the depth image *do not* correspond to the straight line distance from the Kinect sensor to the object. The distance given represents only how far *in front* of the camera the object is. The x and y coordinates for a point on the depth image are given by the runtime as a value between 0 and 1, with (0, 0) being at the far left and top of the depth image, and (1, 1) being at the far right and bottom of the image.

The third coordinate system is used by the Kinect for identifying points on an image frame from the colour camera. Unlike the other two systems, it is two dimensional. It uses only an x and a y axis. These axes correspond to the x and y axes in depth image space. As in depth image space, the units on those axes do not correspond to real world distances. However, where depth space values would range between 0 and 1, colour image coordinates use pixels as their units; thus a colour image will have coordinates ranging from 0 to the maximum resolution of the image (e.g. 0 to 480 on the y axis, and 0 to 640 on the x axis for a 640 x 480 resolution image). When normalised for each other, coordinates of an object on the depth image, and coordinates for the same object on the colour image will be similar but offset from one another due to the physical separation of the colour and depth cameras on the device. The degree of offset will depend on the distance from the Kinect sensor to the object, with more distant objects appearing to have a less significant offset than objects that are closer.

The Kinect SDK provides a number of ways to translate coordinates between these different systems. There are two methods for translating skeleton space coordinates into depth image space coordinates. Both of these methods take a Kinect vector representing a set of skeleton space coordinates, and a series of output parameters into which the results of the conversion are placed. Both methods return a value between 0 and 1 for the x and y coordinates of the pixel corresponding to the given skeleton space coordinates. The difference between the two methods is that one of them also has a third output parameter; it is a short value that represents the depth value stored at the point on the depth image corresponding to the provided x and y coordinates. Note that the depth value provided may not be the actual depth of the point indicated by the skeleton space coordinates. It is possible that another object could be between the Kinect sensor and the skeleton space point, thus the depth value at the corresponding point in the depth image may be for an object in front of the skeleton space point in the camera's field of view.

The Kinect SDK is also capable of translating from depth space coordinates to skeleton space coordinates. The method for doing this takes three parameters and returns a Kinect vector. The three parameters are the x and y coordinates of the depth space point (as float values between 0 and 1), and the depth image space depth value for the coordinates (as a short representing the distance in millimetres).

The final transformation the Kinect SDK is capable of is the translation of depth space coordinates into colour image space coordinates. The method provided for this takes a large number of parameters. The first is the resolution of the colour image. The second is the view area of the colour image camera. The view area is a structure that holds information about the current pan and view settings of the Kinect camera. Every image frame provided by the runtime includes a view area field populated with the correct data for that image frame. The next parameters to the method are the depth space x and y coordinates. Unlike the conversion from depth space to skeleton space, these coordinates must be provided as integers, not floats between 0 and 1. To convert the coordinates into meaningful integers, the float values are multiplied by the resolution of the depth image, as follows:

$$x_{int} = x_{float} \times depthImageWidth$$

$$y_{int} = y_{float} \times depthImageHeight$$

Equation 2: Converting depth space coordinates from float to int

The fifth parameter is the depth value stored in the depth image pixel corresponding to the accompanying x and y coordinates. It is provided as a short value. The final two parameters are the output integers used to return the image space x and y coordinates of the pixel in the colour image that corresponds to the provided depth coordinates. These values are integers which indicate the row (x) and column (y) of the pixel they represent.

The process of converting coordinates from 3D depth image space to 2D colour image space is not reversible. This is because information is lost in the conversion. There is no way to determine from the colour image coordinates alone, how far an object in a certain pixel is from the Kinect sensor, thus the position offset between the object in the depth image, and in the colour image cannot be determined. For this reason the Kinect SDK provides no methods for converting from colour image space back to skeleton or depth image space.

4.2.4 SDK Structure

The Kinect SDK employs several data structures that are used to pass around information. VMX utilises several of these structure extensively in its handling of data from the Kinect.

The full data for a single skeleton is stored in a structure called SkeletonData. This is where the collection containing all of the data for all of the joints is kept; SkeletonData also contains the skeleton's position, user ID, and tracking state (active, passive, or not tracked). In addition there is a Tracking ID which gives an ID corresponding to the one used in the player segmentation data in the depth frame for the user that the skeleton represents. Also provided in SkeletonData is a value called Quality. Quality indicates what parts of the user are actually visible to the Kinect camera; it will indicate if part of the user is out of the depth camera's frame to the left, top, right, or bottom. This information can be useful for setting the tilt of the Kinect sensor array.

The Camera class is used by the SDK to provide data and allow control of the Kinect’s camera. It is this class that provides the method for converting coordinates from depth image space to colour image space. The other notable feature of this class is the ElevationAngle property. It is this property that is used to control the tilt of the Kinect sensor array. When ElevationAngle is changed by VMX, an attempt is automatically made by the Kinect hardware to adjust its current tilt to match the new angle. The Kinect’s tilting mechanism has a range of 54 degrees; allowing movement of up to 27 degrees either up or down from the horizontal position. The Kinect’s tilting mechanism was not designed with the intention of it being used to change the Kinect’s angle on an extremely frequent basis. If too many commands to change the elevation angle are made in too short a space of time, then the Kinect runtime will throw an exception and the change will not occur. The exact limits on changes to elevation angle are: no more than one change can be made per second and no more than 15 changes per 20 seconds. Also of note is that when ElevationAngle is changed, sometimes the mechanism will not match the provided angle perfectly and may be one degree out. If this does occur the Kinect runtime will update the value stored in ElevationAngle to accurately reflect the true angle.

The Joint class provides all of the data for a given joint. It contains three members, all of which are properties. The first is the ID which is a value that indicates which of the 20 points the Kinect picks up on a user’s body this joint represents (e.g. Spine, HandLeft, Head etc.). The ID is of the type JointID which is an enumeration which links joint names to the index of their usual position in a skeleton’s joint collection. The Joint class also contains a Kinect Vector which represents the position of the joint in skeleton space. The final member of the Joint class is the TrackingState, which indicates how the joint’s position is being determined. There are three possible values for the tracking state: Tracked, Not Tracked, and Inferred. ‘Tracked’ indicates that the joint is currently in view of the depth sensor and its position is known. ‘Not Tracked’ indicates that the joint cannot be seen by the sensor and its current position is not known. ‘Inferred’ is a special case, it indicates that position data is available for a joint, but that joint is either out of the sensor’s field of view or is obscured by another object. To acquire the position, the Kinect runtime looks at the position of ‘Tracked’ joints and uses that information to infer the position of the non-visible joint.

The Kinect SDK represents 3D coordinates with its own Vector structure. The Kinect’s Vector structure is actually equivalent to an XNA Vector4 with space for an X, Y, Z and W coordinate. Typically when used, the X, Y, and Z coordinates correspond to their respective axes in skeleton space, with W unused. The W coordinate is used in some other situations where the Vector is being used to carry information beside coordinates (for instance, when the runtime is providing its estimate of the location of the floor plane).

The image frames that are provided to event handlers for a DepthFrameReady or ImageFrameReady event store the actual image data they contain inside a special structure called a PlanarImage. This structure is simple, containing four fields. The first field is called Bits and is a byte array containing the data for the image itself. BytesPerPixel is the second field and is used to determine how many bytes in the ‘Bits’ array are used to represent a single pixel in the image. The other two fields are the width and height (in pixels) of the image. Those last three fields are necessary for a program to be able to convert the single dimensional array into a two dimensional image.

The Runtime class is the most important class in the entire Kinect SDK, it is through this class that a user program interacts with all parts of the SDK (except for those part relating to the Kinect audio capabilities). It is this class that provides the events that a user program can hook into to receive Depth, Video and Skeleton frames from the Kinect. It is also provides direct access to the video and depth streams so they can be manually polled for new frames. Another noteworthy part of the Runtime class is the Status property which gives information on the status of the Kinect sensor itself, able to report if it is connected, disconnected, not powered, not ready, or if there has been some other error. The Runtime class also provides direct access to an instance of the Camera class. The Runtime provides two methods, Initialise and Uninitialise. These methods are respectively responsible for setting up and shutting down all of the runtimes subsystems. The final major purpose of the Runtime class is to provide access to the Skeleton Engine.

The Skeleton Engine class is responsible for managing the skeleton tracking system. It is the class that provides the methods for converting coordinates between skeleton space and depth image space. It also provides the method used by a user program to poll for new skeleton frames. Its other main function is to provide control of skeleton smoothing operations. Skeleton smoothing is a way of reducing apparent jitter in the position of individual joints in the skeleton. There are several customisable parameters related to smoothing, the skeleton engine maintains the current settings for these parameters in a data structure called TransformSmoothingParameters.

TransformSmoothingParameters contains five properties, each affecting some part if the skeleton smoothing process. The first is called Correction; it is a float value between 0 and 1 that affects how quickly the smoothed position of joints will change in response to changes in the position in the raw data. Low values will cause adjustment to be slow, which results in smoother movements in the skeleton, having the trade-off that a user's movements may appear to lag behind their actual movements, and that fine movements may be filtered out entirely. Higher values for this parameter have the opposite effect; more responsive movement with the trade-off of having more jitter. The second parameter is called JitterRadius; it is also a float. It represents a radius (in metres) which limits the distance that a joint may move in a single frame. The purpose of this is to prevent jitter from causing joints to jump around extensively. The third parameter is called MaxDeviationRadius. The purpose of this parameter is to limit how far away from the raw data the smoothed data can be. It too is a float value representing a radius in metres. If a smoothed data point would be outside this distance from its equivalent raw data point, then it will be clamped to this distance. The fourth parameter is called Prediction. It is a float that determines how many frames are predicted by the smoothing algorithm (i.e. how far ahead the smoothing algorithm should extrapolate movements when factoring them into smoothing). The final value is called Smoothing; it is a float between 0 and 1 which simply determines how much smoothing should be carried out. A value of 0 for this parameter will cause the algorithm to return the unchanged raw data. Increasing the value increases the effect of smoothing on the data, subject to the same trade-offs mentioned in regard to low Correction values.

Chapter 5: Development & Implementation

The software that has been created as a part of this project is called VMX (Virtual Meeting XNA). It is the successor to VirtualMeeting which was the early test bed that was used in this project for early research into the Kinect device’s capabilities. VirtualMeeting utilised the OpenNI framework to interact with the Kinect device and was programmed in C++. Unlike its predecessor VMX is programmed in C# and is built on the XNA libraries for graphics, and the official Microsoft Kinect SDK for interacting with the Kinect device.

This chapter will give the details of the development and implementation of each of the features of VMX. It will include a broad overview of the structure of the program, and detailed descriptions of the structure of each system within the program. It will also include detailed descriptions of all of the important algorithms within the system, including a discussion of the problems and obstacles that were encountered as the each algorithm was developed.

The chapter starts in Section 5.1 with an overview of the whole program describing its overarching structure. After that the chapter moves on to Section 5.2 which discusses the details of how the program interacts with the Kinect runtime in order to retrieve information from, and issue control commands to, the Kinect device. Next, Section 5.3 looks at the graphics system is described, including the key graphics classes and overall implementation of the system. After that, the chapter moves on to Section 5.4 which goes over the various aspects of the user avatars that are used within VMX, including how they work with the data from the Kinect and link into the graphics rendering system. This section also includes a history of the evolution and improvement of the user avatars over the course of the project. In Section 5.5 the various types of gesture recognition used by the project are discussed. This includes discussion of ways to recognise gestures based on finger positions, despite the fact that the Kinect runtime offers no explicit support for finger tracking. Section 5.6 talks about the features and functions of the display screen that is present in the virtual meeting room and how it can interact with a corresponding real world display screen. Section 5.7 looks at the camera control systems including details of the implementation of both the manual and automatic systems. The chapter concludes with Section 5.8 which discusses the networking system within VMX, including details of the network protocol used.

5.1 VMX Structure

Broadly speaking, the classes that make up the VMX program can be divided into four groups: Core, Graphics, Network, and Kinect.

The Core group is made up of a single class. This class is called VMX and is the central class in the program. All communication between the other three groups goes through the core. The VMX class is a sub class of the XNA Game class. Game is class that is provided by the XNA libraries. This is the class that actually contains the code that provides the program’s main loop, however it is the VMX class that contains most of the logic that is executed in that loop. Throughout this chapter the VMX class will be referred to as ‘the VMX core’ or just ‘the core’ to differentiate between it and the program as a whole.

The Graphics group contains all of the classes that are specific to the graphics engine in VMX. It is the largest of the groups, containing ten classes. Most of the graphics classes inherit from the abstract Drawable class; these classes mostly correspond to particular kinds of objects that appear in the virtual environment, with differing classes used for objects that require particular graphics functionality. Despite the size of this group much of the control of the graphics engine lies within the core class, mainly due to the close relationship between graphics and the XNA Game class.

The Network group is the second largest of the groups and contains the classes that are responsible for communication between separate instances of the VMX program across multiple computers. The Network group is more autonomous than other groups. Some of its classes utilise threads that run separately from the main thread that the VMX core runs on. Very little network specific code is contained in the core class and the classes in the network group are designed to make the details of network communications as transparent as possible to other parts of the program.

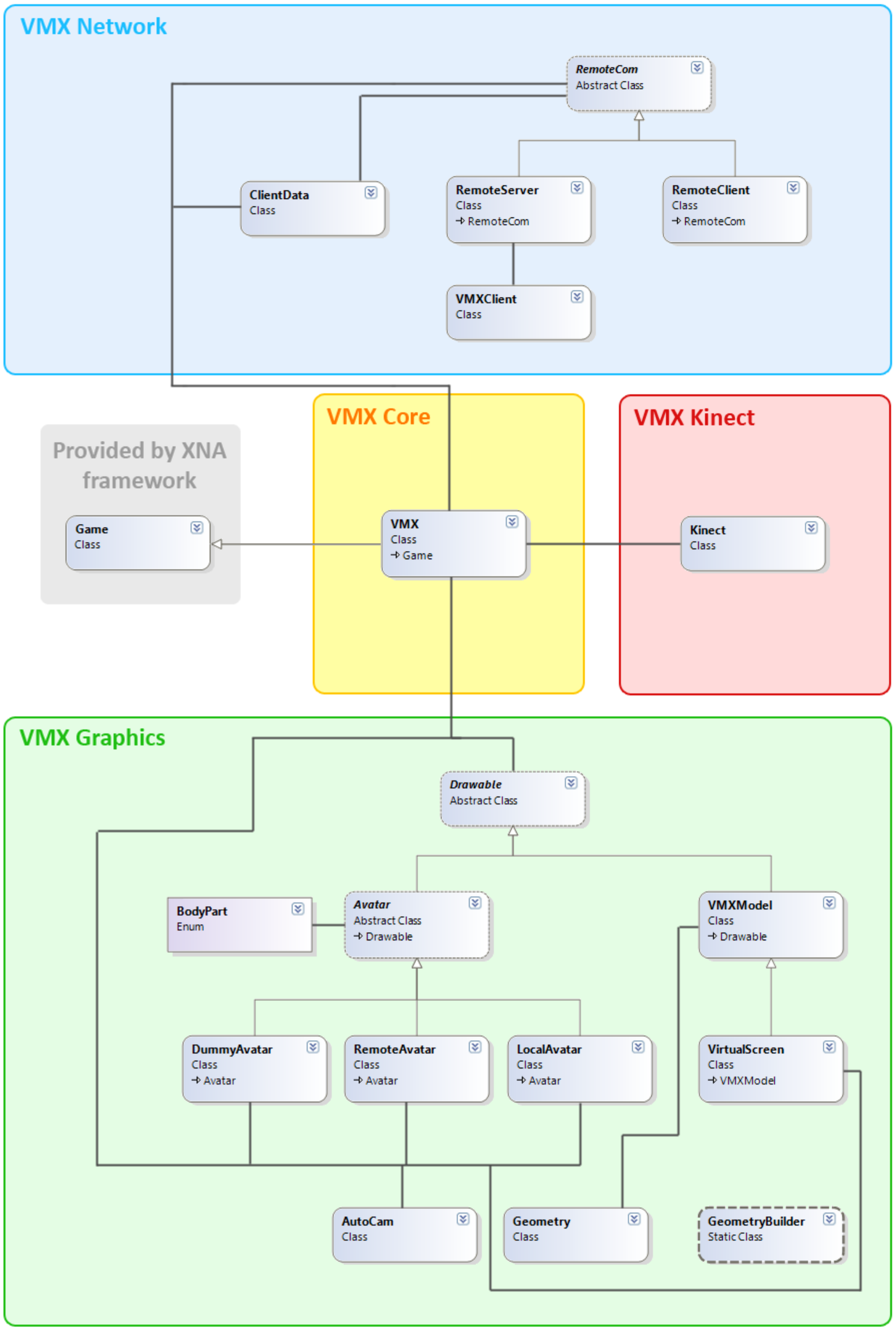


Figure 8: VMX program structure, with respect to the core class.

The Kinect group is another one-class group; it is responsible for providing access to the Kinect runtime. The Kinect group only provides methods for configuration as most of the data from the Kinect device is passed directly from the runtime to the core class by use of events.

Figure 8: VMX program structure, with respect to the core class. Figure 8 shows the structure of the whole VMX program, showing each group and the classes it contains. The diagram does not show all relationships between classes; only inheritance and relationships that demonstrate how a class relates to the VMX class are shown. More detailed diagrams of the individual groups can be seen in the following sections.

5.2 Kinect

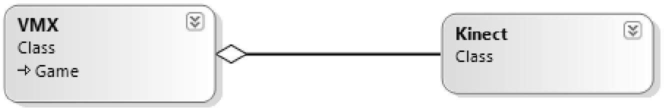


Figure 9: Kinect Class Structure

5.2.1 Initialisation

The VMX Kinect class handles many of the interactions between VMX and Kinect; although as noted before, the Kinect runtime itself directly communicates information back to the VMX core using events. VMX does not require a Kinect device to be plugged into the computer in order to run. This makes it possible for users who do not have a Kinect to join a meeting and watch what is going on. It also allows the instance of VMX that is functioning as server to run on an independent system with no Kinect device or user. It takes an instruction from the user to initialise a connection to

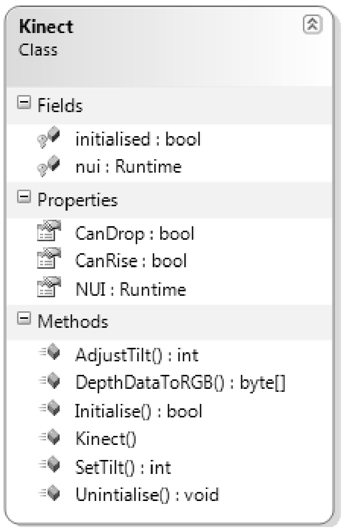


Figure 10: Kinect class structure

an attached Kinect device.

When the user gives the instruction to make an attempt to establish a connection to the Kinect device, several steps are carried out. The first is to initialise a series of arrays that will hold the images that come in from the Kinect's cameras. Following that, a call is made to the VMX Kinect class instructing it to attempt to initialise the Kinect runtime.

During the VMX Kinect class's initialisation process, four main steps are carried out. The first is to initialise the Kinect NUI runtime. This step includes giving a series of parameters to the runtime that specify which components of the Kinect runtime VMX wants to use. These components are the colour camera, the depth camera (including the ability to determine which pixels in the depth image correspond to a particular user), and the skeleton tracking system. The second step in the process is to open the stream from the video camera and define the parameters for it. The most important of these parameters is the resolution in which the camera should provide images (VMX uses 640 x 480, which is the maximum resolution of the Kinect camera (Microsoft, 2011)). The third step is similar and opens the stream from the depth camera. The depth camera is also capable of a resolution of 640 x 480; however this resolution can only be used when Kinect is providing only depth data for each pixel. VMX depends on also receiving data for each pixel that indicates whether that pixel “belongs” to a certain user (i.e. if a part of the user's body is in that pixel); when providing this data, the resolution of the depth camera is limited to 320 x 240 (Microsoft, 2011). The fourth step is to give an instruction to the skeleton tracking engine to use smoothing on its reported joint positions. Smoothing has the effect of greatly reducing jitter in the position of skeleton joints as reported by the Kinect runtime. If all four of these steps are successful the method will report its success back to VMX. If initialisation fails at any point, that will be reported back to VMX instead.

If a successful initialisation is reported, VMX will carry out a few last steps to complete the initialisation process. The first of these is to hook VMX's event handlers to the Kinect runtime's events. There are event handlers for three different events: Skeleton Frame Ready, which is triggered when the NUI has a new set of skeleton joint positions; Video Frame Ready, which is triggered when there is a new image frame ready from the video camera; and Depth Frame ready which is triggered when there is a new frame ready from the depth camera. Once the events handlers are hooked up, VMX sends a command via the Kinect class that sets the angle of the tilt on the Kinect device. The angle that is set is specified in VMX's configuration file. Once initialisation has been completed, interactions between VMX and the Kinect runtime only occur on a user command, or when an event is triggered from the runtime.

5.2.2 The Video Frame Ready Event

The Video Frame Ready event is triggered by the Kinect runtime when a new image frame from the Kinect's colour video camera available; when this occurs the corresponding event handler in the VMX core will be called. The first step of the event handler is to extract the raw image from the data provided in the event parameters. This data is held in a data structure called an ImageFrame. In addition to the data for the image itself, ImageFrame has some extra related information, including a timestamp for the image, the resolution of the image, a frame number, the type of image (i.e. Colour image), and a ViewArea object, which gives information about any zooming or panning used to generate the provided image (Microsoft, 2011). Following that, the image data is passed to the method that extracts any faces from the image for use on user avatars (more details on this are in Section 5.4). To make it easier to manipulate the image itself, its data is transferred from the one dimensional byte array in which it is provided, into a two dimensional array of XNA Color objects.

```
int i = 0;
for (int y = 0; y < image.Height; y++)
{
        for (int x = 0; x < image.Width; x++, i+=4)
        {
        videoTextureData[x,y] = new Color(image.Bits[i + 2], image.Bits[i + 1],
                image.Bits[i + 0]);
        }
}
```

Figure 11: Code for converting image data from a 1D array into a 2D array.

Figure 11 shows how this is done. For each pixel in the new array, three bytes taken from the old array; these bytes correspond to the red, green, and blue (RGB) values for the pixel. The RGB values are passed into a new Color object. It can be seen in Figure 11 that despite the fact that only three of the bytes are used for each pixel, the loop jumps ahead by four input bytes on each iteration, meaning one byte goes unused. This is because the raw data includes a fourth (alpha) channel for each pixel. However, as the pixels from the camera always have solid colours, this channel is not used.

After the loops have finished the event handler then is to call the method that draws extra graphics on to the image. The method draws markers that serve as an indication when detecting a TV screen in the image (For more information see Section 5.6.2). The final output array of Color objects is then stored in an instance variable for later use by other parts of VMX (e.g. the graphics system when generating the texture of the video feed for display in the graphics engine. See Section 5.3 for details).

5.2.3 The Depth Frame Ready Event

The Depth Frame Ready event handler is very similar to the Image Frame Ready handler. The Depth Frame has a payload of information that is similar to an image frame (the actual image data, a timestamp, the image resolution, the image type, a frame number, and a view area). This extra data is stored, and then the one dimensional byte array is copied into a two dimensional Color array in a similar way to the colour image. However, there is an extra step that must be carried out before the colour array can be filled. The depth data provided in the image frame includes both information on the depth of the pixel in question, and the user ID of the user whose body is occupying that pixel (if any). The two components of the data must be separated in order to create an image that is comprehensible to a user (the reason for creating an image at all is for diagnostic display of what the Kinect is seeing and what object(s) it is interpreting as a user. Figure 14 shows an example of such an image). The way the data is encoded is fairly complicated. It is provided in a byte array. Each pixel is stored in a two byte value, structured as shown in Figure 12.

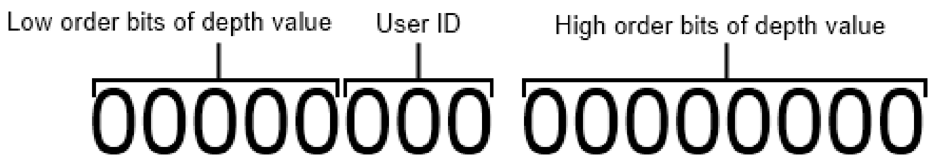


Figure 12: Byte structure of depth pixel data.

In order to produce an image, the byte array is passed to a static method in the VMX Kinect class to process it into something usable. The method takes the data from each two byte pixel and extracts it into a new array that is structured in the same way as a colour image frame array (i.e. four bytes per pixel, one byte for each of the red, green, blue, and alpha channels of the pixel). First, bitwise ‘and’, ‘or’ and shift operations are used to extract pixel data into a user ID and depth value, as shown in Figure 13.

```
int userID = depth[depthIndex] & 0x07;
int pixelDepth = (depth[depthIndex + 1] << 5) | (depth[depthIndex] >> 3);
```

Figure 13: Depth data extraction

Next, the pixel data is manipulated for display. Pixels that are closer to the Kinect device are made to appear brighter and the pixels that are further away appear darker. To achieve this, the pixel depth value is divided by 4095 (the maximum depth value), then the result is multiplied by 255, and finally the result of that is subtracted from 255; if the final result is less than zero then it is set to zero. This leaves a number that will be 255 if the pixel depth is 0mm from the Kinect, and range down to 0 if the pixel depth is greater than or equal to 4095mm from the Kinect. The number is then used as the basis for each of the red, green and blue channels of the pixel in the new colour array. The user ID value is used to modify which of the pixel’s channels are set in that pixel. This has the effect of colouring each user differently. If the user ID is 0 (i.e. there is no user in this pixel) then all channels are set, making the pixel grey; for user ID 1 only the red channel is set making the pixel appear red scale. User’s 2, 3, 4, 5, and 6 have their channels set so they appear green, cyan, yellow, magenta, and blue respectively. Once all of the pixels have been processed into the new colour array, that array is passed back to the event handler in VMX’s main class. From there the new array is processed in exactly the same way as the array in the colour image; with its data being extracted into a two dimensional array of Color objects. Figure 14 shows the final depth image, a single user can be seen in the middle, coloured on the yellow scale.

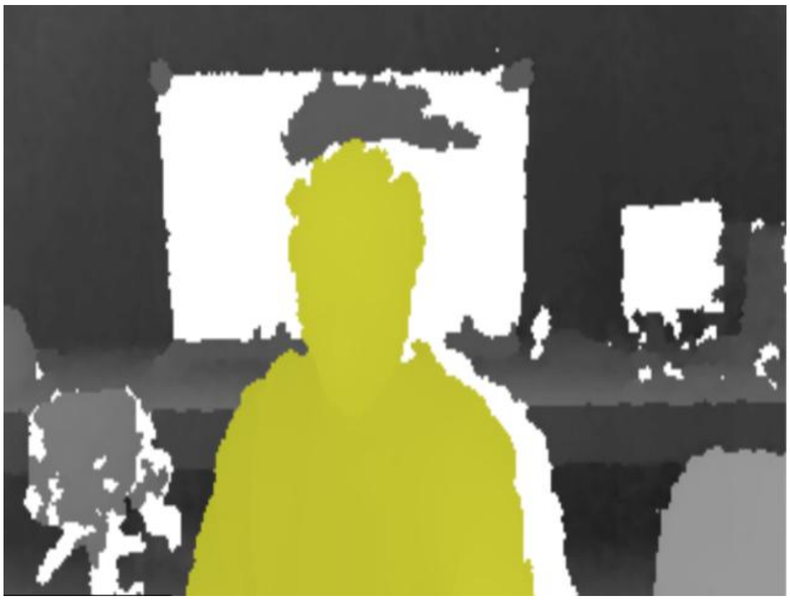


Figure 14: A processed depth image frame

The areas in Figure 14 that appear bright white occur where the Kinect is unable to determine the depth of the pixel. There are a number of potential causes of this, including: objects that are too close to the device (Kinect cannot pick up the depth of objects that are closer than 82cm away from the device), objects that are too far away, and any situation where the Kinect's infrared projections are not visible to its infrared camera (for instance, in the image above on the right hand side of the user, a white strip appears; this strip is a part of the user's body that is in the field of view of the infrared camera, but obscured from the view of the Kinect's infrared projector. Also the large flat white areas in the background are reflective surfaces that are deflecting the infrared signals away from the Kinect's camera).

5.2.4 The Skeleton Frame Ready Event

The Skeleton Frame Ready event handler differs significantly from the other two handlers. Its main function is to process the skeleton data from the Kinect runtime, and assign it its data to the appropriate avatar. Upon receiving a skeleton frame update, the event handler first extracts the data for the skeletons included in the update and puts that data into an array. Regardless of how many skeletons are actually being tracked, the array will contain data structures for exactly six skeletons, one for each of the six users supported by the Kinect runtime at any one time.

For each skeleton, first the tracking state is checked. The tracking state indicates what information is actually available for that skeleton. There are three possible tracking states: ‘Not Tracked’ means that no data is available for that skeleton, ‘Position Only’ means that the overall position of the skeleton is available but data for individual joints is not, and ‘Tracked’ means that all joint position data is available for that skeleton. VMX is only interested in ‘Tracked’ skeletons, so if a skeleton’s state does not equal ‘Tracked’ then it will be ignored, or in the case that the checked skeleton was previously tracked, its corresponding avatar will be deactivated. If a skeleton is marked as ‘Tracked’ then a series of steps occurs.

First a check is done to see if the skeleton’s corresponding avatar is already active (i.e. if the skeleton was also tracked in the previous skeleton frame update). If the avatar is not already active, then it is activated. The most important effect of activating an avatar is that it flags it as ‘Visible’ so that it will be drawn by the graphics engine.

Next, the data for that skeleton is passed as a parameter to the corresponding avatars update method (details on this method can be found in Section 5.4). Once that method returns, the index corresponding to the skeleton’s position in the array of skeleton data is taken. The index is stored in one of two variables, firstFound or secondFound (while the Kinect supports six simultaneous users, it will only perform full skeleton tracking on two). Which variable it’s stored in depends on whether it was the first or second skeleton in the array to be marked as ‘Tracked’.

Those two variables are used in the next stage of the handler, which decides which avatar to assign as the ‘main avatar’. The main avatar is the avatar whose data will actually be sent across the network, and used for things such as gesture detection. The first step is to check if there is already a defined ‘main avatar’. To check if there is currently an active ‘main avatar’, the avatar in the array at the index given for the ‘main avatar’ is checked to see if it is currently activated (this works because the Kinect maintains stable indices). If it is active then it will remain the current ‘main avatar’, and if there was a second skeleton marked as ‘Tracked’ in the skeleton frame update, its corresponding avatar will be assigned to be the ‘secondary avatar’. If there is not a currently active ‘main avatar’, then the first skeleton in the frame update that was marked as ‘Tracked’ will have its corresponding avatar assigned to be the ‘main avatar’, and the second marked as ‘Tracked’ will become the ‘secondary avatar’. If no tracked skeleton were found earlier then no change will be made to the currently assigned main and secondary avatars, but both will be kept in a deactivated (invisible) state. The final step carried out after a skeleton frame update is to run the algorithm that picks out colours for the avatars (more information in Section 5.4.6).

The reason for using this robust system to keep track of which skeletons are in use rather than just picking the first detected skeleton on every frame comes down to the environment that the program was developed in (a busy lab with many people walking in and out of the Kinect’s field of view) and also the Kinect’s tendency to occasionally misidentify inanimate background objects (chairs especially) as users.

5.3 Graphics

The graphics in VMX are based on the XNA graphics library, which itself is built on top of DirectX (Grootjans, 2009). There are several key classes that make up VMX’s graphics engine. At the lowest level there is the Geometry class that serves as a data structure for basic graphics data. A special static class called the Geometry Builder is used to produce instances of the Geometry class that are preloaded with the graphics data for basic shapes. An abstract super class called Drawable is used as the basis for any class that is able to be drawn on screen by the graphics engine. Finally a class called VMXModel provides the higher level functions for building and displaying objects in the virtual world. Figure 12 shows the relationship between these classes and the core VMX class.

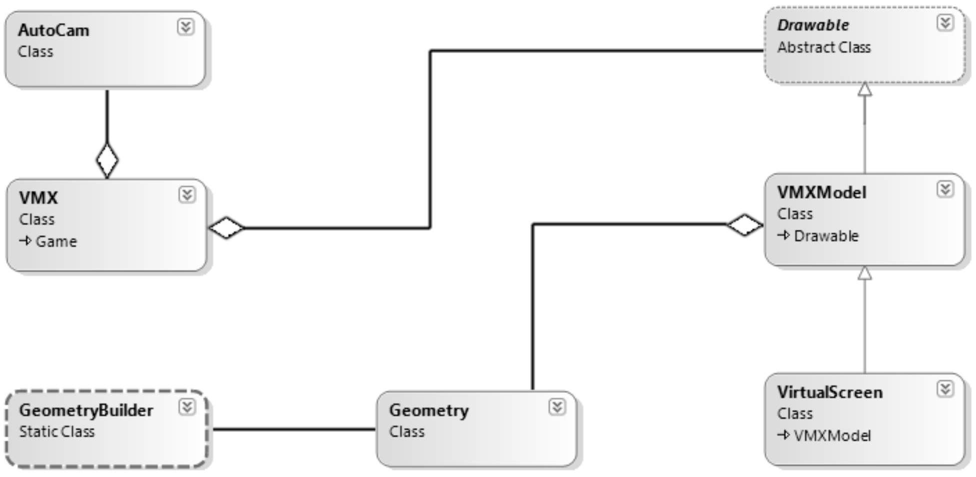
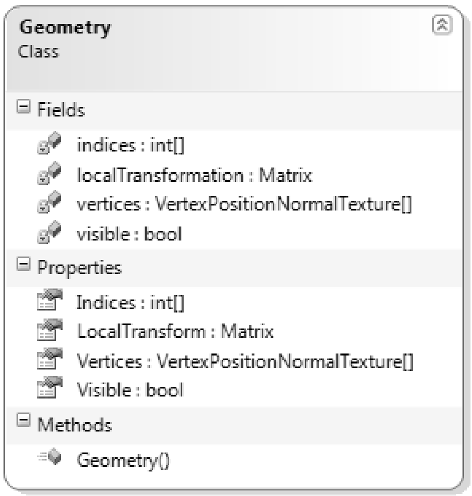


Figure 15: Graphics Engine Class Structure, not including avatar classes

5.3.1 The Geometry Class, Geometry Builder, and Drawable

The Geometry class forms the basic unit of all graphics in VMX. It is simply a data structure that holds one set of vertices, one set of indices to make polygons out of the vertices, and a world transformation matrix to be applied to those vertices.

The Geometry builder is an extensive class that used to create 3D shapes that have been preloaded into an instance of the Geometry class. It is capable of generating a variety of shapes including cuboids, inverted cuboids (whose polygons face inwards), spheres, textured quads, flat textured circles, cylinders, inverted cylinders, and tori. All of these can be generated according to specified parameters (e.g. level of detail, radius, width, height, length) depending on what kind of shape they are. Quads, cubes and circles are the only geometry that is generated with proper texture coordinates (as they are the only types of geometry that require them in VMX). All of the algorithms for generating these shapes are based on standard geometric formulas.



Geometry
Class

Fields
- indices : int[]
- localTransformation : Matrix
- vertices : VertexPositionNormalTexture[]
- visible : bool

Properties
- Indices : int[]
- LocalTransform : Matrix
- Vertices : VertexPositionNormalTexture[]
- Visible : bool

Methods
- Geometry()

Figure 16: Geometry class structure

Drawable is an abstract super class that provides the means for classes that inherit it to be drawn by VMX's graphics engine. It provides just two things to those classes. The most important is an abstract declaration of the draw method. This method is used by VMX to tell objects to go ahead and draw their geometry. The second purpose of Drawable is to store a flag that indicates whether lighting calculations should be applied to an object. The reason for this is to make it possible to draw certain objects at full brightness, irrespective of the actual lighting in the scene. An example of such an object is the main presentation screen in the virtual meeting room. Being drawn at full brightness improves the visibility of the data on the screen and makes it look as though it is a projected image or active display screen. For other objects an appropriate lighting model supports the 3D appearance of the scene. In VMX all objects that need to be drawn in the virtual environment are kept in a single list of Drawable type objects; thus no 3D object that doesn't inherit from this class can be drawn in the VMX graphics engine.

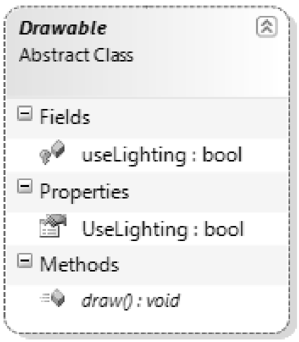


Figure 17: Drawable class structure

5.3.2 The VMXModel Class

VMXModel is the class that is responsible for the high level management of a set of geometry. All 3D objects in the virtual world are displayed using the VMXModel class. It provides facilities for easily adjusting the transformation matrices of a model, texturing a model, and drawing a model. Stored within the VMXModel class there is a list of one or more instances of the Geometry class, which provides shapes for the model.

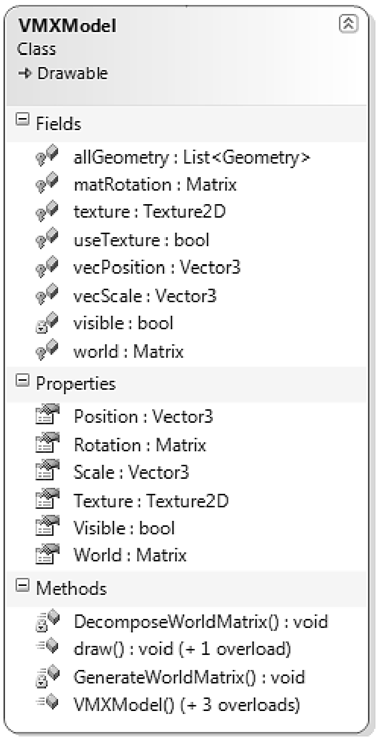


Figure 18: VMXModel class structure

To make it easy to manipulate the position of a model, two ways to modify the world transformation matrix are provided by the VMXModel class. The first allows a world transformation matrix to be passed into the model directly. When this is done the VMXModel will store the matrix and utilise it when it needs to draw itself. When passed such a matrix, VMXModel will also run an algorithm to decompose it into its individual components: a translation vector, a rotation quaternion, and a scale vector (the method for doing this is provided by the XNA libraries). The translation and scale vectors are stored in the VMXModel as is, where they can be read by other parts of the program. However, the rotation quaternion is first turned into a rotation matrix (by a call to an XNA routine) before being stored in a similarly accessible location. The second way of modifying the world transformation matrix is to change the individual translation, rotation, and scale components directly. This is done by passing in the appropriate data structure for each one (vector, matrix, and vector respectively). When any one of these components is changed, a new world transformation matrix is immediately generated using the updated copy of the changed component, and the existing copies of the other two components. The new world transformation is then made publically accessible.

The VMXModel class is also responsible for the actual drawing of the geometry it contains onto the screen. It will do this with a method called draw, the call to this method is made either from the main VMX program or from draw methods in other objects that themselves are called by VMX. When the call is made, VMX will provide the graphics device which is to be used, and the shader to apply to the geometry. When entering the draw method, first the VMXModel will determine if it set to be visible. If it is not, then the method will immediately return and nothing will be drawn. If the model is set to be visible, then the next step is to tell the shader about any texture to be applied to the geometry. First the shader is told whether or not to use a texture at all; then it is passed whatever the VMXModel has stored in its texture property. Following that the model will begin iterating over each of the Geometry objects it has stored within it. For each one a complete world transformation will be calculated and passed to the shader. Two or three different transformations are involved in the calculation shown in Equation 3.

$$World\ Transformation = Local \times Model \times Base$$

Equation 3: Calculation of the final world transformation for a piece of Geometry.

Local is the model space transformation supplied with the particular instance of Geometry involved; Model is the world transformation of the entire model, stored in the VMXModel class; and Base is an optional additional transformation that can be passed into the draw method to further adjust the Model transformation. Note all of the values in the above equations are matrices and thus the order in which they are multiplied is important. The final transformation is then passed into the shader.

At this stage the shader is told whether or not to use lighting based on the value stored in the UseLighting property of the object to be drawn.

The last step of the draw method does the actual drawing. For each effect pass in the shader the pass is applied and the graphics device is instructed to draw the series of indexed primitives based on the vertices and indices stored in the current piece of geometry.

5.3.3 The Core Graphics System

Much of the work for creating and displaying the graphics is done by the core class of VMX. This includes the initialisation of the graphics system, loading and creating graphics content, and issuing the commands to the other classes used in the graphics engine.

Initialisation of the graphics device is handled by the XNA game class code when the program is started. The first major step that occurs in the VMX class itself is the creation of the window for display of the 3D graphics. This is done using the Windows.System.Forms library. Depending on the program configuration a second window may be opened at this point.

This second window is used when the user is using a second, real-world screen set up behind them for the purpose of running presentations in VMX. The second window is opened and set with a solid background colour (red). The background colour is used by the algorithm responsible for determining the position of the screen in the Kinect's field of view. (More details can be found in Section 5.6).

Following window creation, the shader to be used is created. VMX uses the BasicEffect shader that is provided as a part of the XNA libraries.

Following this, the lights and cameras in the virtual environment are set up. VMX uses low level ambient lighting and three different directional lights to fully light a scene (two of the directional lights are deactivated if for some reason the background graphics are not in use). The camera set up involves initialising a series of settings which relate to both automatic and manual cameras (more detail can be read in Section 5.7).

The next step in the initialisation process involves loading in, and creating content to be displayed. The only external files used by VMX's graphics are texture files. They are loaded in using XNA's content manager, which takes normal image files and constructs Texture2D objects (Texture2D is the class provided by XNA for storing and managing two dimensional textures). After that is done, VMX begins setting up data structures for storing the various kinds of avatars (see Section 5.4 for details). Then, the background graphics for the virtual meeting space are generated, including the walls, table, chairs and the various screens and added into a list of Drawables. The Drawables list is used as a central data structure for accessing all of the objects in the system that can be drawn on screen. The final step in the initialising process is to load a screen font for use when drawing text onto the screen.

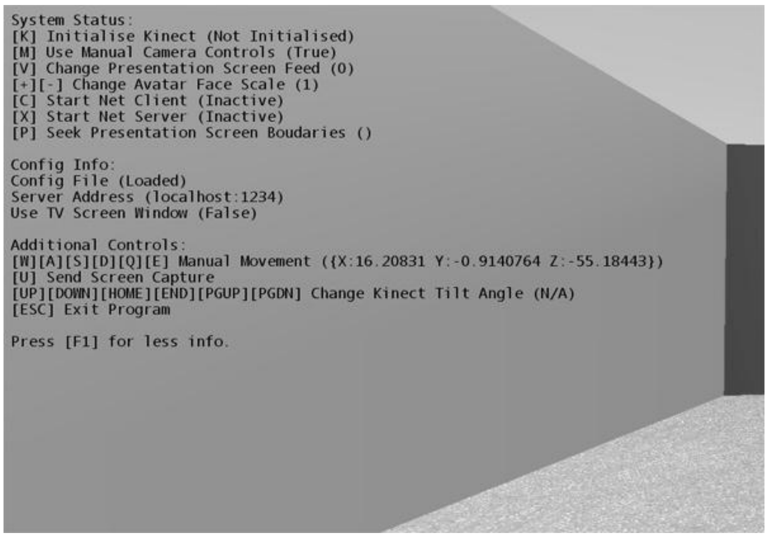
During normal program operation the main function of the graphics engine is carried out on every iteration of the main program loop. The program loop itself is within the XNA provided super class of the VMX main class. The super class is called Game. From Game a call to the 'Draw' method is issued. VMX's main class provides the actual implementation of Draw. When it is called a series of steps is carried out which ultimately draw all of the graphics for one frame of the program.

To begin with the graphics from the last frame are cleared off of the screen render surface. Then the view and projection matrices are passed to the shader. These matrices are generated from the camera system and affect the user's view of the virtual environment. The view matrix determines the location of the camera in the virtual space; and the projection matrix determines certain properties of the camera, such as the field of view, and how far it can see.

If the Kinect is currently active and providing data to the program, then at this stage the textures that show the raw depth and video images from the Kinect are updated to show the latest available data. The first step of doing this involves creating a new texture in the system with a width and height matching the resolution of the camera feed from which the texture will get its data. VMX maintains two two-dimensional arrays in which the latest images from each of the Kinect's respective cameras are stored. In order to be converted into a texture, the contents of these arrays must be extracted into one dimensional arrays. So, new arrays of colour values are initialised; their lengths matching sizes of the two dimensional arrays. Once filled the one dimensional arrays are copied into the texture data for appropriate images. The final result is two textures that can be used anywhere in the graphics engine, one showing the output from the Kinect depth camera, the other showing the output from the video camera.

The next stage of the drawing process involves the actual drawing of the 3D graphics in the scene. This process is made simple by the use of the Drawable superclass, and the list of Drawables that VMX maintains. The Drawables list is iterated. Each Drawable object has its UseLighting property checked and the associated setting in the shader is made accordingly. Following that, the only thing that needs to be done is to make a call to the Drawable's draw method with the shader and graphics device passed in as parameters.

Once the 3D graphics are drawn, the two dimensional HUD (Heads Up Display) is drawn. This is made up of text that informs the user of various aspects of the program state, and of the various controls the user has access to. The text is drawn by a SpriteBatch object (provided by XNA) using a SpriteFont that was loaded in at content load time. The HUD itself has two display modes. The first shows only an instruction to the user about how to open the full HUD, and possibly an important notification from some part of the program. The other mode gives a list of the user's controls, and data from various parts of the program (for example the server and client systems both show their upload and download rate on the HUD when they are running).



```
System Status:
[K] Initialise Kinect (Not Initialised)
[M] Use Manual Camera Controls (True)
[V] Change Presentation Screen Feed (0)
[+][-] Change Avatar Face Scale (1)
[C] Start Net Client (Inactive)
[X] Start Net Server (Inactive)
[P] Seek Presentation Screen Boudaries ()

Config Info:
Config File (Loaded)
Server Address (localhost:1234)
Use TV Screen Window (False)

Additional Controls:
[W][A][S][D][Q][E] Manual Movement ({X:16.20831 Y:-0.9140764 Z:-55.18443})
[U] Send Screen Capture
[UP][DOWN][HOME][END][PGUP][PGDN] Change Kinect Tilt Angle (N/A)
[ESC] Exit Program

Press [F1] for less info.
```

Figure 19: The full read out HUD

Once the HUD is drawn, VMX’s drawing process is finished and a call back to XNA’s libraries is made for it to finish the process of getting the graphics onto the user’s computer monitor.

5.4 User Avatars

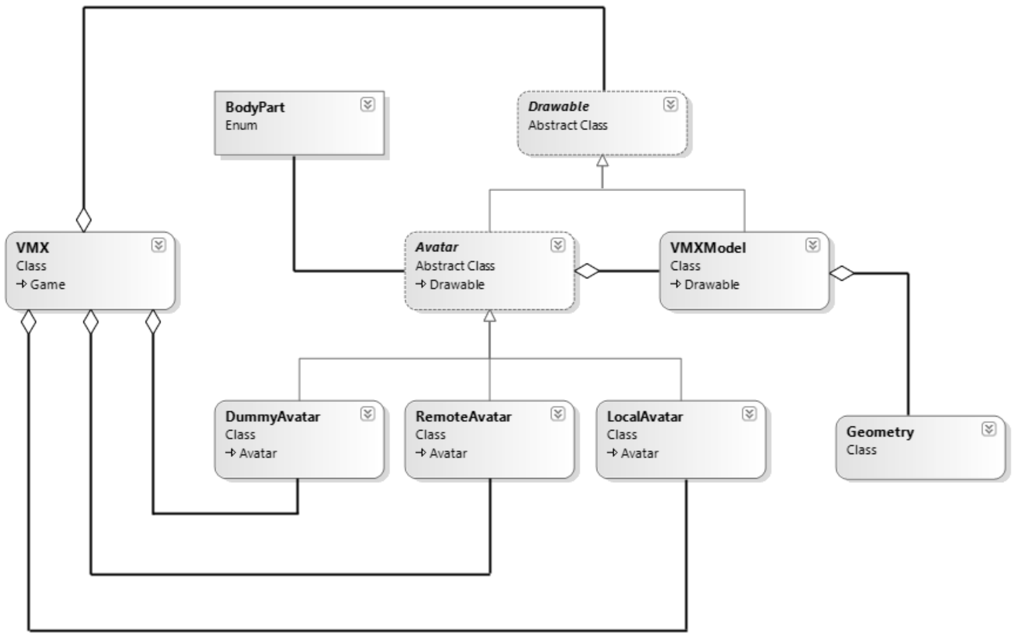


Figure 20: User Avatar classes within the graphics system

5.4.1 Overview

VMX requires a way to represent the skeleton information provided by Kinect on screen in the graphics system. This is achieved in a series of classes which store and manipulate the information from the Kinect and use it to draw onscreen graphics representing users in the system.

The data that forms the base of all of an avatar’s functions is the skeleton data provided by the Kinect runtime. This data is in the form of twenty sets of three dimensional coordinates. Each set corresponds to a single “joint” in the Kinect skeleton, and represent runtime’s estimation of where that joint is positioned in 3D space based on real world data from the depth camera. The names and relative locations of the twenty joints are illustrated in the figure below.

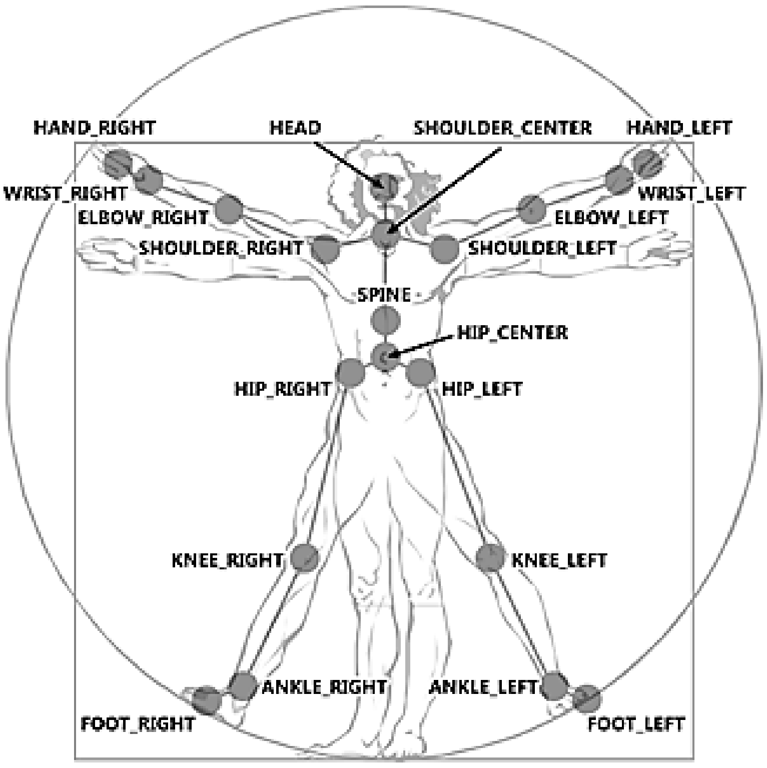


Figure 21: Kinect skeleton joint diagram. (Microsoft, 2011)

In normal operation VMX utilises two kinds of avatars ‘local avatars’ and ‘remote avatars’. Both kinds of avatar utilise the same graphics system and basic geometry. The principle difference between them is how they receive joint position data and where it comes from. Local avatars are provided with data from a Kinect attached to the same computer the program is running on. Remote avatars receive their data over a network from other computers running an instance of VMX.

5.4.2 Geometry

When an avatar is first initialised, its basic graphics geometry is generated. The current avatar graphics are made up of several different components and have been through several iterations. Originally the entire avatar was made up of twenty spheres; each sphere represented a single joint in the Kinect skeleton. They were positioned based on the coordinates of their corresponding joints. This version of the avatar was good for visualising the data coming from the Kinect but provided no way to determine the identity of the person in control of a given avatar.

To rectify this, the sphere that represented the head on an avatar was replaced. The initial replacement was a simple forward facing square that was textured with an image of the users face. This was successful in allowing the identity of each avatar to be determined; however it did suffer from one problem. Visually the new face image didn’t stand out well from the background graphics in the virtual environment. This was especially true of avatars that were standing in front of the presentation screen when it was displaying a complex image. The first attempt to rectify this involved replacing the square head with a circular one; the idea being that the curved lines this would create would stand out against the background which tended to be made up of straight lines. While this did improve the situation, it did not entirely fix the issue. The final solution was found by drawing a border around the circle that displayed the face. This was achieved by adding another piece of geometry, a torus (donut shape) which was positioned around the edge of the circle. This solution was successful in making the face stand out against the background. The torus solution also had a secondary benefit. Previously, when viewed from behind, the avatar’s head was invisible. This is because only one side of the face circle is visible, due to the way the graphics engine works (a primitive is only drawn when viewed from a particular side, though this could have been changed in the graphics settings). The torus however, being a fully 3D shape with primitives facing in all directions, is visible from all sides; allowing users to determine the location of other avatar’s heads from behind. The fact that from behind only the border of the head is visible, serves the secondary purpose of allowing users to easily see past other user’s avatars when they are facing away.

To retrieve an image of a user’s face to put onto the circle representing the head, another algorithm is needed. The source of the face image is the video feed from the Kinect’s colour video camera. A single frame from the video camera contains a lot more than just a user’s face, and because of the limited size of the circle the image is going on, it is desirable to isolate the part of the image that contains only the user’s face. The method for doing this starts by retrieving the coordinates of the user’s head from their Kinect skeleton, these coordinates are then translated by the Kinect libraries from skeleton space to depth-image space. At this stage the actual depth of head is retrieved and stored for later use. The coordinates are then translated again from depth-image space to colour-image space. The resulting set of coordinates can then be used to determine the location of the pixel in the video image that corresponds to (approximately) the centre of the user's head. From there it is necessary to determine how large an area is needed for a sample from the image to include the user's entire head. This is done using the depth value for the head that was stored earlier. Dividing an appropriate constant by that depth value yields a value for the length (in pixels) of a side of a square area to be sampled. On the scales that Kinect uses, this gives sufficiently good results. The specific constant value that VMX uses is 1,300,000; this value was chosen through experimentation. Larger constants will result in more area from around the head being captured, and smaller constants will focus the captured area more tightly on the face. After an area to capture is determined, checks are made to ensure that all of the boundaries of the area fall within the bounds of the video image. If they don't the area is adjusted, usually by moving it across so that it no longer intersects a boundary, but if the area too large to fit within the image regardless of how it is moved, then it will be reduced in size. Once the appropriate area to sample has been confirmed, the latest frame acquired from the Kinect's video camera is sampled and the appropriate pixels copied into a new texture that is applied to the face circle of the user's avatar. The following figure illustrates how the sample area is determined.

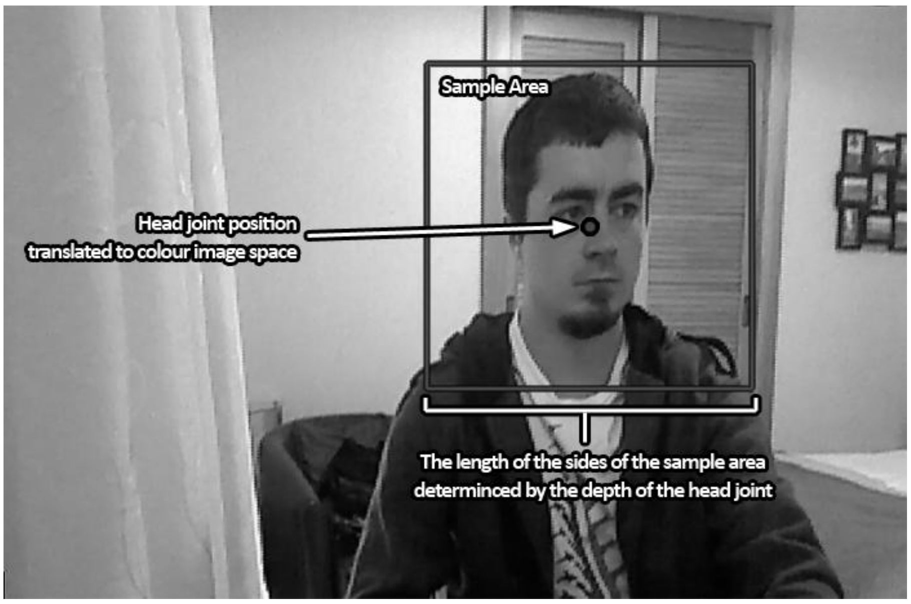

Sample Area
Head joint position translated to colour image space
The length of the sides of the sample area determined by the depth of the head joint

Figure 22: Illustration of the face capture algorithm

With the original avatar design, when multiple avatars were on screen at the same time, it could become difficult to tell which spheres belonged to which avatar, making it confusing for a user to tell how an individual avatar was moving. Another problem occurred if the Kinect was having trouble determining the precise location of any joints as it became difficult to tell which sphere was representing which joint (for instance, if the user's legs weren't visible to the Kinect, their foot joints might spontaneously jump above their head joint); this lead to the inclusion of the final part that makes up an avatar's geometry. Eighteen one unit diameter cylinders are used to connect neighbouring joints. They provide much of the shape of the avatar and make it clear which joints belong to which avatar, and how each joint in the avatar is related to the other joints. The cylinders share the same diameter as the joint spheres, and their start and end points lie in the centre of those spheres. This gives the avatar a smooth and contiguous shape, forming a connected body frame for the avatar.

Figure 23 shows the development history of the appearance of the VMX avatar. Moving left to right shows the progression from each version of the avatar to the next. The first avatar is the original, made only of twenty spheres. The next is the version where the head is replaced by a forward facing quad. The third has the quad changed to a circle. The fourth avatar was the first to include connecting cylinders to better define the avatar's shape (note that the hand, foot, and head joints are *not* connected to the rest of the avatar's body). The final avatar is the current version, displaying a torus around the face.

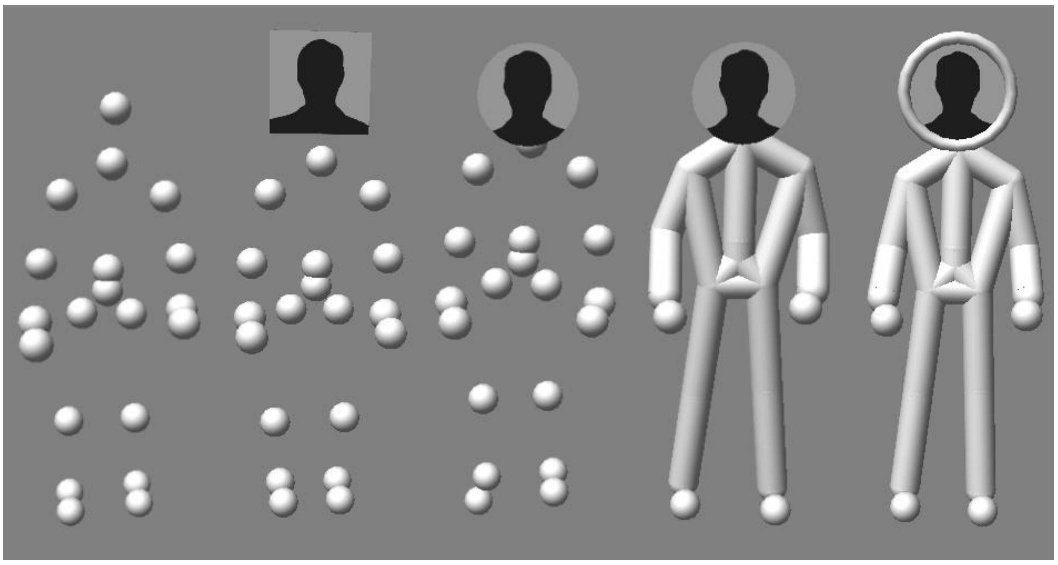


Figure 23: Development history of the VMX avatar.

As a user moves around, the relative location and distance between joints in a Kinect skeleton constantly changes. Because of this the shape and orientation of the cylinders must be constantly updated. The algorithm to do this is common to both avatar types and is run every time new joint data is provided to an avatar. The cylinders are all stored in a single array of all avatar geometry. They are accessed using an enumeration which gives each a name related to the part of the avatar's body it represents (e.g. ForeArmRight, UpperLegLeft, Spine etc.). The algorithm works by iterating over every cylinder in the avatar and for each it calculates a new translation, scale, and rotation matrix which when applied to the cylinder, shape and position it to perfectly bridge the gap between two joints.

All cylinders when first created are generated with a length of one unit, and run end to end along the skeleton space z-axis (which is the axis which follows a line moving out horizontally from the front of the camera). These facts are important to the function of the cylinder positioning algorithm. The first step of this process is to acquire the vector that connects the two joints that will sit at each end of the current cylinder. This vector is calculated using vector subtraction, shown in Equation 4.

$$c = j_1 - j_2$$

Equation 4: Vector subtraction

Where c is the vector that connects the positions of the two joints (here-after referred to the connecting vector). j_1 is the vector that represents the position of one of the joints, and j_2 represents the position of the other. c will run in the direction from j_2 to j_1. From this new connecting vector the correct length and orientation for the cylinder can be derived. The length of the cylinder is changed by adjusting its scale matrix. A scaling factor is acquired simply, with the following equation:

$$ScaleFactor = |c|$$

Equation 5: Cylinder scale factor

Because the cylinder is aligned along the z-axis, in order to increase its length without changing its diameter the scale factor is applied to the z-axis only, with the x and y-axes left unscaled. This results in a cylinder that is the correct length to connect the two joints.

The next step is to rotate the cylinder so that it lies on a line parallel to the line that connects the two joints. This is a two step process. The first step is to find the angle by which to rotate the cylinder. The second is to find the axis around which to perform the rotation. By default, the cylinder is aligned along the z-axis, to connect the two joints it needs to be aligned to the connecting vector between them; this means that the angle between a unit z vector and the connecting vector is the angle that the cylinder needs to be rotated by. This and can be acquired as shown in Equation 6.

$$Rotation\ Angle = cos^{-1}\frac{c \cdot z}{|c|}$$

Equation 6: Determining the angle between a vector and the Z axis.

Here c is the connecting vector, and z is a unit z vector (it is important that the z vector be of length 1). Note that the dot refers to a vector dot product operation. Having acquired the angle it is then necessary to find the rotation axis. The correct axis to rotate on is represented by the vector that is perpendicular to both the cylinders default orientation (the z axis) and the desired orientation (the connecting vector). This new vector can be found with a vector cross product as shown in Equation 7.

$$Rotation\ Axis\ Vector = c \times z$$

Equation 7: Determining an appropriate rotation axis.

Again c is the connecting vector, and z is a unit z vector. As a final step before using it, the rotation axis vector is normalised. The final rotation matrix for the cylinder is calculated using the XNA graphics libraries with the calculated angle and axis.

With the cylinder scaled and rotated, the final step is to translate it. The cylinder now has the same length and orientation as the connecting vector. The connecting vector reaches exactly from j_2 to j_1 (from the vector subtraction equation above); by placing the cylinder at the position given by j_2 it too will reach from j_2 to j_1. Thus the translation vector for the cylinder is set equal to the vector j_2. With the rotation, scale, and translation calculated; the final world transformation matrix for the cylinder is calculated and applied. Before this is done the scale and translation vectors are turned into matrices using XNA provided routines. The final world transformation is calculated as shown in Equation 8.

$$W = S\ \times R \times T$$

Equation 8: Calculating a World Transformation.

Where W is the final world transformation matrix, S is the scale matrix, R is the rotation matrix, and T is the translation matrix.

This process is repeated for every cylinder in the avatar, until all are in the correct position. This leaves the avatar in its final position, ready to be drawn by the graphics engine.

5.4.3 Local Avatar Data

Local Avatars are user avatars that are rendered from the data coming directly from the Kinect device that is connected to the computer on which VMX is running. In practice two local avatars may be active within VMX at any given time, however the second of these two avatars does not get used in VMX, beyond rendering it in the virtual environment (i.e. the user controlling the second avatar cannot use gesture controls, and the avatar data is not transmitted across the network, so it does not appear to other users in the meeting).

As described earlier the skeleton frame will include information for six skeletons, corresponding to the six users that the Kinect can support at any one time. However while it can identify six users simultaneously, Kinect only supports skeleton tracking for two of those users at a time. These two skeletons are assigned the Skeleton Tracking State: 'Tracked' by the Kinect. VMX maintains an array of six local avatars, each one corresponding to a potential Kinect user. Only two of these avatars are set to be visible at any one time, and only the first of those two to be detected is generally used by the program (for gesture recognition, sending avatar data over a network etc.); this avatar is designated the 'main avatar'. If a tracked user leaves the field of view of the Kinect sensor, their avatar will disappear and any non-tracked user that remains in view will be upgraded to tracked state if possible. If the user that left was in control of the main avatar then the next tracked user will take control of the main avatar.

When a skeleton is identified as tracked in the SkeletonFrameReady event handler its data is passed on to the corresponding avatar which begins processing that data. The first thing an avatar does is extract the joint coordinate data from the skeleton data. Each joint's position data is read in sequence and the position of the avatar's corresponding geometry is updated to reflect that. The data is provided as Kinect vectors, which need some translation before they can be used by VMX's graphics engine. The first step is to repackage the Kinect vector data into a XNA Vector3 class; this is a straight forward process involving taking the X, Y, and Z coordinates stored in the Kinect vector and using them directly as parameters to create a new Vector3. The second step is a matter of scaling; the scales used in Kinect skeleton space are smaller than the scales used in VMX graphics space. To compensate for this all Kinect data must be scaled up to appear at a reasonable size. This is done by simply multiplying the resulting Vector3 by a global constant (VMX uses 10). When the data for the head joint is read, the additional calculations relating to head position are done at the same time (details presented earlier).

5.4.4 Advanced Positioning and Movement

The raw positions of an avatar’s joints as they are provided by the Kinect do not always result in the most desirable position of the user avatar. The raw data can contain misjudged positions of joints which will cause the avatar appear oddly shaped (e.g. an avatars limbs might be arranged in ridiculous positions). In addition, information about the rotation of terminal joints (i.e. hands, feet, and head) is not available (so for instance, one can’t tell which way a users head is facing from the head joint data). For these reasons VMX utilises a series of algorithms that tweak the raw position data to make avatars move in ways that are more informative to other users.

The first of these tweaks is used when a user is sitting down. If an avatar is sitting at the virtual table (i.e. not doing a presentation) it is generally expected that the user is also sitting down. There are two problems with this. The first is that the Kinect is not well equipped to estimate the position of a user’s body when they are sitting down (though it does do a passable job). The second is that there is a fairly good chance that if the user is sitting down, their legs will be obscured from the Kinect’s field of view by their desk. The current version of the Kinect SDK has no built in way to deal with this. The result is that when a user is sitting in full view of the Kinect device, their avatar appears to be more squatting than sitting; and when a user’s legs cannot be seen at all by the device the avatar’s legs will tend to appear in bizarre positions. The first two avatars in the following figure illustrate these two situations respectively.

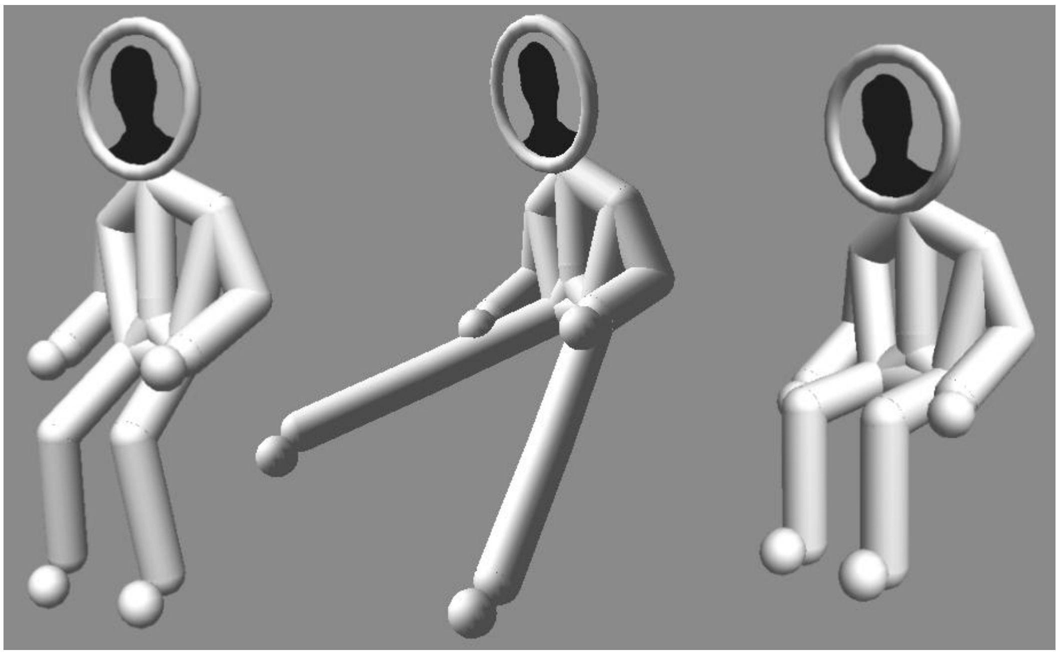


Figure 24: Avatars sitting down. Left: A user sitting with legs in view of the Kinect; Centre: A user sitting with legs obscured from the Kinect; Right: Same as the centre image, but the avatar's legs are forced into a sitting position.

The third avatar in this figure illustrates the solution to the problem. When a user is known to not be currently doing a presentation, VMX will force their avatar into a sitting position. To do this three steps are taken. Firstly, all joints that are normally above the hip centre joint have their positions changed from being defined in terms of the avatar's local space origin to being defined in terms of the hip centre joint position. Doing this means that the second step can be carried out, which is to change the hip centre joint position to sit at the avatar's origin, while maintaining the relative locations of it to all of the joints above it. The overall purpose of this is to allow the program to control how far away an avatar is sitting from the virtual table, rather than leaving that to be determined by how far away a user is sitting from their Kinect. The reason for doing this is to essentially normalise the data being received from multiple Kinect devices over a network; if this wasn't done, any user connecting in from a remote computer could appear in the virtual environment at a dramatically different height or distance from the table compared with other users. The final step of this algorithm is to simple force the joints that normally sit below the hip centre joint (i.e. the joints that make up the hips and legs) into hardcoded positions relative to the hip centre joint. These hardcoded positions give the legs appearance of being in a sitting position, regardless of where the Kinect believes the legs are.

VMX performs other adjustments to avatars besides forcing an avatar sit down. The second modification it makes relates to the direction that the avatars head is facing.

In a virtual meeting environment it is desirable to give users an indication of which way other users are looking. This information cannot be acquired purely from the head joint for two reasons. Firstly Kinect provides no estimation of how various joints are rotated in space, so it is not possible to determine what way a person's head is facing. Secondly, even if that information was available, or was determined by other means; there would almost certainly not be a one to one relationship between how much a user turned their head (as they would likely always have their head turned towards the computer monitor that they were using), and how much their avatar would need to turn its head to look at the same point in the virtual environment that the user is looking at. Over the course of the project two different systems for solving this problem were created. Both systems remain in VMX; with each being used under different circumstances.

In the first implemented solution to this problem, the orientation of an avatar's head is determined by the same principles which determine how to direct the Automatic Camera (more detail on the automatic camera can be found in Section 5.7.2). Specifically the orientation of the avatar's head is actually based on the movements of the user's shoulders. If the user twists their body to the left such that their right shoulder ends up closer to the Kinect than their left shoulder, then their avatar's head will appear to turn to the left. The opposite occurs if the user twists their body to the right. The algorithm to determine how much to turn the user's head by works by first acquiring the coordinates of the user's left and right shoulder joints from the their Kinect skeleton. The coordinates for the right shoulder are then subtracted from the coordinates for the left shoulder to give a vector that follows the angle of the line between the two shoulders. The angle between the vector and the skeleton space x-axis is determined by again referring to the equation for finding the angle between two vectors, shown in Equation 9.

$$Angle = \cos^{-1}\frac{s \cdot x}{|s||x|}$$

Equation 9: Determining the angle between to vectors.

Where s is the vector representing the line between the two shoulders, and x is the vector representing the x-axis. The x-axis is used because in Kinect skeleton space the x-axis runs horizontally and perpendicular to line running directly out from the front of the Kinect camera, which is the line on which a user's shoulders would both sit if they were directly facing the camera. This use of this equation differs slightly from its earlier use in that the length of vector x is factored into the equation. This difference is due to the fact that the length of x cannot be assumed to be one whereas the length of a unit z vector can be assumed to be one. The angle acquired from this equation is always positive, so an additional step is taken to identify which shoulder is in front of the other. If the left shoulder is in front then the final angle it multiplied by -1. This calculation effectively provides an angle for the 'yaw' of the avatars head.

Before this angle is applied to rotate the avatar's head, a second angle is calculated. This angle represents the pitch of the users head (i.e. whether they are looking up or down). This angle is acquired using the same equation and process as used for acquiring the yaw. Different joints are used however. In this case the vector between the head joint and the shoulder centre joint is used, and it is compared to the y-axis (which runs vertically in Kinect skeleton space) instead of the x-axis. Having acquired angles for pitch and yaw, a rotation matrix is generated from them using the XNA graphics libraries. This rotation matrix is then incorporated into the world transformation matrix for the user avatar's head.

This system for orienting avatars heads was originally used at all times in the program, including when determining which way Remote Avatars were looking and when the user was using manual camera controls. It also suffers from a slight problem in that the automatic camera system uses a multiplier to exaggerate the degree of rotation of the camera's view. All of this meant that the direction that an avatar's head was facing might not really represent the actual direction in which a user was looking; this lead to the creation of a second system that directly uses the direction in which the user’s virtual camera is looking when determining what way to orient an avatar’s head.

The algorithm for the second head orientation system starts by acquiring two vectors. The first of these vectors represents the current direction that the virtual camera is facing; this will be called the CameraDirection vector. The second vector represents the direction that an avatar’s head will face by default (this vector always has the value (0, 0, -1)) it will hereafter be referred to as the DefaultDirection vector. Then these two vectors are used to calculate a yaw and pitch angles for the user avatar’s head.

First the yaw is calculated. This is done with the same equation used in the first head orientation system (Equation 9). In this case the two vectors used in the equation are the DefaultDirection vector and a version of the CameraDirection vector that has had its Y coordinate set to equal zero. As in the earlier use of this equation the angle it produces is always positive; so it must be multiplied by negative one if the X component of the CameraDirection vector is positive. The pitch angle is also calculated using Equation 9. This time the vectors used are the CameraDirection vector with its Y component set to equal zero, and the full CameraDirection vector with its normal Y component value. If the normal Y component has a value that is less than zero then the final angle must be multiplied by negative one to ensure that the rotation it produces is in the correct direction. The resulting values for the yaw and pitch of the head are again stored in the avatar’s class where they are used to create the rotation matrix for the avatars head whenever it is updated.

Under ordinary circumstances, when a user has the automatic camera enabled and their avatar is sitting at the virtual meeting table (i.e. not doing a presentation), the yaw and pitch values calculated above ensure that the avatar will appear to look at exactly what the user is actually seeing through the virtual camera. This however is not true in situations where the virtual camera is not located in the same place as the avatars head. There are two circumstances where this can occur. The first is when the user is doing a presentation, and the second is when the user uses the manual camera controls to change the normal position of the camera.

When a user is doing a presentation the automatic camera moves to a stationary location at the end of the virtual meeting table and points in a fixed direction (at the audience). Doing this allows the user to move around freely while doing a presentation without the camera moving around with them and losing its view of the audience. If the system above was used for positioning the avatars head in this situation, then the result would be to give the avatar the appearance of having a fixed stare towards the audience, which looks slightly unnatural. For that reason when doing a presentation the avatars head orientation is calculated using the old method (where the head direction is determined by relative position of the user's shoulders). This gives a more natural look to how the avatar moves.

When the user is using the manual camera controls, the camera could be absolutely anywhere in the virtual space and looking in any direction. This can pose a problem when using the newer system for determining head orientation. To give an example, say the user moves the camera to the opposite side of the virtual meeting table to their avatar, and then looks back towards the table. Under the system described above the result would be that the avatar would appear to look in the direction directly behind them (i.e. the avatar's head would have spun 180^o from the looking forward direction). Aside from looking unnatural, this tells other users nothing about what the avatar's user is looking at (the user is looking at the table and the avatar is looking away from the table). A simple solution is used for this; constraints are applied to the directions that the avatar can appear to look in when the user is using the manual camera controls. The avatar is limited to looking in directions within the environment where other avatars are likely to be (around the virtual table, and in front of the virtual presentation screen). This solution is not perfect and sometimes causes the avatar to look in a direction that is not representative of where the user actually looking, but the chances of this occurring are reduced nonetheless, and there is no chance of the avatar rotating its head in physically impossible ways (i.e. 180^o backwards).

The newer system for calculating head orientation does hold one major disadvantage over the older system. The older system only required the shoulder position data to calculate head orientation. When communicating across a network this data is sent as part of the skeleton data for each user, thus remote instances of VMX could determine the correct orientation of the head of each avatar in the meeting from data that was already available. The new system however requires information about a user’s camera to calculate the head orientation. Camera data is not exchanged between instances of VMX. This means that additional information must be sent along with the skeleton data in VMX communication. This information takes the form of two single precision floating point numbers that give the pitch and yaw angles of an avatar’s head. These are included in each packet that VMX sends that also includes updated skeleton data (For details see Section 5.8).

Besides forcing sitting positions and calculating head orientations, one other modification is made to the raw skeleton position data when determining avatar positions. This last modification is simply to place the avatar at an appropriate location in the virtual environment. This is done with nothing more than a world transformation matrix for the entire avatar.

The coordinates for the position of the geometry of an avatar are determined by a user’s relative location to their Kinect device. Since all users in a meeting are likely to be sitting at similar distances directly in front of their device, then if nothing was done their avatars would likely occupy the same space in the virtual environment. To prevent this, all user avatars are given a unique world transformation matrix that places them is a particular place in the virtual environment. VMX currently maintains nine hardcoded world transformation matrices, one of which is assigned to each avatar involved in a meeting. Which matrix an avatar receives is determined by two factors. The first is the order in which the client VMX programs connected to the server. When a client connects it is assigned an ID (the server also assigns itself an ID), That ID is used to determine which of eight matrices corresponding to the locations of the virtual chairs around the virtual table is assigned to that client’s avatar. The ninth world transformation positions an avatar in front of the virtual presentation screen; this transformation is assigned to which ever avatar is doing a presentation, and overrides that avatar’s chair position (until they have finished presenting).

5.4.5 Remote and Dummy Avatars

The process for updating a Remote Avatar is much the same as the process for the updating a Local Avatar, differing only in the format in which the Kinect's skeleton data is provided. Where a Local Avatar receives a complete SkeletonData object from the Kinect; a Remote Avatar receives only a byte array which can be decoded into 60 single precision floating point numbers representing the X, Y and Z coordinates of the 20 joints. More details of the encoding/decoding process can be found in Section 5.8 of this thesis. Note however that the encoded data will be the raw data from the remote Kinect, thus the data will not have been processed into the sitting position, even if it has been for display on the remote computer it came from.

It should be noted that VMX has a third kind of avatar built into it for diagnostic purposes. It is called a Dummy Avatar and it functions as a clone of a given local avatar. When first created, a dummy avatar is passed a reference to a local avatar (exactly which local avatar can be changed at any time). This local avatar is source of all of the data the dummy avatar uses. Thus updating a dummy avatar requires no data to be passed to it; instead the dummy avatar will access the data of its local avatar to acquire the joint positions it needs. A dummy avatar will only differ from its assigned local avatar in that it can be independently positioned in the virtual environment, and that it can be forced into the sitting position even if its local avatar is not. The main purpose of the dummy avatar was for testing how the virtual environment accommodated multiple users, without needing multiple real users to connect to it. The following image shows dummy avatars in action. The local avatar they are based on can be seen in the background, in front of the screen.

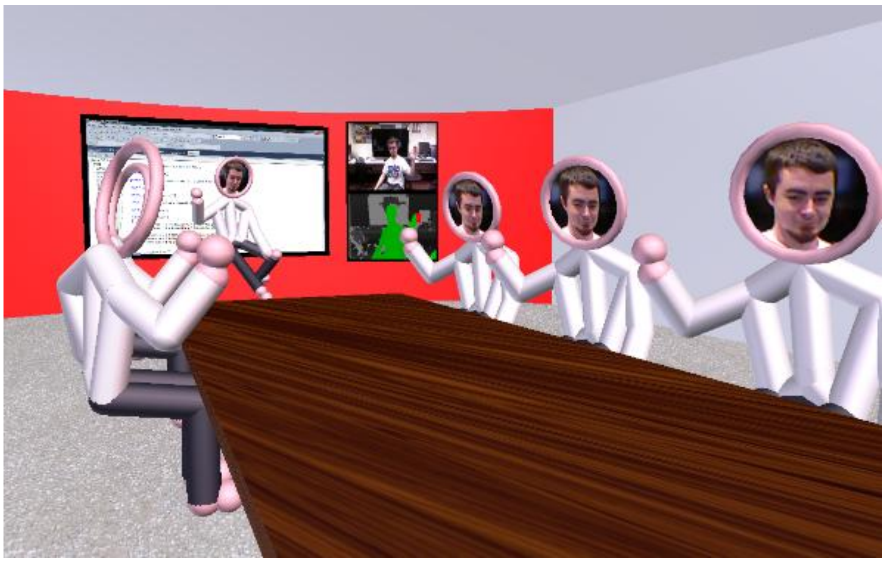


Figure 25: Dummy avatars showing colourisation.

5.4.6 Colourisation

Figure 25 also illustrates another feature common to all avatar types, colourisation. By default avatars are initialised with a single colour for all of their geometry except the face (white for local avatars, yellow for dummy avatars, and blue for remote avatars). In an effort to make the avatars involved in a meeting more visually distinct from one another, a system was added for giving each its own colour. The intent of doing so was to make it easier for users to identify which avatar belonged to which user while in a meeting. It is particularly useful when one user's avatar is facing away from another user, allowing the other user to identify whom the avatar belongs to without being able to see its face. The colours used are taken from the colour of the skin and clothes of the user the avatar belongs to, as seen by the Kinect's video camera.

The colourising algorithm works by picking out three colours: one for the upper half of the avatar (the 'shirt'), one for the lower half ('pants'), and one for the hands, feet and head ('skin'). The algorithm runs on each local avatar, on every program update until a colour for that avatar has been found for each skin, shirt and pants. The basic process for selecting a colour is not too different from the process for extracting a face texture. First, a joint where we would expect to find the desired colour on the user's body is selected. For the shirt colour the spine joint is used, the head joint is used to find the skin colour, and pants colour is taken from the left hip joint. To get the colour for each joint, the coordinates of that joint are taken, translated into depth image space using the Kinect libraries, and then translated again into colour image space. The colour image space coordinates are then used to determine the pixel on the colour image that corresponds to the original joint; this pixel is then sampled to obtain its colour. To improve the chance that the colours found by the algorithm will actually be representative of the actual colour of the user's clothes and skin, surrounding pixels are sampled as well and the colour values for all of the sampled pixels are averaged. Unlike the face capture algorithm, if part of the sample area is outside the bounds of the Kinect's camera image, then the pixels that can't be sampled are simply disregarded; no attempt is made to redefine the sample area's boundaries. If no part of the sample area is within the image bounds then the algorithm will pass back no colour and another attempt to find a colour will be made on the next program update. Once a colour is found it is converted into a 1x1 pixel texture. The texture is passed to the avatar which is then applied it to the appropriate joints for the shirt, pants or skin.

Remote avatars and dummy avatars also use colourisation. Remote avatars have their colours selected by their computer of origin and the colours are sent across the network link (more details are available in Section 5.8). Dummy avatars retrieve their colours directly from the local avatar they are using as a source; they will attempt to retrieve a colour from their local avatar on every update until they successfully get one.

5.4.7 Head Size

By default, the heads of the avatars have a radius of three units. Part of this project's objective is to look at ways in which a virtual environment can be exploited to do things that cannot be done in real world meetings. One possibility is to allow a user to increase the size of the heads of other avatars to get a better view of the faces of the people they are meeting with.

The method for dynamically changing the size of avatars' heads is fairly straight forward. The user is given control of the size of the heads of the avatars with keyboard controls. They are able to smoothly increase and decrease the head size by pressing the + and – buttons on the numeric keypad. When the user pushes the appropriate button a variable in the program is adjusted accordingly. The variable is essentially a factor to scale by; it is a single precision floating point number that defaults to 1.0 and is adjusted in increments of 0.01. This variable is accessed by each avatar while it is updating its joint data, during the phase in which it performs the additional calculations for head position. To achieve the size adjustments, the avatar simply creates a new Vector3 with all of its values set to equal to the scaling variable. This vector is then passed into the head model as a new scale vector, which is then incorporated into the head's transformation matrix.

This achieves the change in size to the head, but leaves a problem. The local origin of the head geometry sits at the centre of the face circle rather that the "neck" of the avatar. This means that when the scale of the head is increased the head will increase in size downwards at the same rate it increases in size upwards. Left unchecked, the result of this is that the head of the avatar intersects large parts of the body of the avatar. To compensate it is necessary to adjust the local translation of the head joint away from its default, Kinect provided location. It is only necessary to adjust the y-axis translation; this is because the x and z axis coordinates of the centre of the head are the same as the x and z coordinates of the bottom of the head (this wouldn't necessarily be true after the head's rotation has been changed, but we can ignore this as scaling is always applied *before* rotation when the head's transformation matrix is recalculated). The new y-coordinate is found with Equation 10.

$$NewY = OldY + R(S - 1)$$

Equation 10: Calculating the new Y-coordinate of a scaled head.

In Equation 10; NewY is the new y-coordinate of the face; OldY is the existing y-coordinate, R is the radius of the unscaled head as measured from the centre of the head to the outer edge of the bordering torus; and S is the scale factor. Using this method means that when the size is adjusted, the avatar's head appears to only grow up and outwards, not down into the avatar's body.

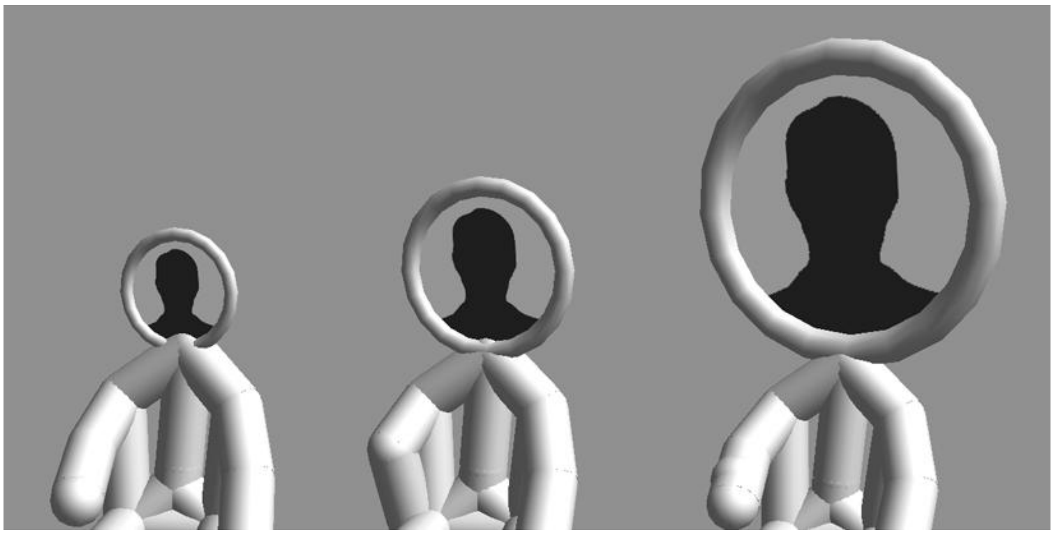


Figure 26: Varying avatar head sizes

This feature of being able to adjust the head size of the other participants in the meeting ties into the theme of finding ways to make use of the fact that the meeting is taking place in a virtual world, and not bound by the laws of physics.

5.5 Gesture Recognition

The Kinect's skeletal tracking abilities provided the opportunity for experimenting with gesture controls for VMX. Because the Kinect SDK itself has no built in support for gesture recognition, a system had to be built into VMX.

VMX uses two kinds of gesture recognition: hand gestures, and finger gestures. Hand gestures involve broad movements of the hands and arms, and gesture detection is based directly on Kinect skeleton data. Finger gestures are more precise and involve the positions of a user's fingers, which must be found by analysing the new depth image stream from the Kinect.

5.5.1 Hand Gestures

Part of this project calls for the evaluation of the usefulness of using gestures to control elements of the program while doing a presentation with Kinect. To achieve this it is necessary to implement a system for picking up specific motions as gestures. Microsoft's Kinect SDK has no built in functionality for doing this.

The gesture recognition system in this project is fairly simple, utilising the joint location information provided by the Kinect over time to determine if gestures are being performed.

The gestures used in this program all revolve around the relative position of the user’s hands to each other and the user’s body. The principle hurdle appeared during the implementation of this system was distinguishing between times at which a user is specifically trying to perform a gesture, and when they are simply using ordinary body language and have no intention of doing a gesture. In order to get around this problem the gestures that have been built into the system are designed to require body movements that a user is unlikely to perform under normal circumstances. Exactly what those movements are, has gone through multiple iterations over the course of the project, each presenting its own advantages and disadvantages. All of the gestures that have been created are used to modify the image that is displayed on the presentation screen in the virtual environment.

The first iteration provided a means of panning across an image on the presentation screen. The gesture itself required the user to use their hand to reach behind themselves so that the hand of their avatar went into the presentation screen in the virtual environment. Once the hand was picked up as being inside the screen, moving it up and down or left and right would cause the image on the screen to be “dragged” along with the hand; thus giving the user the ability to scroll the image. This gesture proved to have a few drawbacks. While functional, it wasn’t entirely comfortable for the user to perform the gesture while simultaneously watching the computer screen in front of them to see what they were doing. Scrolling left and right could also prove troublesome as often that would result in the user’s hand moving behind their body and out of sight of the Kinect. This approach was also incompatible with the intended approach of having a solid, real world screen behind the presenter to correspond to the virtual one.

The second iteration had the user hold at least one of their hands no less than forty centimetres in front of themselves before any gesture recognition would occur. Once beyond forty centimetres, moving their hand up, right, left or down would function in the same manner as in the previous iteration, scrolling the image. In addition to scrolling, support for a zooming gesture was added, it was performed by using two hands and moving them either closer together to zoom out, or further apart to zoom in (analogous to pinch zooming on touch screen devices). While this approach solved the main problems with the former approach, it revealed new problems. In particular, it was difficult for a user to withdraw their hands to end the gesture without unintentionally zooming or scrolling the image in the process.

The third iteration attempted to rectify this problem by including an additional requirement that needed to be met before gestures would be recognised. It called for the user to show their palm, with fingers pointing upright to the Kinect in order for gestures to be recognised. The mechanism for determining when the user was doing this was simple, the system checked if the user’s hand was above their wrist joint on their Kinect skeleton. Unfortunately this method proved unreliable and only partially effective. The act of a user shifting their hand so their palm was not to the camera was still enough to cause some unintentional scrolling (though much less than before). Furthermore a limitation in the Kinect’s mechanism for deciding the location of the user’s hands relative to their wrist was revealed. Specifically the direction from a user’s elbow to their wrist appeared to play a significant part in where the Kinect positioned a user’s hand. In practice this meant if a user’s arm was pointing towards the ground their hand would often appear below their wrist on their Kinect skeleton, regardless of whether or not it really was.

The final and current iteration of this system requires the user to use both hands whenever they want to perform a gesture. One hand functions as the “gesture enabler” and one as the “gesture performer”. To perform a gesture, both hands must be held out in front of the body. The gesture performer is the hand that when moved causes the image on screen to scroll. The only function of the gesture enabler is to allow the actions of the gesture performer to be recognised. Therefore moving the gesture enabler back towards the body will allow gesturing to be disabled without accidently causing any gestures to be picked up in the process. This system works well but has a few drawbacks. The first is that with the one of the user’s hand tied up permanently as the gesture enabler, the zooming gesture can no longer depend on having two hands available; meaning that a new way of telling the difference between a zoom gesture and a pan gesture is needed. The other drawback is that the system requires more coordination and practice to use than the older iterations and is not very intuitive.

The system functions by first picking out the locations of the left hand, right hand, and centre shoulder joint of the user’s Kinect skeleton. The shoulder centre joint provides a location for the user’s body which is then used to determine if the user is holding their hands out far enough in front of themselves for gesture recognition to occur. In the graphics system each hand that is being held out far enough for gesture recognition is coloured red so the user can tell if they are in gesture mode. If both hands are in position for gesture recognition then recognition mode begins. In order to read gestures, the motion of the gesture performing hand must be analysed; its current position and a slightly older position are needed to do this. Consequently when gesture detection first begins, the only action is for the position of the gesture performer to be recorded into the variable holding the gesture performer’s ‘old position’. On subsequent iterations of the program loop true gesture recognition begins, in which the gesture performer’s current position is compared to its old position to determine if and how it has moved. If the gesture performer is deemed to have moved up, then the screen image is scrolled up; and if it has moved down then the image scrolls down. Horizontal scrolling is performed similarly by moving the hand left and right. Finally the current location of the gesture performer is stored into the old position variable for use on the next program loop iteration.

Originally, there was no way to perform a zooming gesture in this system. Later, the system was modified to allow zooming. The modifications involved adding a second distance threshold for the gesture enabling hand that was further away from the user’s body than the first. When this new threshold is crossed the user’s avatar’s hands will turn blue. When in this mode, the user can move their gesture performing hand up and down to zoom in and out.

If at any point the gesture enabling hand is withdrawn back close to the user’s body then gesture recognition ends and future gesturing will have to go through the process of first storing a new value for the gesture performer’s old position before beginning gesture recognition again.

5.5.2 Finger Gestures

The project design called for a way for a presenter to accurately point at positions on an object in the Kinect camera's field of view. To achieve this, a system for detecting and reporting the position of a presenters fingers or at least one extended finger needed to be put in place.

In Chapter 2 of this thesis, the work of Oikonomidis et al in the area of tracking finger positions was examined. The algorithm that was created by those researchers was capable of successfully identifying the positions of a user's finger using data acquired from a Kinect device. While effective, in the context of this project their algorithm is not entirely suitable for use; the main problem being that it is computationally expensive. Its creators required a powerful, modern system and needed to exploit the GPU to even get close to the real-time speeds that would be required for virtual meeting software to be useful. It is desirable for VMX to be able to run a wide variety of systems, including those that do not have GPU's that support this kind of non-graphics related application, so this kind of algorithm is not ideal.

The creators of the algorithm described it as being of a class of algorithm that they called "model-based". They also described a second class of algorithm called "appearance-based". Algorithms of this class map certain image features to particular hand positions that are specifically defined in the program. These algorithms are described as being well suited for problems where there is a small number of known hand positions that need to be detected. This is consistent with the requirements of this project where the only hand position that needs to be recognised is a pointing gesture.

The algorithm applied in this project for gesture recognition is custom made and of the "appearance-based" class. It functions by identifying the location of the hand to perform analysis on, and then passing that information on to systems which utilise that position in tandem with raw depth data from the Kinect to identify certain gestures.

The process of finding the location that a user is pointing at begins by finding the direction in which they are pointing. This is done by acquiring Kinect's skeleton space co-ordinates for the user's right wrist and right hand. These are subsequently translated using the Kinect SDK into their corresponding depth image space co-ordinates. The new co-ordinates are then treated as two dimensional vectors in an equation in which the value for the wrist is subtracted from the value for the hand. This provides a third vector that indicates the direction from the wrist to the hand. It is this vector is used as the direction in which the user is pointing and will hereafter be referred to as the “direction vector”.

Having acquired the direction, it is then necessary to determine where the user’s finger ends in order to get the exact location where they are pointing. In this algorithm the end of the finger is assumed to be the furthest point from the provided location of the hand in the direction of the direction vector. Before this point can be found the vector that is perpendicular (to be referred to as the “perpendicular vector”) to the direction vector is derived. Both the direction vector and the perpendicular vector are then normalised. The two vectors are then used to determine points in the raw depth image data to sample. This is done in a loop where on each iteration, the normalised perpendicular vector is scaled by a factor between -25.0 and 24.5 changing in increments of 0.5. This loop is nested inside a second in which the direction vector is scaled by a factor between 0 and 49.5 also changing in increments of 0.5. The scaled vectors are then summed together along with the depth image space co-ordinates of the hand to provide the pixel on the depth image to be sampled. The sample point is then checked to ensure that it lies within the borders of the depth image before being translated into an index into the array that contains the raw depth data. The raw data is accessed and checked. If the raw data indicates that the sampled pixel is not showing a part of the user then the algorithm progresses to the next sample. If there is a part of the user in the sampled pixel then the location of that pixel is recorded and the remaining iterations of the inner loop are skipped (this is done to reduce the total number of samples taken, speeding up the algorithm). If the inner loop completes without finding any pixels with the user in them, then it is assumed that the end of the user’s finger has been reached and the last recorded location of a pixel displaying the user is taken as the location at which the user is pointing. Figure 27 illustrates this process.

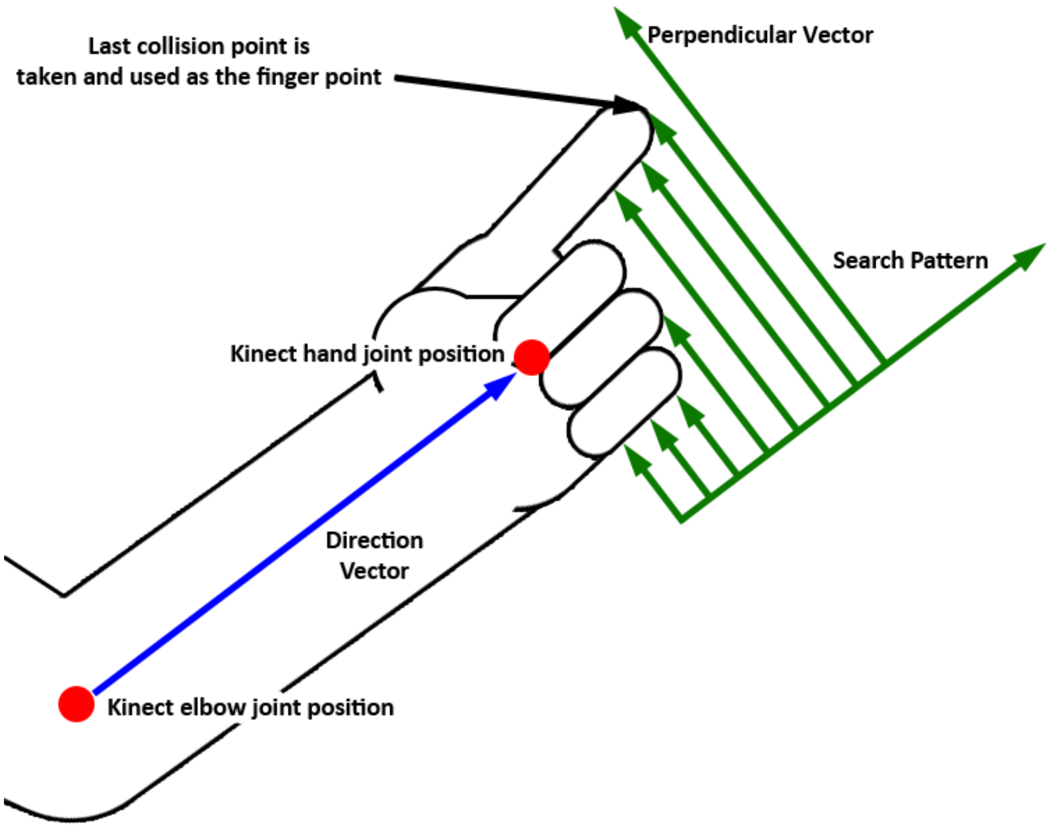

Last collision point is
taken and used as the finger point
Perpendicular Vector
Search Pattern
Kinect hand joint position
Direction
Vector
Kinect elbow joint position

Figure 27: Diagram of finger point searching algorithm.

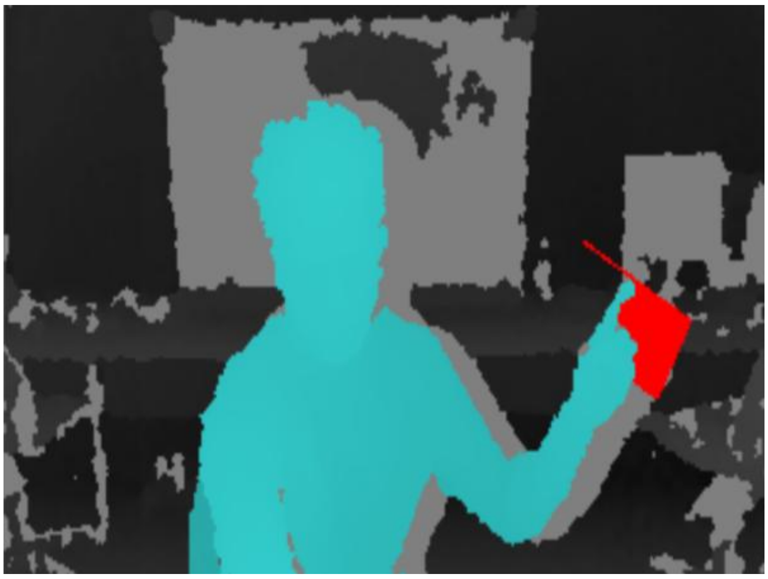


Figure 28: Depth image showing the finger search algorithm in action

The algorithm can be seen in action in Figure 28; it shows a depth image with the pixels identified as belonging to a user coloured light blue, and the pixels that were actually sampled by the algorithm coloured red.

The rational of changing the scaling values in increments of 0.5 rather than 1 is to ensure that all pixels in the sample area are found, regardless of how well the direction vector is aligned to the grid of pixels. The drawback of this approach is that some pixels may needlessly be sampled more than once.

One final step is taken before the co-ordinates the user is pointing at are passed back to the program. Because of the way that the Kinect works, there tends to be a degree of jitter in the depth data. In the depth image this manifests as rough edges with a constantly changing shape on objects. Unchecked this will cause the exact point at which a user is pointing to move about constantly, even is the user is perfectly still. To reduce the impact of this, a rolling average of the pointer location of the last 15 frames is used as the final location.

Two values are ultimately passed back to the program. One is the final smoothed point, which is in depth image space; the other is the same point but translated with the Kinect SDK into colour video image space.

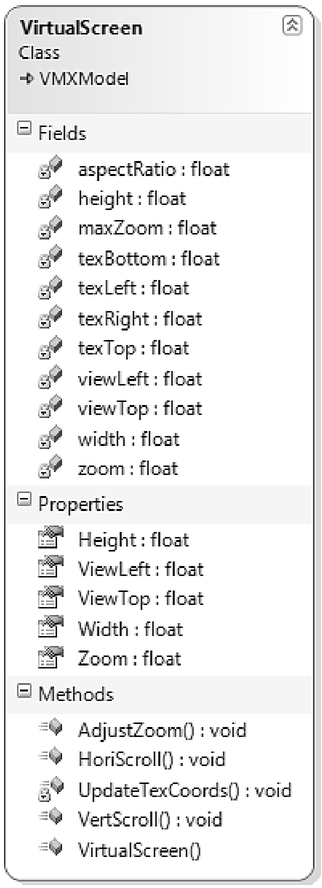
5.6 The Display Screen

Within the virtual meeting room there is a large screen that sits on one of the walls of the room. This screen is intended to function as a place for a user to make a presentation to other participants in the meeting. The screen requires special functionality that is not shared with other objects in the virtual environment. It is also designed to be used in correspondence with a real world counterpart when possible, essentially allowing a user to perform a presentation and have it entirely reproduced in the virtual environment to be seen by the other participants.

5.6.1 The VirtualScreen Object

Because the virtual screen requires additional functionality over other objects in the virtual environment, it was ideal to create a new class to handle this functionality. This class is called VirtualScreen.

The VirtualScreen class is a subclass of VMXModel. Unlike a general VMXModel object, a VirtualScreen object is always contains only a single Geometry object, specifically a textured quad. All of the new functionality of the Virtual screen objects revolves around the ability to manipulate the texture currently used on the screen to give the appearance of zooming and scrolling. This is done by implementing methods that control the texture coordinates of the four vertices of the quad using simple parameters.



VirtualScreen
Class
→ VMXModel

Fields
- aspectRatio : float
- height : float
- maxZoom : float
- texBottom : float
- texLeft : float
- texRight : float
- texTop : float
- viewLeft : float
- viewTop : float
- width : float
- zoom : float

Properties
- Height : float
- ViewLeft : float
- ViewTop : float
- Width : float
- Zoom : float

Methods
- AdjustZoom() : void
- HoriScroll() : void
- UpdateTexCoords() : void
- VertScroll() : void
- VirtualScreen()

Figure 29: VirtualScreen class structure.

The parameters are Zoom, Horizontal Scroll, and Vertical Scroll; all three are defined as floating point numbers. By default zoom is set to 1.0 and both scroll values are set to 0.0. The scroll values determine the texture coordinates of the top-left vertex of the screen. A combination of the scroll and zoom values determine the texture coordinates of the other 3 vertices in the screen, as shown in Figure 30. Note that the Zoom parameter is an inverse scale value, with 0.5 representing a doubling in size.

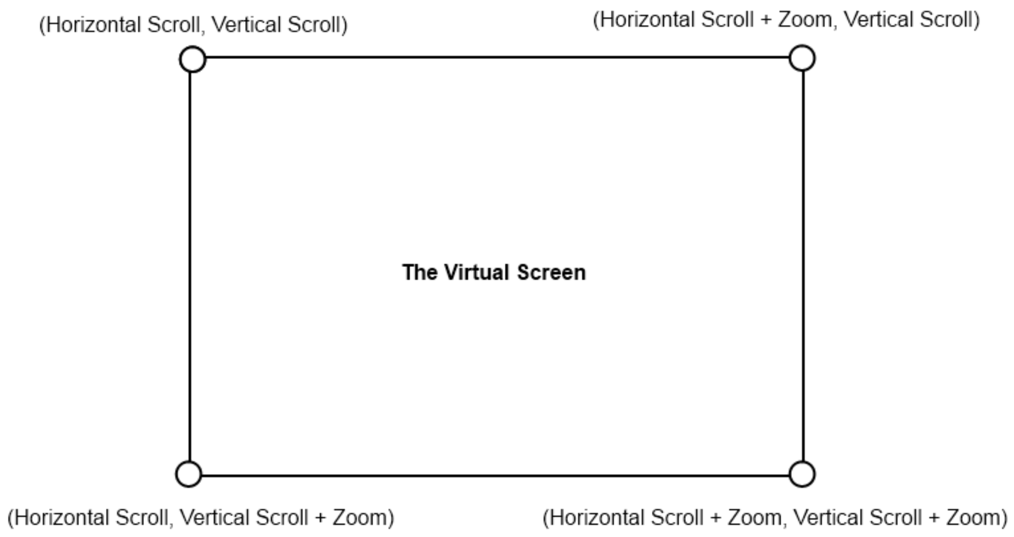

(Horizontal Scroll, Vertical Scroll)
(Horizontal Scroll + Zoom, Vertical Scroll)
The Virtual Screen
(Horizontal Scroll, Vertical Scroll + Zoom)
(Horizontal Scroll + Zoom, Vertical Scroll + Zoom)

Figure 30: Calculation of VirtualScreen texture coordinates.

To make sure that the image is displayed on screen properly, without strange effects like mirror images or tiling, the parameters must be limited. The limits imposed on the scrolling parameters are the same, they must not be less than zero, or large enough that their value plus the current value of the Zoom parameter is greater than one (if the value of the Zoom parameter changes, then this limit must be recalculated). The Zoom parameter must at all times be greater than zero (if it was zero then all of the texture coordinates on the screen would be equal so the screen would show nothing but a solid colour; and if it was less than zero then the image would be inverted on both dimensions). The other limit on the Zoom parameter is that it must not be greater than one (if it was the image would appear repeated and tiled across the screen.)

5.6.2 The Real World Screen

The virtual screen is designed to be used in conjunction with a real world screen. The real world screen would be situated behind the user in the Kinect camera's field of view. The purpose of having this screen is to give a user a real world reference to interact with when using the VMX software to perform a presentation to other users in the virtual environment; with the Kinect device capturing the user's interactions with the screen and manipulating their virtual avatar to reproduce those interactions with the user's avatar against the virtual screen. To do this key information must be acquired, including the image that is currently displayed on the real world screen, so that it can be reproduced on the virtual screen; and the user's physical position relative to the screen, so that their avatar's position relative to the virtual screen can be closely matched.

Throughout this project several mechanisms have been used to ensure that the image on the real world screen matches the image on the virtual screen. Originally it was done with use of HTML documents and web pages that would be displayed in a web browser on the real world screen, and separately rendered to an image that could be used as a texture for the virtual screen, a third party library was used for this but it proved to be slow and unreliable (often failing to render anything at all). Consequently, this was later was shifted to using a direct screen capture of the image displayed on the real world screen; the screen capture would be converted into a texture and applied to the virtual screen.

The mechanism for capturing the image on the real world display screen in the current implementation of VMX utilises the System.Windows.Forms and System.Drawing libraries. A method called CaptureScreen is used to perform the actual capture. The method works by calling the System.Drawing.Graphics.CopyFromScreen method. This method is passed the pixel coordinates of the top left corner of the real world screen, and the full resolution of the real world screen. This results in a full screen capture of whatever is displayed on the real world screen. This data is first placed into a bitmap object; from there it is saved into a memory stream in the PNG format. This memory stream is then passed to a Texture2D method that can read the stream and convert the PNG format data to a Texture2D object. The resulting Texture2D can be used as the texture for the virtual screen.

To maximize the usefulness of the real world screen, there needs to be some way to determine how a user is interacting with it. Specifically it is necessary to have an idea of where the screen is relative to the user so that the relationship between the user’s avatar and the virtual screen can be made to reflect the relationship between the user and the virtual screen.

Finding the position of the real world screen relative to the user is a tricky problem. It requires that the position of the real world screen be detected. Three main ways of doing this have been tried over the course of this project and none of those methods has proved perfect, though each has shown to be effective in certain conditions. All of the methods for detecting the location of the screen have been designed to output the same information. That information is the coordinates of the four corners of the real world screen in at least one of the coordinate systems used by Kinect.

The first method for acquiring the coordinates was also the simplest. It was to simply have the user input the coordinates of the screen’s corners into the program manually; the coordinates would be in Kinect skeleton space. Since the units used in Kinect skeleton space coordinates correspond to real world metres, it is not as difficult to estimate the location of the corners as one might think;

however there would likely be a degree of trial and error involved when doing this. Regardless, the potential difficulty in getting accurate coordinates is not the primary drawback of this method. The main problem with it is that if the screen itself is moved, then the coordinates would have to be re-entered by the user. Worse if the Kinect itself was moved, then the entire coordinate system would be thrown out, and the new set of coordinates would likely end up being dramatically different from the original set. Any movement to the Kinect would result in this, including use of the Kinect’s own tilting mechanism. Because of this problem, it rapidly became apparent that there would need to be a system for detecting the location of the real world screen automatically.

The first approach to automatic detection was to use the depth data from the Kinect. The theory was that depth image stream could be analysed to find the location of the screen. This could be done by simply analysing the contents of the depth image to look for a large flat surface in the background behind the user. The code for doing this would have resembled the code used for the current solution which will be discussed next. When the time came to actually implement this method however, a fatal problem immediately became apparent.

The test screen that was being used for development of this software was an ordinary 52″ LCD television. Like many televisions the screen itself had a slightly reflective finish. This reflective finish revealed one on the primary drawbacks of the Kinect’s method of generating its depth image data. The infrared radiation from the Kinect sensor would be reflected away from the device by the screen. This rendered the depth sensor useless for producing accurate data about the depth of most parts of the screen (the centre of the screen of course reflected the infrared light directly back at the Kinect device so depth data could be acquired for that area of the screen, but this was useless for finding the X and Y coordinates of the screen’s corners). This problem would not have been present on a projector screen (which is designed to reflect light evenly in all directions; however there would be another problem with a projector screen in that the edge of the flat area in the depth image might not fully match the area of the screen that actually had an image projected onto it.

Figure 31 demonstrates the problem with reflective screens. In the centre of the lower half of this image there is a white rectangular shape with a grey blob in the middle. This rectangular area is the television as seen by the Kinect depth camera. The white areas are the parts of the screen for which the Kinect cannot determine the depth, because they are reflecting the infrared light away from the Kinect device. The grey blob in the middle is the area of the screen that is reflecting the infrared light directly back at the Kinect's depth camera. To the right of the image a second smaller, computer screen can be seen exhibiting a similar problem.

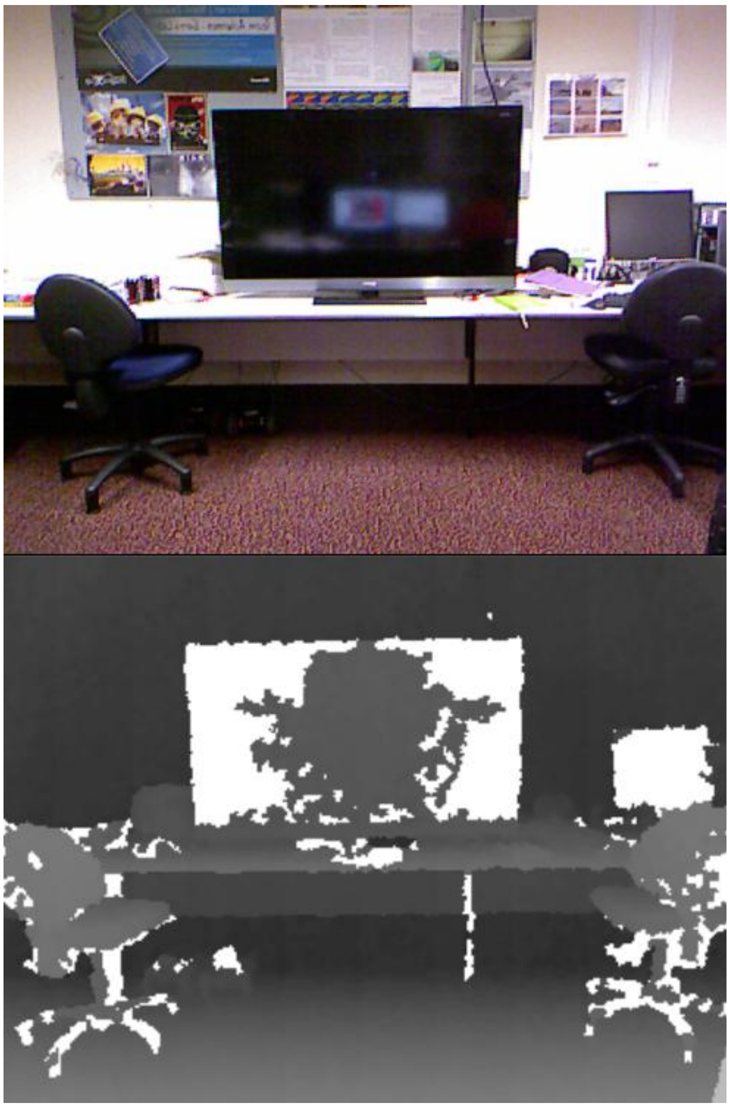


Figure 31: The bottom image is the TV screen as seen by the Kinect's depth camera, the top images is the same scene as seen by the Kinect's colour camera.

These problems resulted in the third and current solution for detecting the location for the screen. This solution uses simple image recognition principles. The system works by having the user issue a command to VMX to search for the corners of a real world screen in the background of the Kinect’s image data. When this command is received VMX will run the algorithm for detecting a screen. The algorithm goes through a six stage process; the first two stages involve finding the left and then the right side of the screen, the other four stages are used to find each of the four corners of the screen.

The algorithm functions by painting the entire screen red and then seeking the boundaries of the red area in the Kinect’s colour camera image. The first step in the algorithm that must be carried out is to make the screen appear red. The program uses windows forms to do this. A large window is created. The size of the window is made to match the resolution of the real world display screen being used. The window is also setup to be borderless, meaning that there will be none of the feature often seen on a typical window (e.g. a title bar, close/maximise/minimize buttons, resizable edges etc.) Instead the window will simply appear as a solid colour rectangle. The window’s background colour is then set to be red. The final step is to position the window so that it appears solely on the display screen. Obviously this method requires that the display screen being used is connected to the same computer that VMX is being run on, this is a requirement of both this algorithm and the algorithm for capturing and sending the contents of the screen to the other user’s virtual screens in the meeting.

Once the screen is red, the algorithm can begin searching for it in the Kinect’s camera image. The first step of doing this is to select a point on the image to start the search. There are several restrictions that are placed on how a real world display screen must be placed in order to use it with the Kinect. The first of the these restrictions is that VMX requires that a real world display screen must be set up such that the television is directly in front of the Kinect sensor, facing it approximately head on, and be behind the user. The second restriction is that to be useful the screen must appear reasonably large within the Kinect camera’s field of view (more than half of the horizontal range), and the screen must appear entirely within that field of view. Figure 32 shows a screen that has been positioned to meet these requirements.

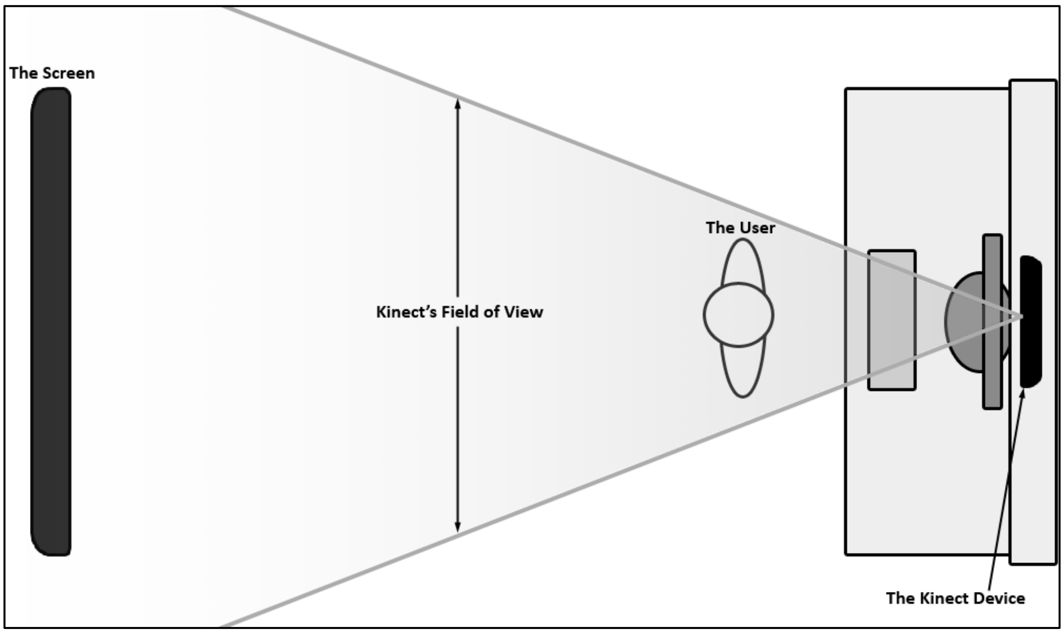


Figure 32: The spatial relationship between the Kinect device and the real world display screen

These restrictions allow the screen seeking algorithm to make certain assumptions about where to start searching for the screen, specifically it can safely assume that the some part of the screen will appear in the centre of the Kinect's camera image. However it is likely that when using the screen to do a presentation the user will position the screen at a level that allows them to easily point at locations on the screen. This means that the screen will be slightly above being vertically centred in the Kinect's field of view. For this reason the algorithm selects a point in on the image from the Kinect colour camera that is horizontally centred and a third of the way down the screen.

To determine if the display screen in visible on the selected point on the image, the colour of the pixel at that point is checked to see if it is red. A simple method called PixelIsRed is used to perform this check. This method takes the colour value of the pixel as a parameter, and returns a Boolean that is true if the pixel is deemed to be coloured red. Because of the properties of the screen and the Kinect camera, even though the screen is set to display only fully red pixels (with RGB colour values of (255,0,0)) it is highly unlikely that the camera will report the colour of the screen as perfectly red (for example, if the camera image is over exposed the screen can begin to appear slightly white coloured). This means that it is not sufficient to simply check if the colour of a pixel has the value (255, 0, 0) when determining if it is red. So, when deciding if a pixel is red, the PixelIsRed method simply ensures that the red component of the colour is above a certain threshold and that the blue and green components are below that threshold.

Even though it is reasonably certain that any screen that the user intends to use will appear at the selected starting point in the image there is one additional consideration that needs to be taken account of. It is likely that the user is going to be sitting between the Kinect device and the screen when they issue that command to run the screen seeking algorithm. This means there is the possibility that they will be partially obstructing the view of the screen when the algorithm begins. To account for this measures are taken in the first stages of the algorithm to handle not being able to see the screen immediately. Figure 33 illustrates this situation.

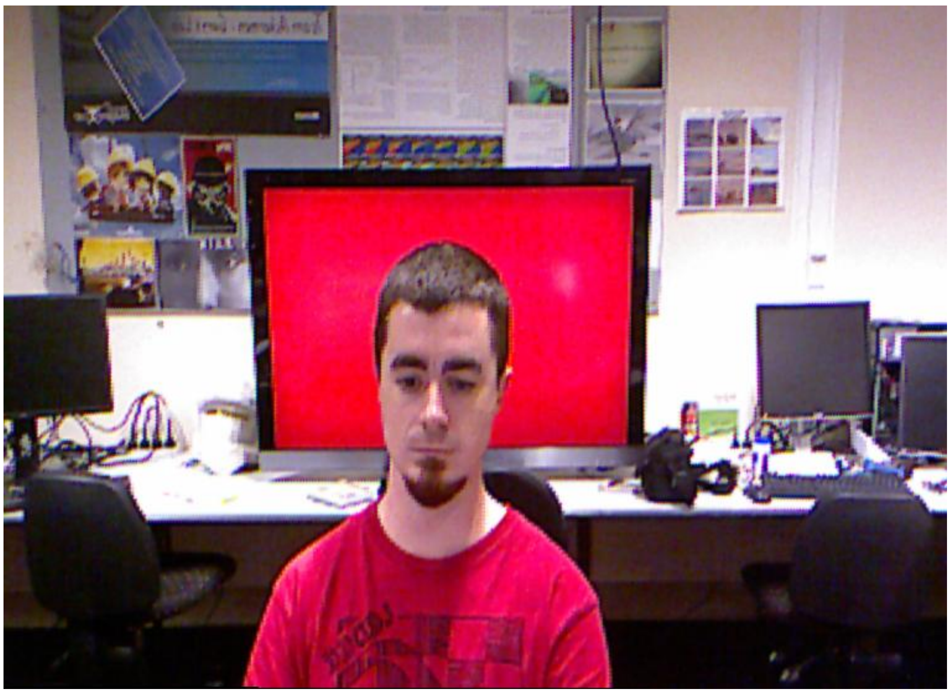


Figure 33: A user partially obstructing the display screen

The first stage of the algorithm moves outward to the left of the selected starting point in search of the left hand edge of the screen. This is where the measures to handle a screen obstructing user come into play. After the pixel at the starting point is checked the algorithm will move the sample point one pixel to the left and check that pixel, regardless of whether the first pixel was red or not. This pattern of moving one pixel to the left and checking again will continue until one pixel does end up being red. Once a red pixel is found the algorithm assumes that it has found the screen. Once this happens the algorithm will continue as before, except that the pixel two steps to the left of the sample point is checked to see if it is black (using a method called PixelIsBlack which functions in the exact same way as PixelIsRed, simply with different thresholds). The reason for this is that the borders of the screen are coloured black, so the algorithm recognised the edge of the screen by finding a place where there is a red and a black pixel in close proximity to each other. The reason for checking the pixel that is two steps over from the sample point, rather than simply the neighbouring pixel is to account for the fact that the colour of the pixel that sits right on the edge of the screen may be a blend of the red of the screen and the black of the screen border. Having the additional check for the black screen border makes the algorithm more robust. It allows the algorithm it to deal with objects partially obstructing the Kinect's view of the screen (so long as those objects are not black); it also reduces the chance of other red objects being misidentified as the screen. The trade-off of this additional robustness is that the screen must have a black edge in order to be recognised, though it would be trivial to change the colours that are searched for simply by changing the threshold values in the PixelIsRed and PixelIsBlack methods.

Once the algorithm has identified a point where there is a black pixel on the left and a red pixel on the right, this point will be stored as the location of the left edge of the screen for later use. From this point the algorithm searches for the top left and bottom left of the screen. Before this happens however, it searches for the right edge of the screen. The process of finding the right edge is identical to the process of finding the left edge, only the sample point on the image moves right from the original starting point with each pixel colour check, and the black pixel must appear on the right of the red pixel. If either the search for the left edge or the search for the right edge fails to result in an edge being found, then the algorithm will report that it was unable to find a screen in the camera image, and will then return.

Once the edges are found the algorithm moves on to find the four corners of the screen, starting with the top left corner. To begin the search for the top left corner, the algorithm will take the point that was stored for the left edge of the screen as its current sample point. From there the algorithm will loop through pixels in a particular pattern in order to find the top of the screen. The pattern is complex and involves a loop that on each iteration will sequentially seek through the neighbouring pixels of the current sample point. As soon as it has found a neighbouring pixel that is red, that pixel will be made the new sample point and the loop will reset, checking through the neighbours of the new starting point.

When seeking the top left corner, the algorithm will look first at the pixel to the left of the current starting point. This is done to handle circumstances where the screen isn’t perfectly aligned to face the Kinect camera; in this situation the edge of the screen will appear on a slight angle (i.e. not straight up and down). By checking first to the left of the sample point, the algorithm can ensure that if the edge of the screen angles away to the left on the camera image, then the current sample point will still continue to lie on it. There is no limitation on how many times the sample point can be moved to the left in a row. This is despite the fact in order for the edge of the screen to angle away at more than one pixel to the left for every pixel upwards it would have to be at such an extreme angle to the Kinect camera that the data resulting from this algorithm would be useless to any other part of the program. The benefit of allowing these repeated moves to the left is that it makes it possible for the algorithm, under some circumstances, to recover if the location of the left hand edge of the screen was misidentified.

Figure 34 shows how allowing unlimited moves to the left can permit the algorithm to recover if it misidentifies the location of the edge of the screen. The X on the diagram shows the point of the screen that was for some reason mistaken for the edge of the screen (this could have been caused by an obstruction between the Kinect and the edge of the screen that is no longer present). The line coming from the X shows the path that the algorithm will take while it searches for the top left hand corner.

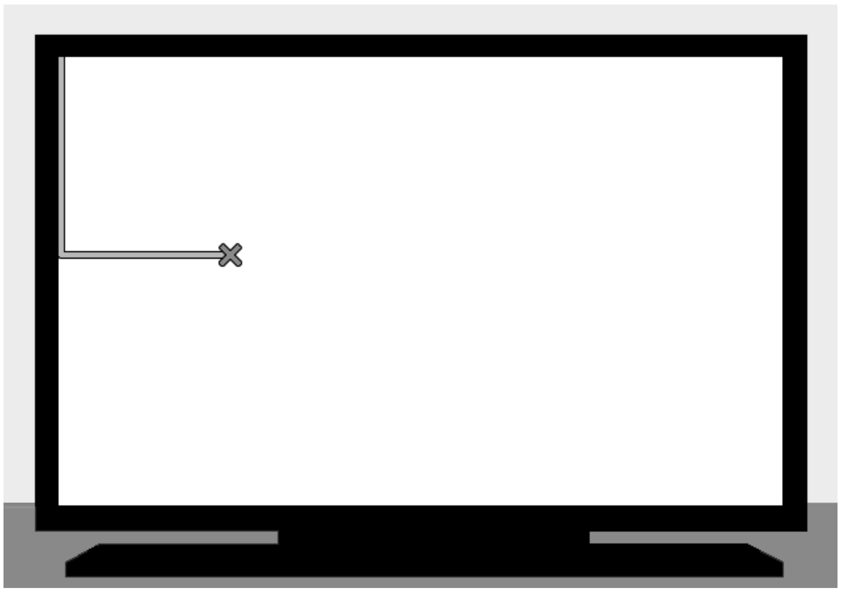


Figure 34: Recovering if the location of the edge of the screen was misidentified.

If the pixel to the left of the sample point is not red then the next neighbour to be checked is the pixel above the current sample point. This is the natural direction to look in as the top left corner will be at the top of the left edge when the screen is perfectly aligned to the camera. It is this pixel that will always be selected as the next sample point.

The final neighbour that will be checked is the pixel on the right of the sample point. This is done as a last resort only if the other two neighbours are not red. It may seem counter intuitive to search to the right when the algorithm is looking for a point on the far left of the screen, but there is a good reason. This check to the right serves to prevent the algorithm from prematurely selecting a point it believes is the top left corner in situations where the screen isn't aligned to face the camera perfectly and as a result the left edge of the screen appears to drift on an angle to the right with increasing height.

There is a special limitation on the sample point being moved to the right. It can only happen if both the pixel on the right is red, and the pixel above that pixel is red. If this special condition wasn't imposed then when the algorithm did reach the top of the screen, it would get stuck in a loop of first changing the sample point to the pixel to the right of the top left corner, and then back to the top left corner again. If after checking all of these neighbouring pixels (there is never a need to check the pixel below the current sample point) the algorithm is unable to find a new red pixel to be the new sample point, then the current sample point is taken to be the top left corner, the location of that point is stored and finally the loop ends.

Table 1 shows each of the potential situations that need to be handled in the search for the top left corner of the screen. The table illustrates each situation with a diagram. Each square in the diagrams represents a pixel on the Kinect’s colour video image. The black squares represent pixels that show the edge of the screen, the white squares represent pixels that show the screen itself, and the grey squares represent the pixel that lies under the current sample point. The arrows on the diagram indicate which neighbouring pixels will be checked to see if they should become the next sample point in each situation. A thin grey arrow indicates that the pixel they point to would be rejected as the next sample point, a thick black arrow indicates that the pixel will be selected.

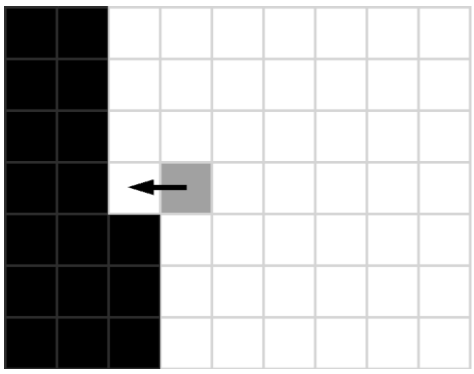
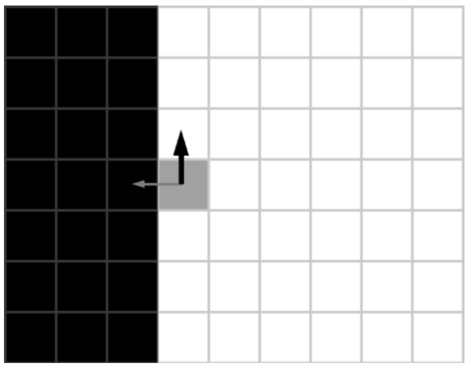
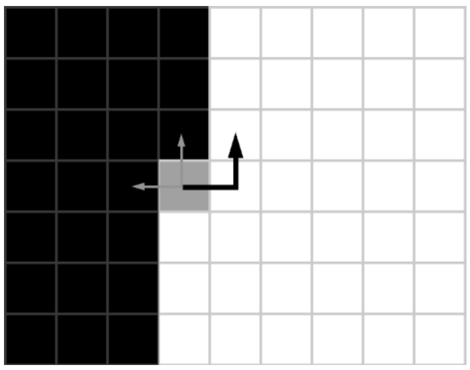
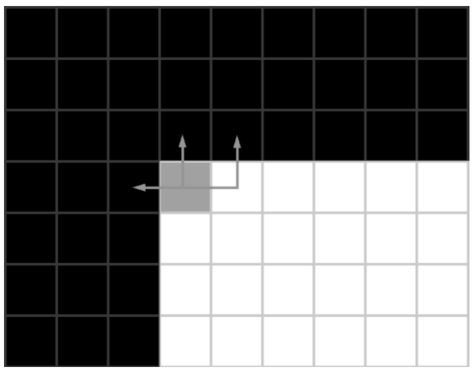
	**First check passes.** This diagram illustrates a situation where due to the angle of the screen to the Kinect camera, the edge of the screen appears on a slight angle. When searching for the top left corner the algorithm will always attempt to stay hard up against the left edge of the screen, so the left pixel is checked first, and in this case the pixel to the left is part of the screen so it is selected.
	**Second check passes.** In this situation there is no apparent angle on the edge of the screen, the pixel to the left is not part of the screen so it is rejected. As a result the algorithm checks the pixel above the current sample point. In this case the pixel above is part of the screen so it will select that pixel as the new sample point.
	**Final check passes.** This situation is similar to the first situation shown, differing in that the angle of the edge of the screen runs in the opposite direction. In this case the checks for both the pixel to the left, and above the current sample point failed to find the screen, so the algorithm checks the pixel to the right and the pixel above the pixel to the right. In this case they are both part of the screen so the pixel on the right is selected as the new sample point.
	**No checks pass.** This final situation demonstrates why it is necessary to check both the pixel to the right and the pixel above it. Here we can see that the sample point is at the top left hand corner of the screen and needs to go no further. All three of the checks failed to find anywhere else to go so the algorithm will terminate and correctly return the current sample point as the location of the top left corner of the screen. If the check above to the right was not done then the algorithm would end up moving to the right in this situation and away from the correct location of the top left corner.

Table 1: Potential cases when searching for the top left corner

Once the location of the top left corner is found, the locations of the other corners are searched for in the following order: bottom left corner, top right corner, bottom right corner. The algorithms for finding these corners are very much the same as the algorithm for finding the top left corner, but do differ in a few details. First and most obviously is that the algorithms that find the corners on the right hand side of the screen use the value stored for the location of the right hand edge of the screen instead of the left hand edge. The other difference in the algorithms for each of the corners is the order in which the neighbours in differing directions of the sample point are checked.

Table 2 illustrates the order in which each neighbouring is checked for each of the different corners of the display screen. The squares on each diagram in the table represent the pixels on the Kinect colour video image.

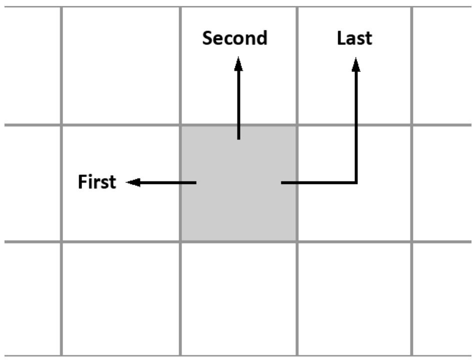
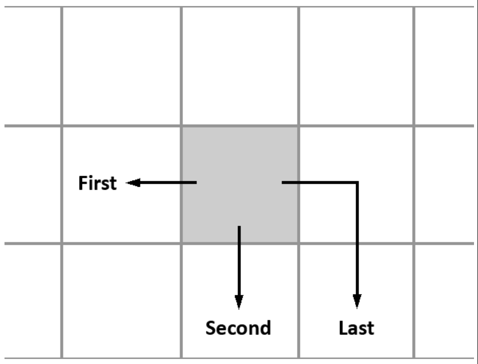
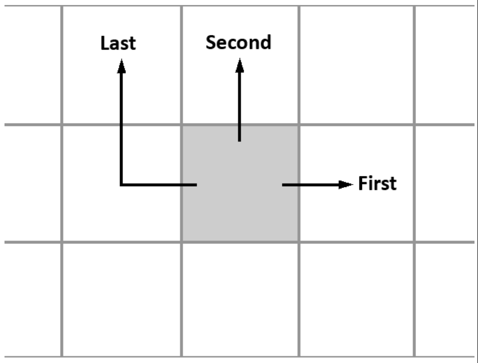
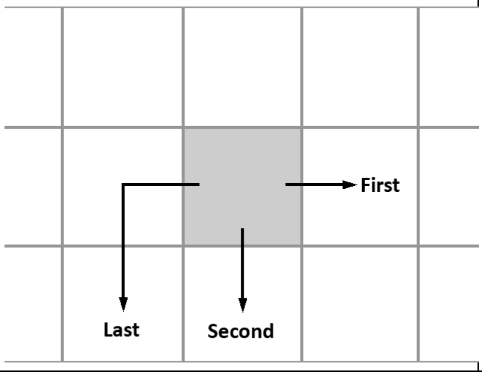
	**Seeking the top left corner.** In this situation the algorithms first priority is to stick to the left hand edge of the screen, so the first check is to the left. The next priority is to move towards the top of the screen so the second check is upwards. The final priority is to check that the top of the screen has been found when it is not possible to move up by checking the pixel to the right and above.
	**Seeking the bottom left corner.** As in the top left situation the first priority of the algorithm here is to stay on the left hand edge of the screen. Unlike the last situation the algorithms second priority is to reach the bottom of the screen, so the second check is downwards. The final check confirms that the bottom of the screen has been reached by checking the pixel to the right and below.
	**Seeking the top right corner.** Being on the right hand edge of the screen means that the first priority of the algorithm when searching for the top right corner is to stick to that right hand edge of the screen, so the first check done on the pixel to the right. The next priority is to move to the top of the screen so the second check is upwards. The final check to confirm when the top of the screen is reached is to the left and upwards.
	**Seeking the bottom right corner.** In this final situation the first priority is to stay on the right of the screen, so the check is to the right. The second priority is to get to the bottom of the screen so the second check is downwards. The final priority is to confirm when the bottom of the screen is reached by checking to the left and down.

Table 2: The order in which directions are checked when searching for different corners of the display screen.

When the locations of each of the corners are found they are stored and a rectangle representing the area that the screen occupies in the camera image is generated from the resulting points, and then the algorithm is complete.

The algorithm that is used to find the screen is subject to a degree of uncertainty about how accurate the final result it produces any given time will be. It could potentially be very problematic if a user was attempting to use the screen and had not realised that VMX's idea of where the screen was located was incorrect. To help prevent this situation feedback about where the algorithm believes the screen is located is given to the user. This feedback is in the form of four markers that are drawn on to the texture of the Kinect's video feed that show the points on the image where VMX thinks the corners of the screen are located. Figure 35 shows these markers. They can be seen at the corners of the TV screen as white dots with black borders. Note that these markers are very small, so it is necessary to look closely at Figure 35 to see them.

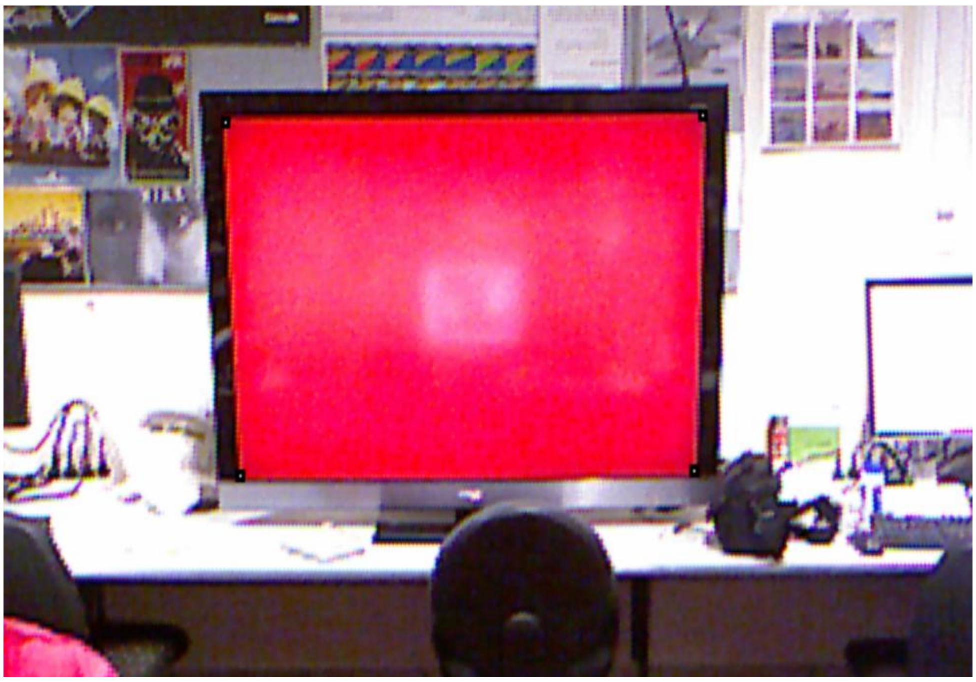


Figure 35: Screen Detection Markers

The method of finding the location of the real world screen that has been given above comes with one major disadvantage over the other methods that were discussed earlier. That is that because the data is found using the colour image feed from the Kinect, the coordinates for each of the corners of the screen are given in colour image space coordinates. The problem with this is that colour image space coordinates are two dimensional and therefore there is no straight forward way to find the depth of each corner of the real world screen. Fortunately, none of the other algorithms used in VMX are dependent on highly accurate information about the distance of the screen from the Kinect device, so a user's estimate of the distance will be sufficient and can be provided in the configuration file for VMX. Another reason that this is not a major problem is that if the real world screen is positioned correctly then there will little difference between the depths of each of the different corners, therefore the user need only estimate a single depth value, not four different ones.

5.6.3 Laser Pointer

After implementing the virtual screen a problem became apparent. If a user pointed at something on their real world screen, then their avatar would need to point to the same location on the virtual screen. In order for this to happen, the relative size and position of the virtual screen to the avatar, would need to match the relative size and position of the real world screen to the user's body. In practice this caused the virtual screen to appear very small, which made it difficult for all participants to see the details on the screen. Figure 36 shows this.

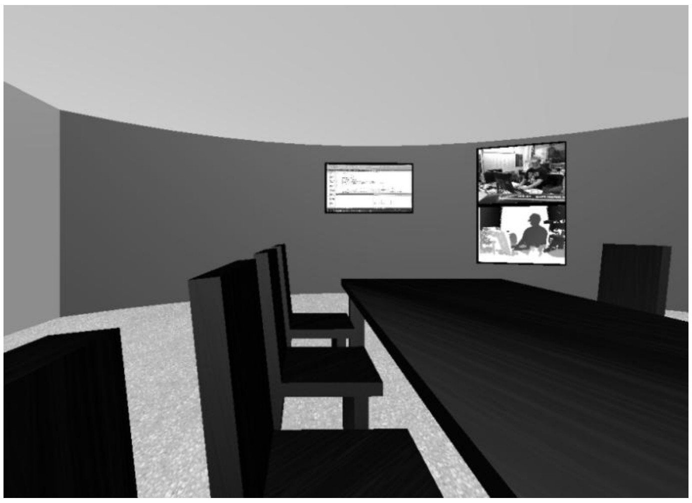


Figure 36: Small Display Screen

The idea of scaling up the size of both the screen and the presenter's avatar was first considered as a solution to this problem, but before that was implemented alternative was thought of. The idea was to use the finger gesture that gives the location in the real world that the user is pointing to data (discussed earlier in Section 5.5.2), along with the real world screen's position data (described in Section 5.6.2) to determine the coordinates (measured in colour image pixels) of where on the real world screen the user is pointing. These coordinates can be converted to represent the equivalent position on the virtual screen as shown in Equation 11.

$$x_{virtual} = \frac{x_{real} - left_{real}}{width_{real}} \times width_{virtual} + left_{virtual}$$

$$y_{virtual} = \left(1 - \frac{y_{real} - top_{real}}{height_{real}}\right) \times height_{virtual} + top_{virtual}$$

Equation 11: Conversion of real world screen coordinates to virtual screen coordinates.

In these 'virtual' values represent positions on the virtual screen, and are given in VMX 3D graphics space; and 'real' values represent positions on the real world screen, and are given in Kinect colour image space. The x and y values are the x and y coordinates of where the user is pointing respectively. The top values represent the y coordinates of the top of their respective screen. The left values give the x coordinates of the left side of their respective screen. The width and height values represent the width and height of their respective screens. The y coordinate needs to be inverted in the conversion, as in Kinect colour image space the y axis increases in a downward direction, whereas in VMX 3D graphics space the y axis increases in an upwards direction.

Once these coordinates have been acquired, a virtual 'laser pointer' is drawn from the presenter's avatar's hand to those coordinates on the virtual screen. The laser pointer is made up of two pieces of geometry: a sphere and a cylinder. The sphere is positioned at the coordinates on the virtual screen. The cylinder is positioned using the same algorithm that positions the cylinders that make up an avatar's body, with the presenter's avatar's right hand joint and the coordinates on the virtual screen serving as the "joints" to connect in the algorithm. Both pieces of geometry are coloured red and do not use lighting calculations to give them a more laser-like appearance.

To prevent situations where a laser pointer appears when the presenter doesn't actually want it (e.g. when the user is just using ordinary body language while speaking and happens to put their hand in front of the screen) the laser is only shown when the user's hand is within a fixed distance from the screen. This requires that the system know the depth of the screen. Because this cannot be determined at runtime with the current system for finding the location of the screen, it must by preset in VMX's configuration file.

The final result makes it possible for an avatar to point at the same location on the virtual screen that the user is pointing at on the real world screen, regardless of the size of the virtual screen. Figure 37 shows the laser pointer in action.

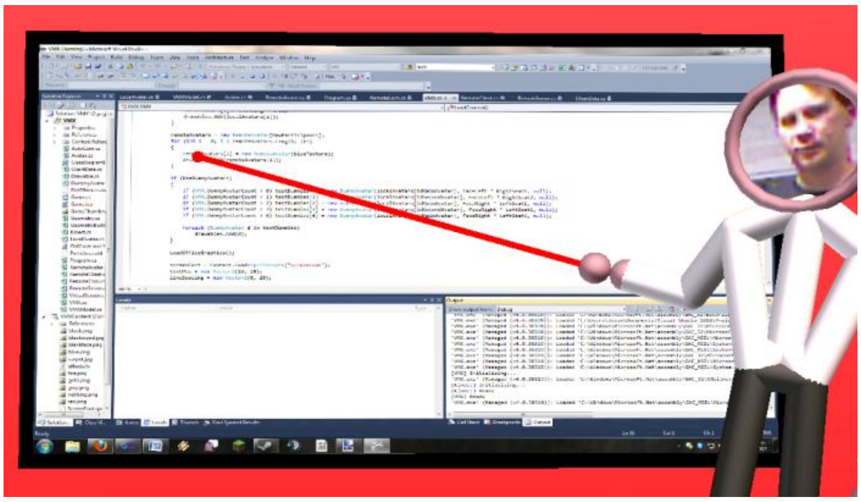


Figure 37: Laser pointer.

5.6.4 Interactive Whiteboard

The algorithm that is used to determine the location on the virtual screen at which to direct the laser pointer can also be adapted to a second purpose. It can be used to allow a user to "draw" on the virtual display screen.

This works by first creating a window on the display screen that will function as a drawing surface. Equation 11 is then modified so that instead of producing coordinates in 3D graphics space, it produces coordinates of the point on a window that corresponds to the place on the screen at which the user is pointing. The modified equation is shown in Equation 12.

$$x_{window} = \frac{x_{real} - left_{real}}{width_{real}} \times width_{window}$$

$$y_{window} = \frac{y_{real} - top_{real}}{height_{real}} \times height_{window}$$

Equation 12: Acquiring the texture coordinates of the point on the display screen a user is pointing at.

The values that previously referred to the virtual screen now refer to the drawing window. The top and left values for the window are not added as an offset at the end of the equations, because these values would always be zero. The other difference is that the equation for the y value does not need to be inverted, as the y axis on the window increases downwards.

Once the coordinates on the window are found, a small solid circle is drawn on the window at this point. The circle will remain there until the user clears the screen. By moving their finger across the screen they are able to draw lines and simple pictures out of the circles.

Drawing is activated by a keyboard command given by the user in control of the presentation screen. The command would open the drawing window on the presentation screen.

Ultimately this feature was not very successful. While it was partially functional, informal testing revealed the jitter in the Kinect skeleton position data had too great an impact to allow the user to draw with any degree of accuracy. For this reason the feature was not included in the formal usability trial. It is an option for future work to try and find ways of solving the accuracy problem.

5.7 Camera Controls

VMX contains two forms of camera control. One requires the user to use their keyboard and mouse to navigate the camera around the virtual environment, much like a video game. When a user is using this form of control, they are said to be using the manual camera. The second form of camera control makes use of the user’s skeleton data from the Kinect to decide how to position and orient the camera. When using this method of camera control the user is said to be using the automatic camera.

5.7.1 The Manual Camera

The manual camera allows the user to manually position their viewpoint in the virtual world. The user does this by using the keyboard to change the position of the camera, and the mouse to change the orientation of the camera.

When the user uses the mouse and keyboard to control the camera, they are essentially changing the values of a few key variables that are used by VMX when generating the view matrix used when drawing objects. There are three of these variables, two of which directly influence each other. The first of these variables is the camera position. The camera position gives the world space coordinates of the camera in the virtual environment. The two closely related variables are the camera target variable and the camera heading variable. The camera target is the location in world space of the point at which the camera is currently directed. The camera heading is the vector that gives the direction from the camera position to the camera target; thus whenever the camera target is changed the camera heading must be recalculated, and vice versa. All of these variables are three vectors. A fourth variable is not modified by the user but plays an important part in the movement process. It is the camera speed variable, and predictably it controls the speed at which the camera moves when the user uses the keyboard to change its position.

The keyboard controls are six keys that control the value of the camera position vector. The keys are arranged into three pairs. Each pair modifies the position vector in a different way. One key in each pair will move the camera in one direction and the other key will move the camera in the opposite direction.

One of the three pairs directly corresponds to, and modifies only one component of the position vector. This pair changes only the Y component of the position vector. The result is to move the camera up and down. The modification to the vector is done by first taking a unit Y vector, then scaling that vector by the value in the camera speed variable. The resulting vector is then either added or subtracted from the camera's position vector depending on which key is pressed.

The other key pairs have a more complicated relationship with the camera position vector. The first of these pairs moves the camera forwards and backwards. This movement does not occur along any particular axis, rather it depends on the camera heading variable. When the user presses one of the forwards or backwards movement keys the camera heading variable is taken, normalised and then scaled according to the camera speed variable. The final result is either added or subtracted from the camera position vector. This has the effect of either moving the camera either forward or backwards in the direction the camera is facing.

The final key pair is responsible for shifting the camera's position left or right. Again, with this pair the left or right movement is relative to the direction the camera is currently facing, not along a particular axis. An extra step is required to acquire the vector that must be added or subtracted to the camera position vector. This vector should be perpendicular to the unit Y vector (so that it will only cause motion on the horizontal plane) and perpendicular to the camera heading (so that it will cause the camera to move sideways). This vector can be acquired by taking the vector cross product of the camera heading and the unit Y vector, this is because the vector cross product of two three dimensional vector is the vector that is perpendicular to the original two vectors(Nykamp). Once this vector is acquired it can be normalised and then scaled by the camera speed variable, to give a vector that will move the camera to the right of where it is currently looking when it is added to the camera's position vector. By subtracting this vector from the position vector the camera can be moved to the left.

In VMX, as with many video games, the user controls the direction of the camera looks in with the mouse. Moving the mouse up and down will pitch the user's view of the environment up and down, and moving the mouse left and right will pan the view left and right. The angle by which the view changes on a given program update depends on how far the mouse has moved since the last update. Normally this would require keeping track of the last mouse position that was recorded, so that that value could be compared to the current position of the mouse. However, this is not necessary as in VMX after reading the position of the mouse, that position is reset to a default location (specifically the middle of the window that VMX is running in. This means that the amount the mouse has moved between updates can be acquired by subtracting that default location from the current location. The location of the mouse is expressed as a two coordinates, one gives the distance of the mouse cursor from the top of the VMX window, and one gives the distance from the left hand border of the window. Both of these coordinates are measured in pixels. The way that VMX acquires the information it needs from the mouse is best illustrated by examining the code directly:

```
MouseState currentMouseState = Mouse.GetState();
float xDifference = currentMouseState.X - graphics.GraphicsDevice.Viewport.Width
              / 2;
float yDifference = graphics.GraphicsDevice.Viewport.Height / 2 -
              currentMouseState.Y;
cameraHoriRot -= cameraSpinSpeed * xDifference;
cameraVertRot -= cameraSpinSpeed * yDifference;
Mouse.SetPosition(graphics.GraphicsDevice.Viewport.Width / 2,
              graphics.GraphicsDevice.Viewport.Height / 2);
```

The first line of this code retrieves the complete set of information about the mouse (as provided by XNA). The second line get the distance that the mouse has moved horizontally since the last time the mouse was checked by subtracting the default horizontal coordinate of the mouse from the actual horizontal coordinate of the mouse. The third line does the same thing for the vertical coordinate, but differs in that this time the actual coordinate is subtracted from the default; this has the effect of reversing the direction of the change it produces in the camera's orientation. In the next two lines we see the two variables that ultimately control the direction the camera points in. They are the cameraHoriRot variable, which gives the angle in radians that the camera should be rotated around the vertical axis; and the cameraVertRot variable which gives the angle in radians that the camera should be pitched away from the horizontal plane. The final line does the job of resetting the mouse cursor's position back to its default location. Note that VMX runs with the mouse cursor hidden, so the user isn't bothered by a flickering mouse cursor in the middle of the VMX window.

The manual camera is a minor example of the use of the fact that the meeting takes place in a virtual environment to do things that one could not do in the real world. In this case the ability is to see the meeting from angles that would not be possible for someone seated at a meeting table in the real world.

5.7.2 The Automatic Camera

The automatic camera relies on using data from the Kinect to position and orient the camera in the virtual environment. For this reason it can only be used after the Kinect device has been initialised and when at least one user is being fully tracked. The camera changes position according to the activity the user is performing (sitting at the table, performing a presentation). When sitting at the table the camera will turn to look in different directions depending on how the user moves.

Originally it was intended that the camera would turn around to match the user's own head movements (e.g. if the user turned their head to the left, the camera would pan to the left). Ultimately however there were two problems with this approach. The first arose from the fact that while it might seem natural to turn one's head left in order to look left, the whole activity is defeated by the fact that a user's computer screen likely only takes up a small area of their vision in front of them. This means that if a user wanted to look to the left, in order to still see their screen they would have to direct their eyes to the right. This is a somewhat unnatural position to sit it, and it could become uncomfortable if a user was required to do it for a long time (which could be the case if the user wanted to look at someone on their left in the virtual environment while that person gave a presentation). The second problem with this approach is that the Kinect runtime as it is provided by Microsoft provides no data about the rotation of a user's head in space. This means that for this kind of camera control to be possible, an algorithm that could infer the rotation of a user's head from the raw depth or image data would have had to have been created. This would have been a very time consuming process for a feature that would likely not be very ergonomic. Consequently a different way to do this was sought.

The solution was to make use of the position of the user’s shoulders instead of the position of the user’s head. To do this the idea was to analyse the position of the user’s Kinect skeleton’s left and right shoulder joints with respect to each other. The distance in front of the Kinect device of each shoulder would be taken and compared with each other. When the one shoulder was closer to the device than the other shoulder, the automatic camera would pan to the left or the right. This meant that in order to turn the camera in the virtual environment the user would need to effectively rotate their body, not their head. This allowed the user to keep their head looking directly at the screen at all times, a much more comfortable position as it is common for a person to have their head facing in a slightly different direction from their body when they are looking at something. Note that body rotation is easy to accomplish in a rotatable office chair.

A second aspect of the user’s body posture was also experimentally used for control of the automatic camera. The purpose of this second control was to allow the user to zoom the view of the automatic camera. It works in a similar way to the mechanism for turning the camera, but instead of using the left and right shoulder joints, the algorithm uses the head and spine joints. In this case too, it is the relative distances of these joints from the Kinect device that is considered by the algorithm. The net effect of the algorithm is that the zoom of the automatic camera will change when the user leans forward or backwards. There is a slight problem with this approach however. The problem stems from the Kinect’s limits on how far a user must be from the device in order for skeleton tracking to function correctly. The minimum distance a user may be from the Kinect is limited to 82 centimetres. As it stands a Kinect device positioned above a user’s computer screen is most likely already very close to the user. This means that if a user leans forward there is a strong possibility that the Kinect runtime will lose the capability to directly track the user’s head joint. Then the runtime will attempt to infer the location of the head instead. This can have unexpected results and cause the camera to behave in ways that could be confusing the user. A solution to the problem is of course to move the Kinect device further away from the user. This comes with trade-offs however, such making the user appear smaller in the Kinect’s camera image, lowering the resolution of the image of their face that is sent to other user’s. Ultimately because the problem does not completely prevent the zooming system from working, the feature was left in as is.

There is one other mechanism that controls the automatic camera. Its purpose is to switch the mode of the automatic camera between one designed for a user sitting at the table, and one for doing a presentation. When a user’s avatar is sitting at the table, the camera sits in the same position as their avatar and will pan and zoom according to the user’s movements as described above. However when doing a presentation the automatic camera changes its operation. There are two main effects of this change. The first is that the camera’s position changes so that it has a view straight down the middle of the virtual meeting room. This allows the presenter to see their audience clearly. The second effect is to disable the panning and zooming functions that are described above. The reason for doing this is that when making a presentation, a user is likely to be quite animated (walking around, performing gestures, pointing at the screen etc.). Left unchecked this would cause the automatic camera to flail about wildly and unhelpfully. For that reason when in presenting mode, the automatic camera does not pan or zoom by itself. To decide whether the automatic camera should be in sitting mode, or presenting mode, the control algorithm looks at how far away the user is from their Kinect device. If they are deemed close enough to the device, the camera will go into sitting mode, if they are far enough away then the camera will go into presenting mode. The user’s distance from the Kinect device is taken from the Z axis coordinate of their Kinect skeleton’s centre shoulder joint.

Early on there was a problem with this mode changing system. Originally there was simple a distance threshold that would cause the switch from presenting mode to sitting mode and back again. This was problematic because if the user was sitting in a position where they were close to this threshold, then there would be a tendency to constant switching between modes as the user moved about (particularly when they were leaning forwards and backwards to zoom the camera). The solution to this problem was to simply modify the system so that there was not a single threshold, but two. One threshold would be closer to the Kinect device and trigger the change into sitting mode, and one would be further away and trigger the change into presenting mode. This means that once a user enters a particular mode, they have to make a significant change in position to switch back out of it. In VMX there is no built in way for automatically determining appropriate thresholds, but they can be set in VMX's configuration file. By default the thresholds are set to be 50 centimetres apart.

There is one final thing that must be done in order to make the automatic camera work well. Even though VMX instructs the Kinect runtime to use smoothing on the user's skeleton data, there is still a degree of jitter in the positions of joints over a sequence of skeleton frames. Add to that the fact that the Kinect is reasonably sensitive to small movements that the user actually makes, and the result can be a very shaky automatic camera. To compensate for this, smoothing is done in VMX whenever certain properties of the camera are changed. These properties are: the camera's position vector, the camera's horizontal angle of rotation (yaw), the camera's vertical angle of rotation (pitch), and the camera's field of view (effectively the camera's zoom). To achieve smoothing all four of these camera properties need two separate values each. One of the values is the target value for that property. The target value is the raw data value that would be used if no smoothing was being done. The other value is the actual value that that has been smoothed. It is this value that is used when calculating the view/projection matrices for the camera.

The smoothing algorithm is quite simple. Whenever a discrepancy is detected between the target value and the actual value for one of the properties, the actual value is recalculated as follows:

$$Actual = OldActual + \left(\frac{Target - OldActual}{10}\right)$$

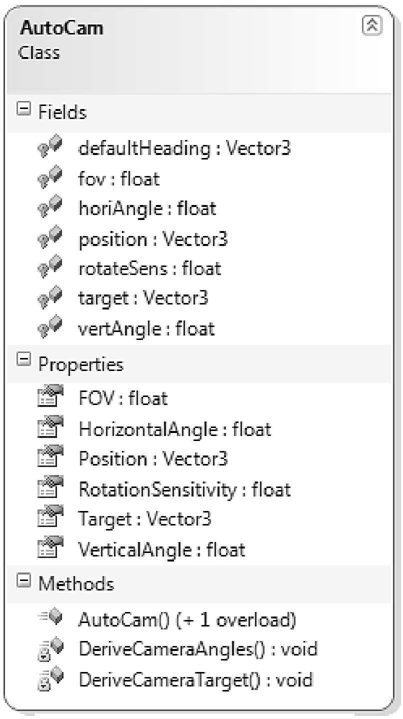
Equation 13: Smoothing camera movement.

This equation results in smooth transitions when changes are made to camera position, with faster movement when the discrepancy between the target and actual values is large.

5.7.3 The AutoCam Class

The AutoCam class is used by VMX for storing data about different automatic camera modes. The two modes that are discussed above (presentation mode, and sitting mode) are the only modes that are used in normal operation of VMX; though there is also a third mode that is used when the automatic camera is enabled but no users are being tracked by Kinect.

In each of these modes the camera has different default positions, orientations and fields of view. For example, the sitting mode camera is positioned at the side of the table, oriented to face the table, and has a somewhat narrow field of view that makes it easy to look at the faces of specific individuals around the table. The presentation camera on the other hand is positioned at one end of the room, is oriented to look across the table straight down the middle of the room, and uses a wide field of view so that everyone in meeting can be seen at once. The AutoCam class provides a place to store these pre-set values.



```
AutoCam
Class

Fields
  defaultHeading : Vector3
  fov : float
  horiAngle : float
  position : Vector3
  rotateSens : float
  target : Vector3
  vertAngle : float
Properties
  FOV : float
  HorizontalAngle : float
  Position : Vector3
  RotationSensitivity : float
  Target : Vector3
  VerticalAngle : float
Methods
  AutoCam() (+ 1 overload)
  DeriveCameraAngles() : void
  DeriveCameraTarget() : void
```

Figure 38: The AutoCam class

The AutoCam class has six public properties that are used to store data. These are called: Position, VerticalAngle, HorizontalAngle, FOV, RotationSensitivity, and Target. The Position property simply gives the world space translation for the camera’s default position. The FOV value gives the default field of view angle (in radians) to use when producing the projection matrix for this camera. The VerticalAngle, HorizontalAngle and Target values are all closely related and affect each other. Essentially between them, there are two different ways to determine which way to orient the camera. The Target property gives the world space translation of a point in the virtual environment that the camera is set to look at. The HorizontalAngle and VerticalAngle give the yaw (left/right rotation) and pitch (up/down rotation) angles of the camera respectively.

Either the Target property or the two angle properties together can be independently used to set the final orientation of the camera. It is not necessary to use all three properties. The target property is best used when there is a specific object in the environment that the camera needs to be pointed at, as the Target value can simply be set to the world translation coordinates of that object. The angle properties are best used when there is a specific direction that the camera needs to be pointed in. The values can be set to the bearing corresponding to that direction. Of course, whenever the angles are changed, the target value must be updated to reflect the new direction; likewise when the target value is changed, the angles must be updated. To do this the AutoCam class has two private methods. The DeriveCameraAngles method is called whenever the Target property is set, and the DeriveCameraTarget method is called whenever one of the angle properties is set.

Both the DeriveCameraTarget and DeriveCameraAngles utilise an additional static value stored in the AutoCam class called defaultHeading. This value is a vector that represents the direction in which a camera will face when the HorizontalAngle and VerticalAngle properties are both zero. The value is always a positive unit Z vector i.e. Vector (0, 0, 1). This base value is important for converting between the Target property and the angle properties.

The DeriveCameraTarget method is the simpler of the two methods. To start, the two angle properties are used as parameters to create two rotation matrices. The horizontal angle generates a matrix that rotates around the Y axis, and the vertical angle generates a matrix that rotates around the X axis. The X axis rotation matrix is then multiplied by the Y axis rotation matrix to give a final rotation matrix for the camera. Applying the final rotation matrix to the defaultHeading vector gives the actual direction vector for which way the camera should now be facing. This vector can be transformed into a vector that represents the target vector by adding it to the Position property of the camera.

The DeriveCameraAngles method is slightly more complex than the DeriveCameraTarget method, because there are two separate values that need to be found. The first step of this method is the reverse of the last step of the other method. The position vector of the camera is subtracted from the new target vector giving the vector that represents the direction the camera is facing in.

With the direction vector acquired work begins on finding the value for HorizontalAngle, i.e. the angle between the X and Z components camera's direction vector and X and Z components of the defaultHeading vector. This is shown as calculated in Equation 14.

$$Angle = cos^{-1} \frac{defaultXZ \cdot camDirectionXZ}{|defaultXZ|\,|camDirectionXZ|}$$

Equation 14: Finding the angle between a pair of two dimensional vectors

In fact Equation 14 only provides the magnitude of the horizontal rotation angle, so an additional step must be taken to determine whether this rotation should be to the left, or the right. If the X component of the camera's direction vector is positive then the rotation will be to the left so the value for the horizontal angle is left unchanged. If the X component is negative however, then the angle must be multiplied by -1 before being stored.

The process for finding the value for VerticalAngle is similar. The main difference is in the two vectors that need to be used in the angle finding equation. Unlike when finding the HorizontalAngle, the VerticalAngle is found using three vectors. The vectors used are the camera direction vector that was acquired earlier, and a duplicate of that vector that has had its Y component set to zero. These two vectors are then substituted into the equation above (Equation 14) to give the magnitude of the angle. To set the sign, this time it is the Y component that it is checked. If it is positive, then the value is left unchanged and the rotation will be upwards. If it is negative, then the value is multiplied by -1 and the direction of rotation will be downwards.

5.8 Network Communication

In order to facilitate meetings across multiple computers in locations, the program needs the ability to send Kinect data over the Internet. To do this VMX utilises a client-server model, supporting up to seven remote client connections to the server. VMX does not have a dedicated server program; any instance of the VMX program can function as either client or server.

5.8.1 Network Structure

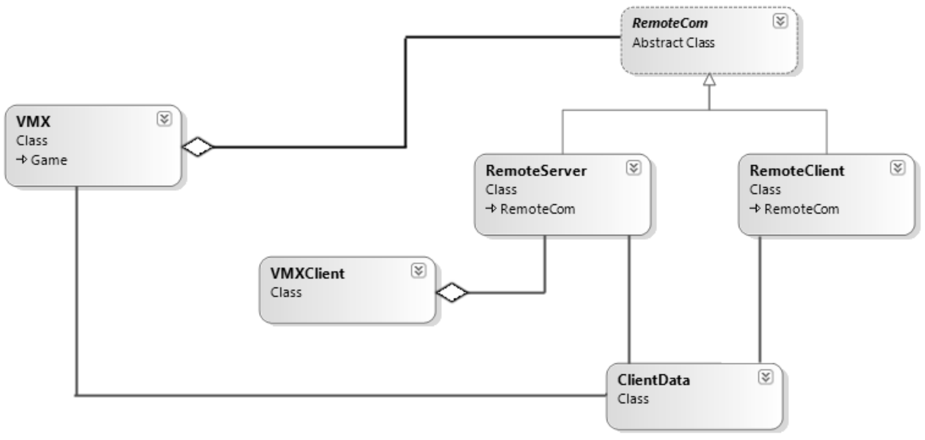


Figure 39: Network Class Structure

As can be seen in Figure 39, the networking system is made up of the VMX core class and five classes specific to the system. The diagram shows the relationship between each of these classes.

RemoteServer is the class that hold the server implementation for VMX. When an instance of VMX needs to host a meeting and function as a server, this class will be used. It contain all of the code necessary for establishing a server, listening for connections from clients, and relaying data between itself and all of the clients connected to it.

RemoteClient is the class that is used for network communication when an instance of VMX is not functioning as the host of a meeting. The RemoteClient class contains only the code necessary to open a connection to a server, and communicate with that server. The server sends all of the information about itself and other clients connected to it to the client, so only the connection to the server is required.

The RemoteCom class is an abstract class from which both RemoteClient and RemoteServer inherit. All of the public members of both RemoteServer and RemoteClient are declared within RemoteCom. As can be seen on the diagram there is no direct relationship between the VMX core class and RemoteServer or RemoteClient. The core class only interacts with RemoteCom. The purpose of this is to allow the VMX core class to interact with its networking system in the same way regardless of whether it is running as a server or a client. This simplifies the code in VMX and helps ensure that all code specific to running a server is kept in the RemoteServer class, and all code specific to running as a client is kept in the RemoteClient class. Importantly, this includes the methods for encoding and decoding packets.

Because the server and client each send packets that contain different information, the RemoteServer and RemoteClient classes each have different ways of encoding the packets they send, and decoding the packets they receive. In order to keep client and server specific code out of the VMX core class there needs to be an intermediate way of storing the information that needs to be sent across the network, or has been received across the network. If the client and server were to pass back information to VMX simply by handing over the packets they received then there would need to be server and client specific code in the core to handle the different packet types. The goal of avoiding this necessity gave rise to the ClientData class.

As can be seen on the diagram above the ClientData class is used by the VMX core, RemoteServer and RemoteClient. It is used as a place to store data when it is being passed between the VMX core, and the networking system. Its existence permits the desired situation of having the core be oblivious to whether it is functioning as a server or a client.

The final class on this diagram is the VMXClient class. This class is used only by the server and is used as a way to consolidate several bits of information relevant to a single client connected to the server. The server maintains one instance of this class for every client currently connected to it.

5.8.2 Packet Structure

VMX essentially utilises two types of packets. One is used by the server to send data to clients; the other is used by clients to send data to the server.

The reason for having different types of packets for sending data from a server and sending data from a client lies in the fact that a client will always be sending one set of data (its own) and a server will often be sending multiple sets of data (its own plus data from other connected clients).

Figure 40 shows the structure of the packet that the server sends to the clients connected to it. The first four bytes of the packet are the payload length, which describes the length of the entire packet, excluding itself. These four bytes are read and processed by the receiving client before other information is read from the TCP socket. It is used to determine how much data must be read from the socket in order to assemble the entire packet. The next byte in the packet is the Client ID, this tells the client what its own currently assigned ID number is. The next byte is the Presenter ID, this tells the client the ID number of the client that is currently doing a presentation. This information is used to decide which client avatar should be placed in front of the virtual presentation screen (if any). This value can be the same as the Client ID (if the client receiving the packet is the presenting client). The value can also be the ID of the server (if the user hosting the server is doing a presentation), or if all of the bits of this byte are set, then no one is presenting. The next byte is a block of eight flags. These flags indicate two things: the first is how many sets of client data are included in the packet, and the second is the ClientIDs that correspond to each set of data. If a given flag is set in this block, it means that the packet contains data for the client with the Client ID associated with that flag. Blocks of clients' data are then placed in the packet in the same order as their associated flags. Immediately preceding each block of client data there is a four byte value that gives the length (in bytes) of that. This is put in so that when a client program is decoding the packet, it can tell where one block of client data ends, and the next begins.

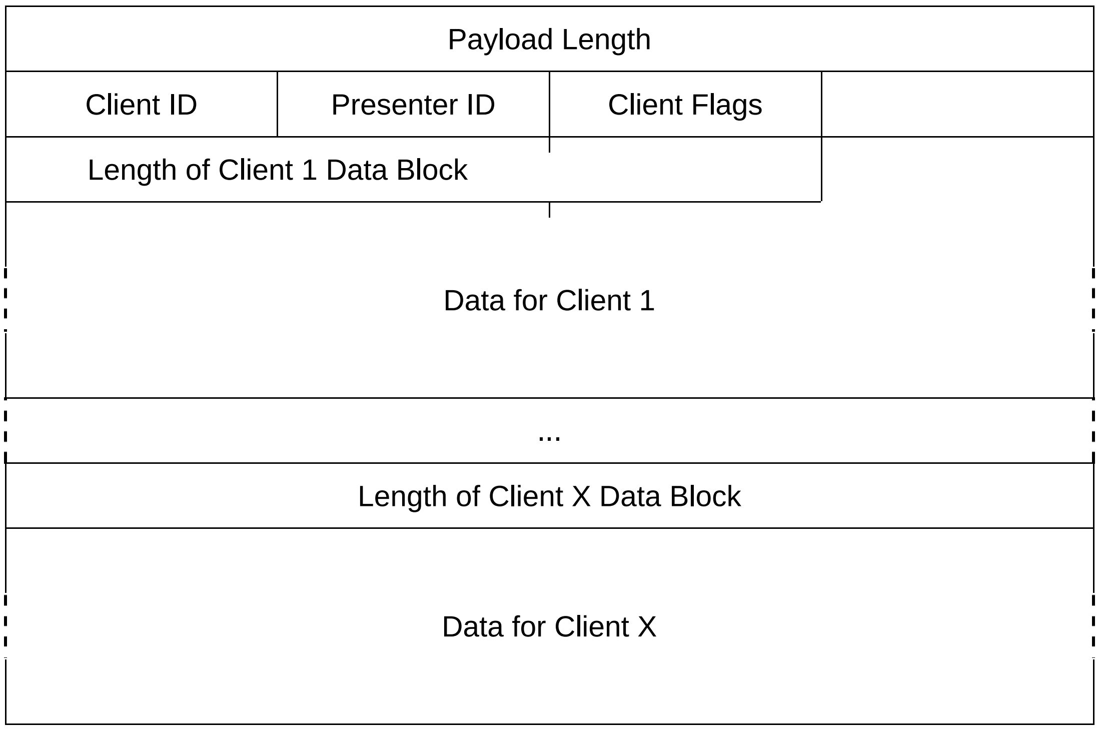


Figure 40: Server Packet Structure

Figure 41 shows the structure of the packets that are sent from a client to the server. These packets are much simpler than the packets sent by the server. They contain a four byte Payload Length which serves exactly the same purpose as the Payload Length value in the server packet. The remainder of the packet is made up of one set of Client Data containing the data for the client that is sending the packet.

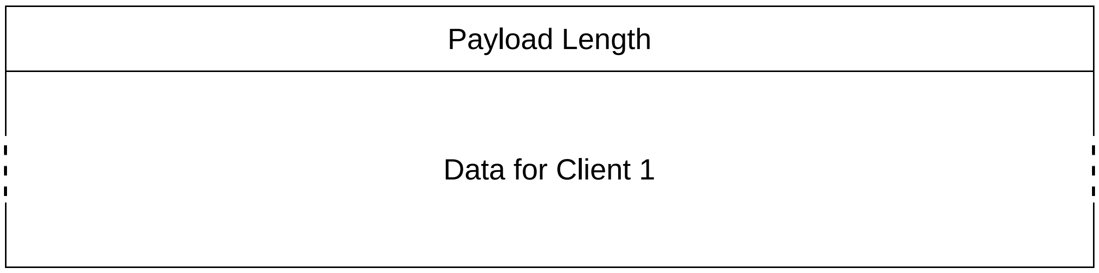


Figure 41: Client Packet Structure

Both the server and the client use the same structure for packaging the data for each set of client data. Figure 42 shows this structure.

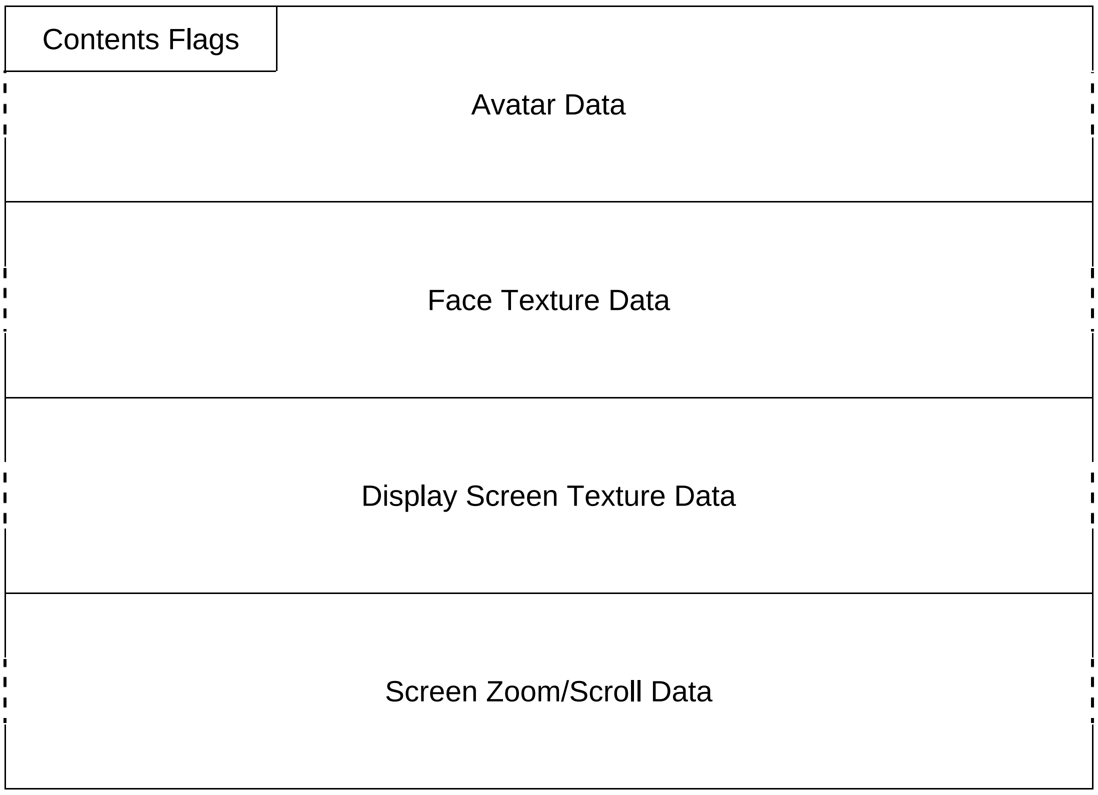


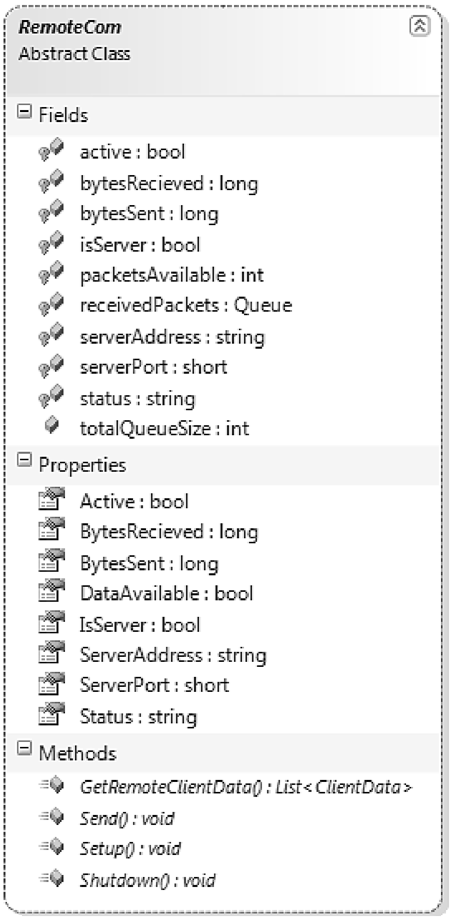
Figure 42: The packet structure of the data for one client.

Most of the blocks of data within this structure are optional. Only the Contents Flags will always be present. These flags indicate which of the other blocks of data are actually contained within a particular packet. The details of the data is stored in each of these blocks can be seen in Table 3 in Section 5.8.4 .

5.8.3 RemoteCom

Whether or not a given instance of VMX is functioning as a client or a server is almost completely transparent to the core program. The core program interacts with an abstract super class named ‘RemoteCom’; this class is inherited by ‘RemoteServer’ and ‘RemoteClient’ which actually implement the methods provided by RemoteCom. Both server and client each use their own packet structures.

Remote communications are initiated either by user command or from an instruction in the configuration file. The VMX core class maintains a variable called remote; this variable is of type RemoteCom. When VMX is instructed to either start a server or a client module, the appropriate class is instantiated and stored in remote.


RemoteCom
Abstract Class

Fields
active : bool
bytesRecieved : long
bytesSent : long
isServer : bool
packetsAvailable : int
receivedPackets : Queue
serverAddress : string
serverPort : short
status : string
totalQueueSize : int

Properties
Active : bool
BytesRecieved : long
BytesSent : long
DataAvailable : bool
IsServer : bool
ServerAddress : string
ServerPort : short
Status : string

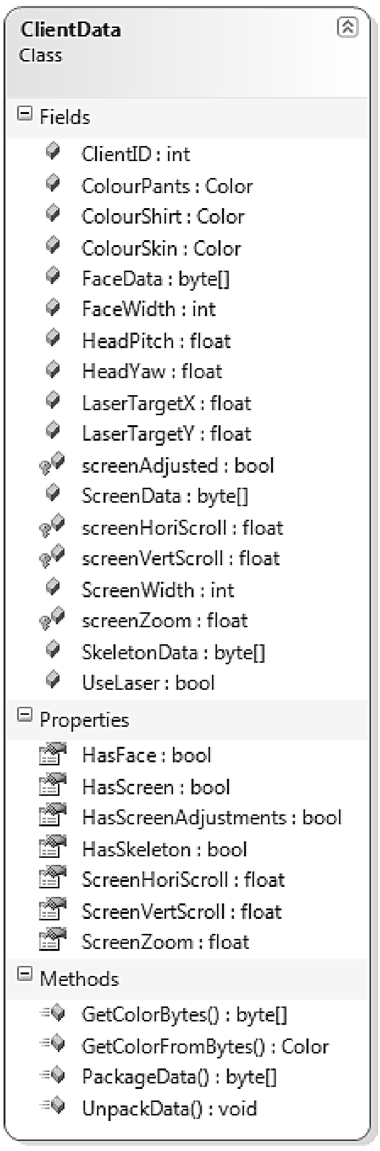
Methods
GetRemoteClientData() : List<ClientData>
Send() : void
Setup() : void
Shutdown() : void

Figure 43: RemoteCom class structure

RemoteCom is abstract and contains declarations of all of the methods that the core VMX class uses when performing remote communications. These include methods for sending and receiving data, and methods for setting up and shutting down the remote connection. In addition to these methods, the RemoteCom class provides a host of properties for providing state information to the VMX core. Among these there are properties which specify: whether this instance of RemoteCom is a server or a client, the total amount of data that has been sent, how much has been received, whether the remote system is currently active and set up, a string which gives any status messages back to VMX, and whether any packets have been received and are ready to be processed. In addition to these read-only properties, there are two settable properties. One is called ServerPort and one is called ServerAddress. The precise function of these two properties depends on whether the system using them is a client or a server. For a client, the properties give the address and port to connect to in order to communicate with a given server. For a server the ServerAddress property is unused, and the ServerPort gives the port on which the server should listen for new connections.

5.8.4 ClientData

As described, the client and server systems each use their own packet formats, requiring slightly different ways to decode each kind. This is problematic when trying to keep their inputs and outputs identical when interacting with the VMX core class. To solve this, an intermediate data structure called ClientData is used to store all the information about a single client when it is being passed between the VMX core and its server or client class. When the VMX core is retrieving data from or supplying data to an instance of ClientData, it will use the class's assortment of public variables and properties; however when the server or clients classes interact with the ClientData class they use a pair of methods. One of the methods (PackageData) is used to encode all of the data in the class into a byte array that can be put directly into a network packet, the other (UnpackData) carries out the opposite function, decoding a byte array taken from a packet to populate the variables in an instance of ClientData.



ClientData
Class

Fields
- ClientID : int
- ColourPants : Color
- ColourShirt : Color
- ColourSkin : Color
- FaceData : byte[]
- FaceWidth : int
- HeadPitch : float
- HeadYaw : float
- LaserTargetX : float
- LaserTargetY : float
- screenAdjusted : bool
- ScreenData : byte[]
- screenHoriScroll : float
- screenVertScroll : float
- ScreenWidth : int
- screenZoom : float
- SkeletonData : byte[]
- UseLaser : bool

Properties
- HasFace : bool
- HasScreen : bool
- HasScreenAdjustments : bool
- HasSkeleton : bool
- ScreenHoriScroll : float
- ScreenVertScroll : float
- ScreenZoom : float

Methods
- GetColorBytes() : byte[]
- GetColorFromBytes() : Color
- PackageData() : byte[]
- UnpackData() : void

Figure 44: ClientData class structure

The ClientData has variables and properties that correspond to all of the kinds of data that an instance of VMX will need to send across a network, Table 3 lists all of those variables. It should be noted that not all kinds of data are transmitted all the time. For example, VMX will only transmit skeleton position information when it has new skeleton position information to send; if there is no skeleton data available, or the current skeleton data has already been transmitted, then no skeleton data will be provided to the ClientData class. Table 3 also shows the conditions under which particular data is included.

Will be included when…	Variable Name	Variable Type	Variable Contents
Included when new skeleton data that hasn't already been transmitted is received from the Kinect runtime.	ColourPants	Color	The colour to use when rendering the legs of an avatar.
	ColourSkin	Color	The colour to use when rendering the hands, feet and head of an avatar.
	ColourShirt	Color	The colour to use when rendering the arms and torso of an avatar.
	HeadYaw	float	The angle to rotate the avatar's head about the Y-axis.
	HeadPitch	float	The angle to tilt the avatar's head up and down.
	UseLaser	boolean	Whether to draw a laser pointer beam from this avatar's hand
	LaserTargetX	float	The X-coordinate of where the laser pointer should point to
	LaserTargetY	float	The Y-coordinate of where the laser pointer should point to
	SkeletonData	byte[]	The position data of all of the joints in the avatar's skeleton.
Included when a new face texture has been generated from the skeleton data and colour image data is received from the Kinect.	FaceData.Length	int	The size of the FaceData array (see below). Note that this is not stored in a separate variable, but is included in the encoded data produced by the PackageData method.
	FaceWidth	int	The horizontal resolution of the avatar's face texture.
	FaceData	byte[]	The texture data for the avatar's face image.
Included when the image on the virtual display screen has been changed or updated.	ScreenData.Length	int	The size of the ScreenData (see below). Note that this is not stored in a separate variable, but is included in the encoded data produced by the PackageData method.

	ScreenWidth	int	The horizontal resolution of the texture on the virtual display screen.
	ScreenData	byte[]	The texture data for the image on the virtual display screen.
Included when the zooming or scrolling functions of the virtual display screen are used and changed.	ScreenZoom	float	The zoom factor to use on the virtual display screen.
	ScreenHoriScroll	float	The horizontal scrolling position to use on the virtual display screen.
	ScreenVertScroll	float	The vertical scrolling position to use on the virtual display screen.

Table 3: Data contained within the ClientData class

In addition to these variables there are four public properties in the ClientData structure. These are called HasSkeleton, HasFace, HasScreen, and HasScreenAdjustments. They correspond to each of the four conditions given in the first column of Table 3 respectively. These properties are all Boolean values, and all are non-settable. They return true if their corresponding condition is satisfied and false if it is not. They are used under two different circumstances. The first is when VMX is reading data out of the ClientData class; the properties are accessed to determine exactly what data to read. The second use is by the ClientData class itself, when it is deciding what data needs to be encoded into a byte array for transmission.

The PackageData method is used by both the client and server classes to encode data for transmission. This method starts by determining what data it needs to include in the byte array and how much space the data will take up (i.e. how big the byte array needs to be). While it is doing this it also generates the inclusion flags that will be placed at the beginning of the byte array. The first stage of this process is to instantiate three variables. The first of these is a Boolean called haveDataToSend, this initially set as false and then only set to true once it is confirmed if there is currently meaningful data stored the ClientData instance. The second is an integer called dataLength that is used to keep a running count of the total amount of data that needs to be sent (in bytes). The third variable is a single byte called flags; which will eventually become the first byte in the final array.

Following this, HasSkeleton, HasFace, HasScreen, and HasScreenAdjustments are each checked in sequence. If any of these values is true then haveDataToSend will be set to true. If HasSkeleton is true then the total size of all of the data that is sent under this condition will be added to dataLength, and the flags byte will be updated by performing a bitwise ‘or’ operation with the skeleton data flag value on the flags byte. A similar process will be followed if HasFace, HasScreen and/or HasScreenAdjustments are true; with the dataLength being increased by different amounts depending on the type of data that is being included. Once this is done, the algorithm will know how many bytes the encoded data will require. If none of the four ‘Has’ conditions are true then at this point the method will return a zero-length byte array.

If at least one of the ‘Has’ conditions was true then the process of encoding it into a byte array will begin. This starts with the creation of a new array with the number of elements given by the value in the dataLength variable. The first data to be added to the byte array is the flags byte from earlier. The method will then go through encoding the data shown in Table 3 into the byte array in the order shown in the table (subject to the inclusion conditions for each type of data in the table). Each variable is handled one at a time, first being converted from its ordinary type (float, int, bool, or Color) into a byte array using the C# system BitConverter class (assuming it isn’t already stored as a byte array) and then that byte array is copied into to the main byte array. Once all of the data has been included, the final byte array is returned by the method.

The UnpackData method carries out the reverse process to the PackageData class. It starts by reading off the first byte in the array it is to decode. This byte contains the flags which indicate what data is included in the remainder of the array. The flags serve the same purpose as the ‘Has’ properties in the PackageData method. The method then goes through the remainder of the byte array looking for each bit of data that needs to be extracted based on which flags were set (following the order shown in Table 3). When extracting a particular piece of data, the bytes needed to encode it are extracted from the array and then parsed back into the normal type for the data (using BitConverter) before being stored into the appropriate variable. Most of the data stored in the packet takes up a known and constant number of bytes in the array. The exceptions to this are the ScreenData and FaceData byte arrays, these have a variable length. This is the reason that ScreenData.Length and FaceData.Length are encoded into the main array separately. They are used to determine how many bytes need to be extracted from the main byte array to extract their respective data arrays.

The ClientData class also has two simple utility methods called GetColorBytes and GetColourFromBytes; these are used to encode a Color variable into a byte array and a byte array into a Color variable respectively. When encoded into a byte array, the red, green and blue values of the Color are each stored in a single byte, meaning that each Color variable encodes into three byte array.

5.8.5 VMXClient

Before talking about the implementation of the server in VMX we must look at simple but important data structure that the server uses. This structure is called VMXClient. The purpose of VMXClient object is to keep track of five objects that all relate to the same remote VMX client on the server.

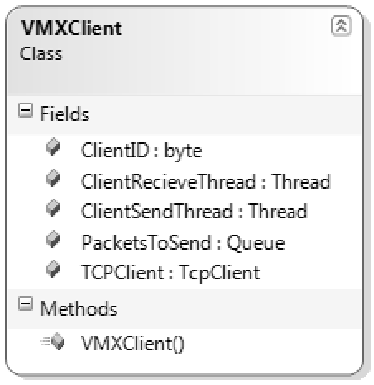
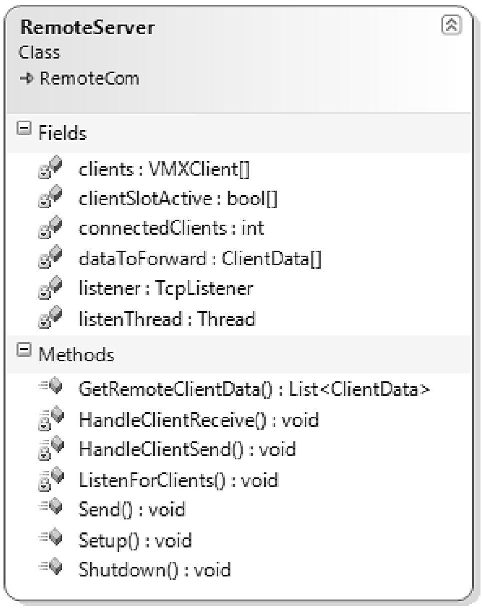


Figure 45: VMXClient class structure

The first of these five objects is an integer called ClientID that is used by the server to keep track of each individual client, particularly when forwarding data from one client to another. The second is a TcpClient object. TcpClient is part of the System.Net.Sockets library and is used to access and manage the TCP network connection between the server and the client. The third object is the thread that is responsible for listening for new communications from the client. The fourth is the thread that is responsible sending packets to the client. The fifth object in the VMXClient class is a Queue object which contains all of the packets that are waiting to be sent by the send thread.

5.8.6 Server

As the name suggests RemoteServer is the class that is used to run and manage a server for VMX. Aside from providing its own implementation of the methods declared in RemoteCom, it also maintains the variables and private methods that are necessary for managing communications between it and multiple clients, having also to serve as a relay for communicating information from clients to other clients. The server class relies heavily on threading to carry out its functions.



RemoteServer
Class
→ RemoteCom

Fields
- clients : VMXClient[]
- clientSlotActive : bool[]
- connectedClients : int
- dataToForward : ClientData[]
- listener : TcpListener
- listenThread : Thread

Methods
- GetRemoteClientData() : List<ClientData>
- HandleClientReceive() : void
- HandleClientSend() : void
- ListenForClients() : void
- Send() : void
- Setup() : void
- Shutdown() : void

Figure 46: RemoteServer class structure.

When it is first instantiated the server class does nothing. Before it can communicate with any clients, its Setup method must be called from VMX's core class. The Setup method is charged with setting several key variables, and initiating the first stages of establishing connections. The first thing done in the Setup method is to initialise and set a series of variables. This includes initialising a new queue that holds packets as they are received, setting the property that states whether this instance of RemoteCom is a server to true, and wiping the array which states whether the a particular client ID is in use. Once that is done, the method moves on to start the process of listening for new client connections. To do this, a TcpListener object is instantiated and given the port that it will need to listen on when activated (provided in the ServerPort property of RemoteCom). The method that accepts new connections as they are made would block the program. For this reason VMX puts it into its own thread (listenThread), which is instantiated by the Setup method. The listenThread's task is to handle the process of accepting new client connections via the TCPListener object. The listenThread is then immediately started. The final act of the Setup method is to set the property that states if a RemoteCom object is fully active to true.

The body of the listenThread is a method is called ListenForClients. The first step of ListenForClients is to instruct the TCP listener to start running. The thread then loops continually until the server is told to shut down. The first step of the loop is to check if the maximum number of connected clients has been reached; if it hasn’t then a new TcpClient object will be created. This TcpClient object is used as a place to store the return value of the TcpListener’s AcceptTcpClient method (this is the method that blocks). This method will only return once a connection has been established to a new TcpClient. Once this happens a new VMXClient is instantiated and passed the returned TcpClient. Then the array of client IDs will be searched for an available ID. Once one is found it will be marked as in use and the ID will be assigned to the VMXClient object. Following that the VMXClient will be added to the list of currently connected clients, and the count of currently connected clients will be incremented.

In order to receive communications from multiple clients, it is necessary to wait for new data from each of those clients continuously. The method for reading from a client socket is also a blocking method. Thus for every client that connects, a new thread must be created to listen for incoming data. The last stage of the loop in the ListenForClients method is to establish the new thread, using the method HandleClientReceive as its body. HandleClientReceive is responsible for receiving incoming data from the client and doing the first stage of processing on it. It is then stored in the appropriate VMXClient object. The thread is then started. Unlike the ListenForClients method the HandleClientReceive method takes a parameter (the VMXClient object), this is passed in with the call to start the thread; it provides access to the queue for storing incoming messages. Finally the ListenForClients method loops back around to listen for the next incoming connection.

The HandleClientReceive method is similar in structure to the ListenForClients method. It starts with a TCPClient (socket) and a queue for storing messages read. Once its initialisation is done, the method enters a loop which will continue until: the server is shutdown; there is a problem with the connection; or the client closes the connection.

The loop starts by immediately entering a try/catch block. The purpose of this block is to catch any exceptions that occur during the communication process so the connection and the thread can be safely shutdown. Upon entering the try/catch block the routine attempts to read the payload size header from the client socket.

The Read method returns a value, which is the number of bytes that were actually read. It will block until some communication is received from the client or the connection is terminated. This is why each client must have its own thread on the server. The next stage of the HandleClientReceive method determines whether the Read method returned because there is new data or because the connection was terminated. It does this by checking the number of bytes read; if it equals zero then it is known that the Read method returned without reading any data; thus the connection must have been terminated. If this is the case then the clientDisconnected variable is set to true and the loop is broken. If payload size header was actually received then the number of bytes that were read (4) will be added to the RemoteCom property for the total number of bytes downloaded. The value for the payload size is parsed out and used to determine how many bytes need to be read from the socket to reconstruct the incoming packet. A new byte array is created to hold the full packet, and then the program enters a loop that repeatedly calls the Read method on the socket until enough data has been received to fill the array. After each Read call the number of bytes read will be added to the total bytes read property of RemoteCom.

Once the loop completes the try/catch block ends. The message array and the ID number of the client from which the packet was received are then placed in a simple data structure called QueuedPacket which is added to a queue of all packets that have been received by all of the threads that read data from clients. The data will remain in that queue until the server is ready to process it.

When the loop ends, the server is no longer communicating with the client and some final tidy up is done. This includes: decrementing the number of connected clients, freeing the associated client ID in the array of client ID’s available, removing the VMXClient object from the list of currently connected clients, and finally closing the TcpClient’s stream. The thread responsible for sending data to the client will stop by itself when the TCPClient connection is closed.

Once a packet is on the queue of received packets, there is still some additional processing to be done. This processing is done when a call to the GetRemoteClientData method is made from the VMX core class (in the program’s main thread). The GetRemoteClientData method is one of the methods declared by RemoteCom and therefore there is also an implementation of it in the client for VMX. GetRemoteClientData takes no parameters but does return a list of ClientData objects. Each object in this list represents one packet that has been processed.

The method starts by instantiating the list of ClientData objects that will be returned at the end. Next it identifies the number of packets that are currently in the queue and need to be processed. This value is stored in a variable called 'packetsToRead'. From there the method iterates over each packet in the queue until the number of packets that have been processed equals the value stored in packetsToRead. The reason for not simply continuing to process packets until the queue is empty, is that while the main program thread is doing this processing, each of the client threads can be adding new packets to the end of the queue, which could result in this method continually processing new packets as they came in, and consequently not returning at a reasonable speed (or not returning at all). It should be noted that whenever the queue is accessed from any thread, it is first locked to prevent concurrent access issues.

The process of iterating over a packet is fairly complex. It begins by first dequeuing a QueuedPacket from the packet queue, and then instantiating a new ClientData object. The next step is to extract the Client ID from the QueuedPacket structure; this value is immediately stored in the Client ID property of the ClientData object. Then the actual information in the byte array is decoded. This is done by the ClientData object itself. The byte array is extracted from the QueuedPacket and then passed into the ClientData object's UnpackData method which decodes the information from the packet and uses it to populate its various properties and variables.

With all of the data from the packet extracted, the resulting ClientData object is added to the list of ClientData objects to be returned by the method. While this handles the task of getting data from each client instance of VMX back to the server's core class, the server is also tasked with forwarding this data onto other clients. To do this the data must be stored somewhere within the server class itself while it waits to be forwarded. The server class uses an array of ClientData objects for this purpose. This array is large enough to hold one ClientData object for each of the clients connected to the server. If a new set of data arrives from a particular client before the last set of data from that client is forwarded, then the new data will overwrite the old, unsent data. There is one exception to this, Screen Texture Data is very infrequently sent, thus if it was overwritten before being forwarded to other clients, then there would be a situation where a presenter had a image on their display screen that differed from that which each other participant in the meeting could see. To prevent this situation, if an old unsent ClientData object in the array contains screen texture data, and the new ClientData object to replace it does not contain screen texture data, then the screen texture data will be copied from the old ClientData object to the new one. This ensures that screen texture data is always forwarded to all clients. It is not necessary to do this with other kinds of data as they are updated much more frequently, so any data that isn’t forwarded will be replaced by newer data before any user would notice.

After the ClientData object is stored for forwarding the method will continue on and loop around to process the next packet if there are more packets that need to be processed. If all packets have been processed then the method will end and return the list of ClientData to the VMX core where it will be used to update the state of the virtual environment.

The other main responsibility of the server is to send data to all of the clients connected to it. This includes data from both the server’s instance of VMX and from all of the connected clients. There are two parts to the system for sending data to clients. The first part is the Send method of the server. This method is responsible for assembling the packets to be sent to each client. The second part of the sending system is the collection of threads responsible for actually transmitting data to the connected clients.

The Send method is inherited from the RemoteCom class and requires a ClientData object be passed to it as a parameter. In the VMX server this ClientData object can be one of two things; it will either contain all of the data from the server’s instance of VMX that needs to be transmitted to the clients, or it will be blank if the server has nothing of its own to send. If a client program has nothing to send, then it will not call the Send method at all. The server cannot work like this due to its responsibility for relaying data between each client. It is likely that the server will have client data to forward, even if it has no data of its own to send.

The first task carried out by the Send method is to check to see if there are any connected clients. If there are none, then the send method stops and immediately returns. If there are clients connected then will begin to build a packet to send to them. To start, a list of bytes called payload is created; this list will be progressively appended with the data from each client that needs to be sent. Also initialised at this point is the byte that contains the flags that indicate which clients have data in the packet (This is the Client Flags byte that was shown on Figure 40).

The method will then move on to loop over each entry in the array of ClientData objects where data to be forwarded is stored. Each entry will first be checked to ensure that it actually contains data (i.e. not a null entry). If data is found the PackageData method will be called on that ClientData object to acquire a byte array that contains the encoded client data. The byte array is checked to ensure that it actually contains something. If it does then the Client Flags byte is amended so that the bit for the client that provided the current data is set; the size of the byte array is encoded into four bytes and added to the payload byte list; and finally the byte array itself is appended to the payload list. The last step of the loop is to set the entry in the array of data to forward to be null, thus indicating that the data for that client has been forwarded. This process is repeated until all of the entries in the array of data to forward have been processed. The array of client data to forward is only ever accessed by main thread, so there is no risk of concurrency issues here.

Once all of the client data has been added to the payload, the server’s own data will be added (if any was provided). The process for adding the server data is similar to the process for adding the client data. First the server’s data is packaged into a byte array. If the resulting byte array actually contains data, then the server’s flag is set in the Client Flags byte, the size of the array is encoded and added to the payload, and finally the byte array itself is added.

At this stage of the full length of the packet, not including the four byte Packet Length field that goes at the start of the packet is calculated. This value is equal to the size of the other fields that go at the top of a server packet (Client ID, Presenter ID, and Client Flags) plus the current size of the payload list. This value will be used for the Packet Length field. Next another byte array is created, large enough to store the entire packet (Packet Length field included). The byte array will serve as the final packet. First the Packet Length is encoded and stored. Next a value for the Client ID is added, this value is just a place holder as it will be changed for each client that the packet is sent to (to match that client's own ID). Next the ID of the current presenter will be added. This value is taken from a public, static variable in the VMX core class which is set by the user in control of the VMX server. After that, the Client Flags byte is added to the array. The final step of building the packet is to copy the contents of the payload list into the byte array.

With the packet built, the Send method moves onto its last stage. It iterates over all of the clients that are currently connected to the server. For each, it created a new copy of the packet that was just built. The Client ID field of the copied version of the packet is then amended to be equal to the ID of the client that is currently being processed by the loop. The copied packet is then added to queue of packets to be sent to that client. Once all of the clients have been processed the Send method ends.

The responsibility of actually delivering these packets to each client falls to the second part of the data sending system. Originally the task of sending data to clients was a part of the Send method. This meant that the main program thread would have to carry out the process of writing the packets to the TCP sockets for each client. Under normal operation this worked well, however trouble would arise if there was a problem with the connection to a client. The TCP socket's write method is a blocking method. This meant that if some problem occurred on the client or with the connection to that client, or a client socket's output buffers were full, then the entire server would freeze, preventing any data from being sent to any client. To rectify this problem the system was changed so that each client would be assigned a thread on the server that was exclusively responsible for sending packets to that client. This means that if something goes wrong sending data to one client, then the operation of the server would not be affected and other clients would continue to get their data.

Each client send thread runs the code in a method called HandleClientSend. HandleClientSend takes one parameter: the VMXClient object for the client for which the thread is responsible. The method itself is fairly simple. It continually runs through a loop. On each iteration of the loop it first checks if the queue of packets to send in the VMXClient object has any packet in it. If it does not, then the thread will sleep for 10ms and then loop back to check again. If there is a packet to send, then it will be dequeued. The TCPClient object that manages the connection to the client will be accessed through the VMXClient object, and its write method will be called. This will send the packet to the client. Once this is done the method will loop back to the start. If a problem occurs with the connection to the client, then the loop will terminate and the thread will exit.

5.8.7 Client

RemoteClient is the class that is responsible for handling the network communications for an instance of the VMX program that is running in client mode (i.e. is not hosting a server). It has a similar structure to the RemoteServer class, inheriting the same methods and variables from RemoteCom. While it is similar to RemoteServer, it is simpler as it is required to manage only one remote connection (the one to the server).

RemoteClient has only two instance variables on top of those it receives from RemoteCom. These are the TcpClient object that is responsible for communication with the server, and the Thread that is used to listen for communications from the server. The TcpClient object is used somewhat differently in the RemoteClient class compared to RemoteServer's use of it. In RemoteClient the TcpClient object is used to establish the connection to the server and is instantiated at setup. In this way it serves a purpose that resembles the function of the TcpListener object in the server. Unlike the server the client object

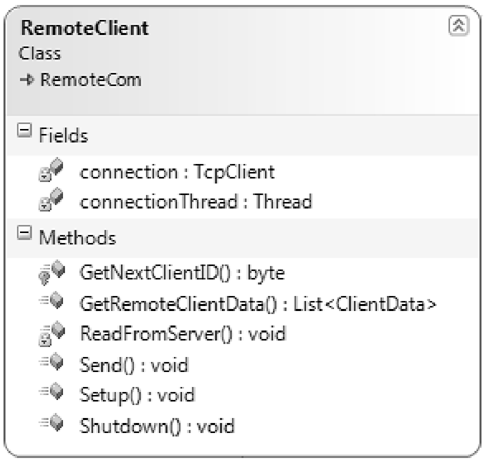


Figure 47: RemoteClient class structure.

only ever uses one extra thread; this is simply because the client has only one remote computer from which to monitor communications (the server).

Like RemoteServer, after instantiating RemoteClient it is necessary to call the Setup method before it can be used for communications. The Setup method for the client is carries out two main functions. The first is to establish a TCP connection to a VMX server. This is done by instantiating a new TcpClient object. The TcpClient object’s constructor takes two arguments, the address of the computer to connect to, and the port to connect on. The ServerAddress and ServerPort properties provided by RemoteCom are used for these parameters. Once the connection is established, a new thread is created based on a method called ReadFromServer. The purpose of the thread is to receive and process packets from the server. Once instantiated the thread is immediately started. The final act of the Setup method is to set the Active property inherited from RemoteCom to true, thus notifying the VMX core that it is ready to send and receive data.

The first stages of the ReadFromServer method are the same on the client as they are on the server. First a loop is entered that will continue until the connection to the server is terminated. The first step of the loop is to call the Read method on the TCP connection to the server. The Read method will be repeatedly called until four bytes are read; these four bytes represent the encoded Packet Length field of the packet. If the Read method ever returns with zero bytes read then the connection to the server has been terminated and the thread that is listening to communications from the server will exit. Once the Packet Length is known a similar series of commands reads in the rest of the packet data. Once read, the packet is added to the queue of newly read packets and the method will then loop back around to await the next packet.

The packets in the received packets queue are processed by the main thread of the VMX program. This occurs when the VMX core makes a call to the client class’s GetRemoteClientData method. This method is inherited from RemoteCom; it is similar to the equivalent method in the server class except that each packet that it must decode is in the server packet format (see Figure 40), and it does not need to store any data to be forwarded later. The method starts in the same way as in the server implementation checking if there are any packets on the received packets queue. If one is available then it will be dequeued and decoded. The decoding process starts by reading off the Client ID stored in the packet. This value is immediately stored in a static variable in the VMX core class called CurrentID (this is how the server gives a client their assigned ID). The next value to be taken out of the packet is the Presenter ID which is stored in another static variable in the VMX class called PresenterID. Next the Client Flags are read from the packet. The method then enters a loop that will on each iteration extracts and decodes the data for a single client. The loop continues until the end of the packet is reached (i.e. all of the client data blocks are read). On each iteration of the loop, four bytes are first taken from the packet. These bytes are converted to an integer that gives the size of the next block of client data. That block is then copied out of the packet and passed to the UnpackData method of a new ClientData object. This decodes the data block and populates the ClientData object's properties and variables with the results. The final step of the loop is to determine which Client ID should be assigned to the new ClientData object. This is done by looking at the Client Flags byte that was read in earlier. Client data blocks for are stored in the packet in a specific order, with data from clients with lower IDs first. The Client Flags byte has a bit for each possible Client ID; this enables the method to determine the pool of Client IDs assigned to the clients that have data in the packet. When reading in client data blocks, the method will assign the next lowest available ID to that data block. Once the appropriate ID is stored into the ClientData object, the object will be added to a list of all of the ClientData objects that have been decoded from the packet. The loop will then go back to the start to decode the next data block. When the loop ends the method will finish by returning the list of ClientData objects back to the VMX core where they will be used to update the program state.

RemoteClient provides an implementation of RemoteCom's send method that is far simpler than the RemoteServer's counterpart method. The Send method takes a ClientData object from the core VMX class that contains the data that VMX needs to send to the server. This data is encoded with ClientData's PackageData method and added to the packet to be sent. The length of this data is then appended to the front of the packet (forming the Packet Length header) and the packet is sent.

Chapter 6: Usability Trial

Throughout the project informal testing was done to refine individual aspects of the software. Chapter 5 detailed many techniques that were tried, and then replaced or refined. The goal of the testing described in this chapter was to evaluate the software in a practical setting. This meant that is was necessary to perform a more formal user test to acquire feedback on the VMX. The experiment was designed to follow an ordinary meeting format, using the VMX software instead of having the participants face-to-face. To get detailed feedback, at the end of their meeting, the participants were asked to fill out a questionnaire that asked questions relating to their experience using the software. The experiment required and was given ethical consent, details on this and more material on the experiment itself can be found in Appendix II.

This chapter starts by giving an overview of the experiment itself, including a description of exactly what was involved and how it was intended to progress. The chapter will then move on to talk about the outcomes of the experiment, including details of feedback received from the participants and the observations of the researcher present during the experiments. This part of the chapter will also include brief notes of what could be done to address some of the issues that were identified in the feedback from users; these notes will be expanded upon in Chapter 7 and Chapter 8.

6.1 Experiment Design

The basic format of the experiment called for a group of people to use the software to hold a meeting. It was intended that the meeting would be ‘real’, that is to say that it would likely have taken place even if it was not part of the experiment. The group of participants for the meeting would be made up of people who had a reason to meet with each other.

The meeting would involve two main phases. In the first phase, the VMX software would be used by a participant in the experiment to perform a presentation to the other participants who would serve as an audience. In the second phase, all of the participants would sit around the virtual table and have a discussion (about the presentation that was given).

Before the experiment could begin, the appropriate equipment needed to be set up. In order to use VMX, a participant must have had, at a bare minimum, an network connected computer with an attached Kinect device, along with the software necessary to run a Kinect device, and the software to run a .net framework application. In order to perform a presentation using the display screen, a participant was required to have a second (large) physical screen attached to their computer and positioned behind them as was shown earlier in Figure 32.

VMX itself has no built in way to transmit audio. This means that if participants of a meeting were to talk to each other, then another application needed to be used. An application called TeamSpeak was selected for this task. TeamSpeak is a piece of freely available software that allows people to connect to a server on the Internet, and have an audio conversation with other people on that server. Team Speak is often used by people who are playing a video game together, and so is designed to be able to run in the background behind a full screen application using 3D graphics; this made it suitable for use with VMX. Most participants used the Kinect’s onboard microphone array to capture their voice, and were equipped with a wired headset for sound playback (using ordinary speakers would have caused audio feedback with the microphone array). The participant who was presenting was provided with a wireless headset instead so they could move around freely while doing their presentation.

In order to ensure that people are only communicating by using the virtual meeting software and Team Speak, the participants in the experiment were all in separate rooms. The participants were instructed to connect to a designated VMX server (controlled by the researcher), and a Team Speak server (provided by the researcher). Once everyone was connected the experiment would begin.

At the beginning of the meeting, the researcher would explain to the participants the features of the virtual meeting software by using the software itself to do a presentation. The features covered included: how to use the manual and the automatic camera controls; the ability to adjust the sizes of the heads of avatars in the scene; how to use gestures to control the display screen when presenting; and how to update what was shown on the display screen. Along with the demonstration of the software’s features, the researcher also gave an explanation of the various parts of the virtual environment, including the various display screens and most importantly, the other avatars in the scene and how they moved and acted with respect to their users’ own movements. Because the software is running throughout the introductory presentation, participants would be encouraged to experiment with their controls during the presentation. Once the researcher had completed their presentation, the participants were invited to ask any last questions that they began holding their own meeting.

When the participants were ready to begin their meeting, the researcher would (in avatar form) sit down at the virtual table where they would remain for the duration of the experiment to observe the participants while they carried out their meeting. The experiment would continue with the participants performing presentation and/or sitting at the virtual table having a discussion. The meeting would go on until the participants were finished.

After the meeting, all of the participants would be asked to fill out a questionnaire. The questionnaire would be handed out to the participants before they began their meeting. This was done to give the participants the chance to see what kinds of things would be asked, thus giving the participants the opportunity to think about their answers while they were using the software. The questionnaire contains questions about the participant’s experiences using the software.

The questionnaire was split into five sections. Each section had questions about a particular part of the program. There were sections about: the features of the VMX software; the experience of giving a presentation; the experience of the meeting in general; how the virtual meeting software compared to other methods of holding a meeting; and a final section for general comments. A copy of the full questionnaire can be seen in Appendix II.

6.2 Outcomes

The experiment was held between five people (with the researcher observing as a 6th participant). It involved one participant giving a presentation to the others, followed by a discussion of that presentation between all of the participants.

This section looks at the observations that were made by the researcher during the course of the experiment. It also discusses the feedback that was received from the participants of the experiments on their questionnaire forms. Identifying both the successful and unsuccessful elements of the program, and giving consideration about how to address the problems encountered.

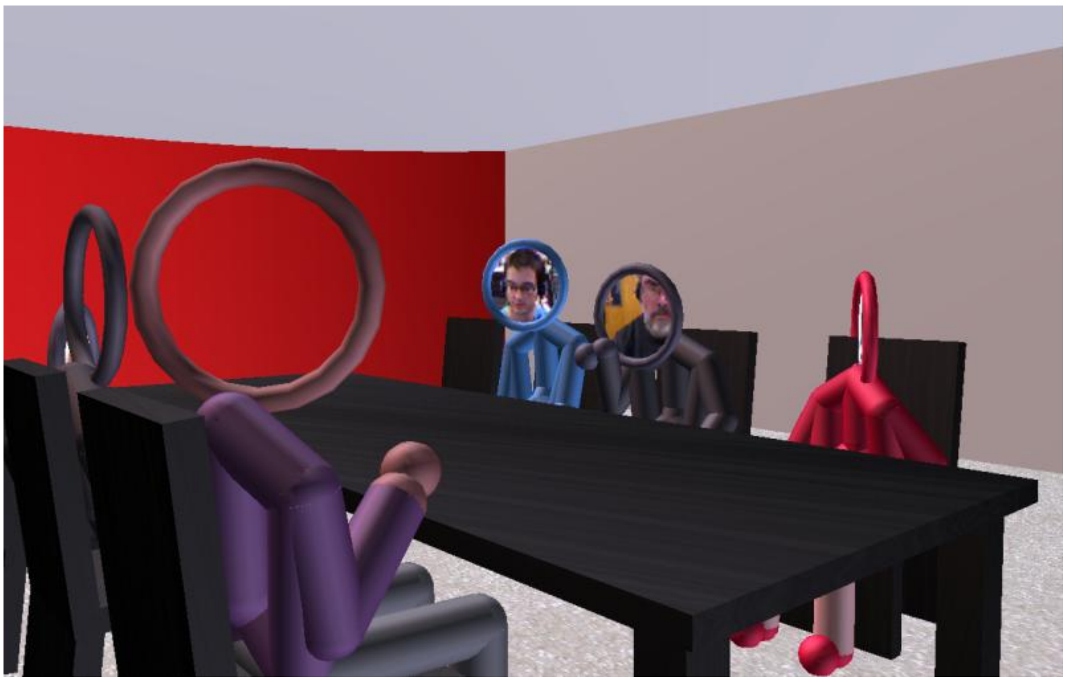


Figure 48: The meeting in progress.

6.2.1 The Avatars

The first trouble that was encountered with avatars was the obstruction of view caused by the heads of the avatars. Participants in the meeting commented on their inability to easily see past their neighbours at the table to see other participants further down, and in some cases to see the virtual display screen. The fact that the back of an avatar's head is transparent did alleviate the severity of this problem, but it did not fully solve it. Some participants reported that they used the manual camera controls as a means to get around this problem, especially when trying to watch a presentation. No one reported attempting to shrink avatars' heads to see past them. Possible solutions for this could include, making the avatars' head even less visible from behind, or investigating different ways to arrange avatars in the environment, to minimize the chance that they would cause an obstruction. Figure 49 shows the problem in practice, the avatars in the foreground are partially obstructing the virtual display screen.

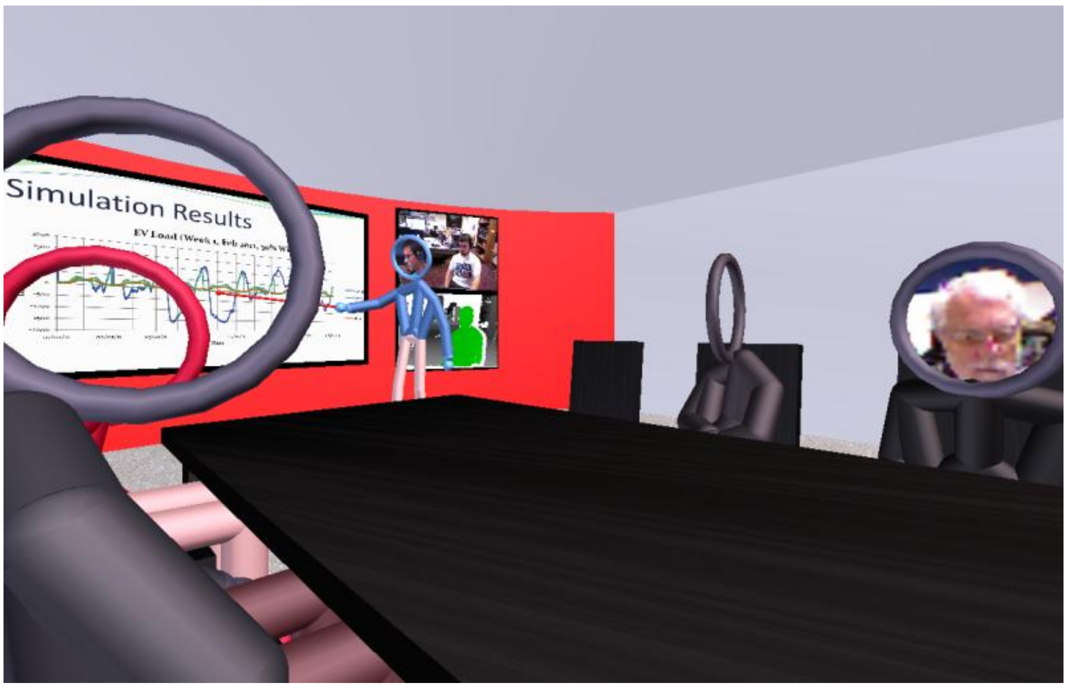


Figure 49: The presentation in progress.

In tests of the VMX system that were run before the experiment was carried out, a problem regarding the amount of data that was being sent across the network was identified. This necessitated a restriction on how detailed the face texture for an avatar could be. In the experiment this resulted in a lower than ideal resolution for the face textures (specifically 35 x 35 pixels). It was still possible to see and recognise people's faces at this resolution, but participants reported some difficulty in seeing the facial expressions of others in the meeting. This was particularly true of the presenter, while they were performing their presentation. The experiment setup (shown in Figure 50) illustrates a possible reason for the presenter's trouble with this. It shows the area in which the presenter did their presentation during the experiment. The real world screen appears on the far right of the image, this is where the presenter was standing while they were giving the presentation. On the far left of the image the monitor on which the presenter’s audience appears can be seen. The presenter was standing a significant distance away from this monitor (approximately two metres), meaning that their audience would have appeared quite small, compounding the difficulty in making out facial expressions. Despite having trouble making out facial expressions, participants did note that they were able to see if a person’s lips were moving from looking at their face texture. One participant also reported being able to see in which direction people’s eyes were looking. The presenter did not attempt to change the size of their audience’s avatar’s heads to get a better view of their faces. Doing so may have helped the situation; however there was no way for the presenter to this from where they were standing, so they would to have had to return to their keyboard. This could have resulted in several trips back and forth to settle on an appropriate head size for the viewing distance.

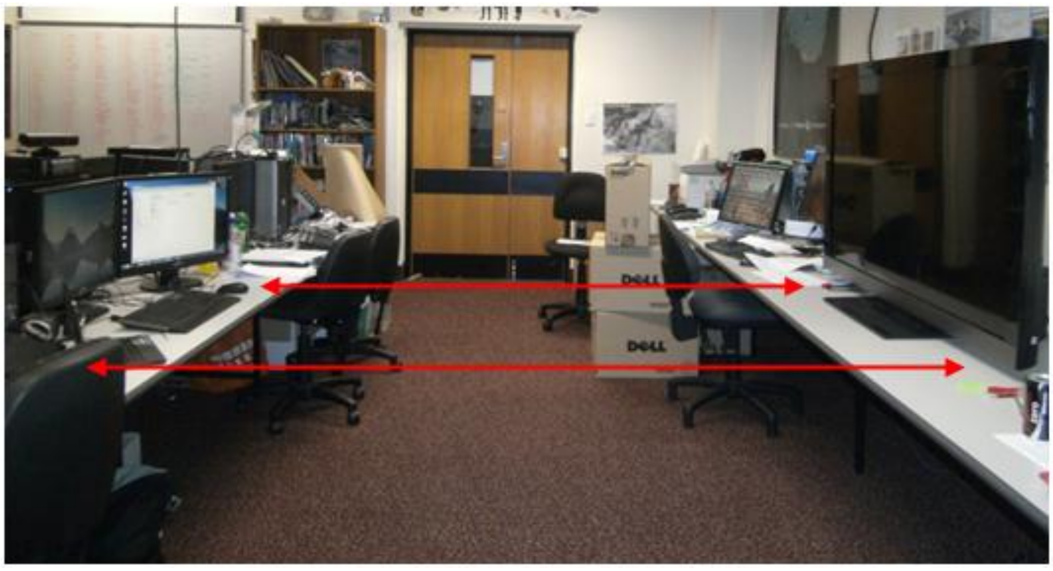


Figure 50: Presenter's experiment setup.

Other participant’s views on the usefulness of the head size adjustment feature were mixed. Most attempted to use the feature and reported that they felt it might be of some use, but only one noted that they actually used it beyond simply trying it out. The reason that they reported for this was to see the faces of speakers more clearly, especially when that speaker’s avatar was far away in the virtual environment.

In the questionnaire, the participants were asked what kind of body language they were able to see from other participants. No participant reported being able to identify minor aspect of body language from other user’s avatars (e.g. body posture). However most participants reported being able to recognise when somebody was applauding, pointing somewhere in the environment, raising their hand to ask a question, or anything that involved significant hand or arm movements. At the end of their presentation the presenter asked the other participants in the meeting if they had any questions, and instructed them to raise their hands if they did. It was clear from their avatars who was doing this and the presenter was able to pick those who had0 questions. It should be noted that one of the participants stated that they has raised their hand during the meeting and that it had gone unnoticed by the presenter. In response to this the presenter noted that they had not been paying close attention to their audience’s actions during the presentation. It is worth noting that the same issue can arise in a real meeting. It might be possible for an avatar to make itself more noticeable to the presenter in a virtual meeting.

A common complaint amongst three of the participants regarding avatar’s body language was that it was hard to pick out which avatar movements were intentionally caused by users and which were caused by jitter in the skeleton position data from the Kinect. Another complaint one user made was sometimes avatars appeared positioned in bizarre and unnatural ways; this was likely caused by the Kinect misinterpreting its depth data when evaluating joint positions.

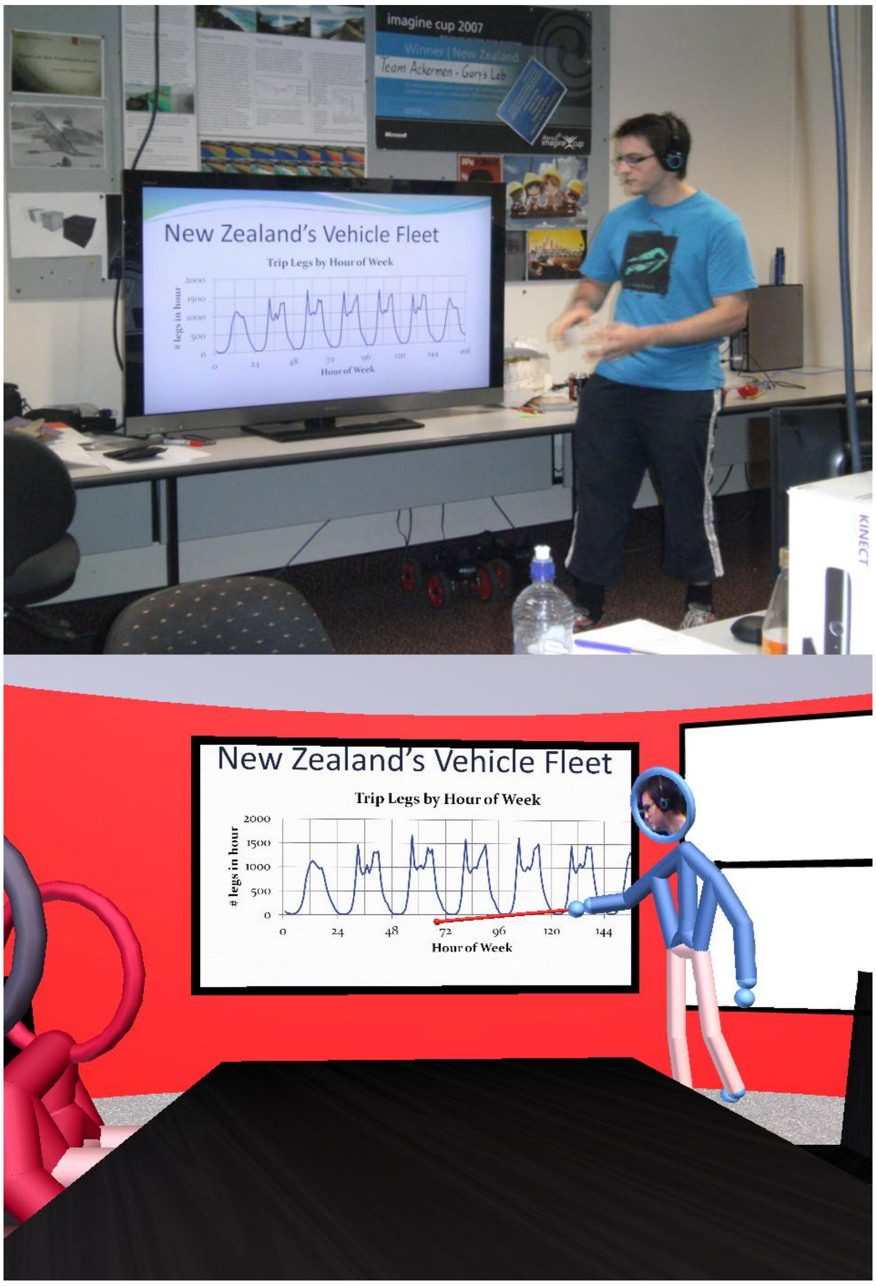
Participants reported mixed results when it came to identifying which way other participants were looking in the virtual environment. All participants reported that they usually had a good idea of where other users were looking, but some reported that it was not always clear. In particular, some participants said that they had trouble deciding if someone was looking at them directly or not in some cases.

Four of the participants stated that they were able to pick out who was talking by looking at the speaker’s avatar (though one reported that they could not). The main reasons cited by participants for why this was possible were the ability to see lip movement on the speaker’s face texture, and hand gestures in the speaker’s avatar’s movements. Despite these indicators, multiple participants suggested that it would still be useful to include a user interface element that show who is speaking, either in the form of an icon that appears above the head of a user's avatar, or by modifying their avatar directly somehow (e.g. altering the shape of its head).

One part of the questionnaire asked participants if they could tell when other participants were involved in activities that were not directly related to the meeting (e.g. browsing the web, checking emails). Most reported that they did not notice anyone else doing any of these activities, though most of those involved in the meeting reported that they did in fact engage in activities outside of the meeting. There were however two participants that said that they did notice this behaviour. One reported that they were able to tell when somebody was doing something else because that user's avatar would appear to reach forward towards the virtual table (presumably this was caused by users reaching forward to use their keyboard or handle some object in front of them). The other reported that it was the direction that a person's eyes were looking and the movements of their head that revealed when a person was not paying attention.

6.2.2 Virtual Screen

In the experiment, the gesture recognition system was used by the presenter when they were doing a presentation to control the contents of the virtual screen. During the course of the presentation a problem with the system immediately became apparent. The presenter was standing to the side of the Kinect's field of view (so as not to stand in front of the real world screen). Sometimes the presenter would briefly leave the Kinect's field of view. This would cause the Kinect's skeleton engine to make wildly inaccurate assumptions about the user's skeleton joint positions. These poorly predicted joint positions were sometimes able to trigger the gesture recognition system which would cause the image on the screen to be accidentally zoomed and scrolled. This problem was compounded by the fact that zooming and scroll operations only affect the image on the virtual display screen, not the real world display screen. The presenter did have a monitor showing them an up to date image of the virtual display screen (to allow them to see what they were doing when they actually wanted to use gestures), however it appeared that they did not normally look at this monitor, so the accidental changes would go unnoticed. This suggests that the system should be made to ignore gestures when there is reason to believe that joint positions are not accurate. Figure 51 shows the display screen and presenter in action; note the accidental zooming that has occurred on the virtual screen (parts of the image are cut off).



imagine cup 2007
Winner | New Zealand
Team Ackermen - Gary's Lab
New Zealand's Vehicle Fleet
Trip Legs by Hour of Week
# legs in hour
Hour of Week
KINECT
New Zealand's Vehicle Fleet
Trip Legs by Hour of Week
2000
1500
1000
500
0
# legs in hour
0
24
48
72
96
120
144
Hour of Week

Figure 51: Real vs. Virtual Environment

The presenter expressed that there was some difficulty when using the scrolling in zooming gestures, though they also thought that with practice it would become easier. During the meeting it was observed that the presenter had some trouble in selecting the gesture they actually wanted (i.e. they would change the zoom on the image when they meant to scroll it). This suggests that either a new set of gestures are needed, or better feedback needs to be incorporated into the existing system.

One significant observation that was made during the experiment was that due to the way the presenter interacted with the screen, the command to transmit the screen texture would be more frequent than was necessary. Because of the size of the image that must be transmitted, a noticeable pause occurs (for about one second) in all client programs when this happens. The pause is caused by the time it takes the server to upload the image to the clients, and the time that the clients take to decode the image data and convert it to a texture. During their presentation, the presenter would often point at the screen and then drop their hands to their side; sometimes they would do this repeatedly, this would trigger the screen texture to update. The result was a quick succession of pauses in the rendering of 3D graphics on each of the clients as new screen textures arrived (a single pause would last approximately one second). There a possible solution to the issue would be to improve the code that converts images to textures to operate more efficiently, using compression to decrease the amount of data that is needed to send the screen image could also help. An alternative solution would be to change the mechanism for triggering a screen image transmission instruction to make it harder to trigger unintentionally.

The “laser pointer” feature that allows a presenter to point at specific locations on the virtual presentation received mostly positive feedback from those participants that commented on it. During their presentation, the presenter commented at one point that they needed two laser pointers – one for each hand. This happened when the presenter was attempting to use both hands to point at two different places on a graph for comparison.

During the meeting an unexpected use for the virtual screen was developed by the participants. One participant suggested that the screen be used to keep notes of important points made in the discussion phase of the meeting. This use of the screen was successful; however the interface for using the display screen was not designed for it, and consequently was clumsy to use. A person had to be put into presentation mode (i.e. placed at the front of the virtual meeting room, in control of the screen) during the discussion phase, and type the notes manually onto the screen. That person would have to go over to the real world screen and tap it to trigger transmission of the updated screen image to other users after each note was written. This suggests that it would be desirable to create a new interface element to allow participants to collaboratively modify the contents of the screen during a discussion in a meeting.

6.2.3 Camera Controls

There was a wide array of preferences among the participants about how and when they used the manual and automatic camera controls. Two participants decided to forgo use of the manual controls entirely, others found uses for both types of camera controls, and some preferred the direct control offered by the manual camera.

A common theme among the participants that used both kinds of camera was a preference for using the automatic camera during the discussion phase of the meeting, and the manual camera during the presentation phase of the program. Those participants stated that the reason for preferring the manual camera during the presentation is that it allowed them to select a place where they had a clear view of the presenter and virtual display screen. The reason they gave for using the automatic controls during the discussion phase was that it was useful for changing who they were looking at around the table at any given time.

A common criticism made by two participants was that the automatic camera controls were not very stable. This was likely a consequence of two factors. The first being that the controls were overly sensitive, small movements in a user's position would cause large movements in the camera's view. The second being that jitter in the Kinect's estimation of joint positions can cause the camera to shake about slightly. One participant stated that their struggles with the automatic controls broke their immersion in the experience. After the experiment one participant suggested that a button to freeze the position of camera at any given time would have been useful. This can actually already be achieved by the switching the camera into manual control mode, and then not actually using the manual controls; the camera will remain where ever it was before the mode was switched. However this use of the manual camera mode was not considered before the experiment so was never suggested to the participants. The negative reactions to the automatic controls were not shared by all participants with some stating that they found them easy to use, intuitive and immersive. One participant, who otherwise liked the automatic controls, reported that they would have liked a better way to trigger the automatic camera to zoom in.

The manual camera controls were not used by all participants (two didn't use it at all), but one preferred it at all times. Commonly cited advantages were that the camera was more stable when this mode was used, and that the added control allowed participants to get a better view of what they wanted to look at. Some participants suggested alternate control methods for the manual camera. A common suggestion was to move the control for the camera's direction from the mouse to the arrow keys on the keyboard; this would have the secondary benefit of freeing the mouse up for other uses in the program (e.g. interacting with a GUI). One participant also suggested using explicit gestures (along the lines of what is used to control the image on the presentation screen) to control the camera.

6.2.4 Comparison to Other Types of Meetings

When compared with other ways of holding remote meetings (video conference, teleconference, non-Kinect virtual meeting) all participants reported that the experience of using VMX was as good or better. In general participants seemed to feel more comfortable in the virtual meeting, often reporting that the experience felt more "relaxed", "fun", and "informal" than a video conference or teleconference. Participants also described the experience as more "immersive" or "engaging" than other forms of remote meeting. One participant, who had been involved in the Second Life experiments in "Virtual Worlds as Meeting Places" stated that they felt that the experience of meeting in VMX was significantly better then Second Life, citing the more natural controls and interface in VMX giving them the feeling of “being there”. That participant also said there was a feeling of having a shared space with the other participants, a feeling not echoed in video conferences.

Comparisons to real world meetings were not as favourable. No participant felt that their experience using VMX was as good as holding a real life meeting. However some participants felt that with additional development and improvements that the VMX meetings could become comparable to real life meetings.

Chapter 7: Conclusion

The overall goal of this project was to use the Kinect to improve the experience of participating in a virtual meeting. In the opening chapters of this document, a wide variety of areas where improvements could be made were identified. This chapter starts by looking at the improvements and features that were implemented during the project, considering the value of each in terms of how successful it was at fulfilling its intended purpose. The chapter will then move on to discuss in broader terms what was achieved in terms of the core goal of the project: to explore how the Kinect can be used to improve the experience of participating in a virtual meeting.

The user avatars used in virtual meetings were identified as the largest area for improvement. The list of existing problems with avatars that were discussed in Chapter 2 included: lack of a means to confirm the identity of a person controlling an avatar; difficult and cumbersome to use controls for manipulating avatars; limited ways for expressing body language; and limited non-verbal communication in general.

The lack of ability to identify the user behind an avatar was identified in Virtual Worlds as Meeting Places. It stemmed from the fact that avatars in Second Life appear the same regardless of who is controlling them. VMX addressed this problem by incorporating a video feed of a user’s face (acquired from the Kinect device) onto their avatar’s head. This was immediately successful in solving the problem by allowing all participants in a meeting to visually identify the each other.

The incorporation of a user’s face onto their avatar also partially addressed the problem of a lack of means for non-verbal communication. The idea being that the video feed allows participants to pick out details of a user’s facial expression. Participants in the usability trial reported that they were able to pick out various details of other participants faces, including the direction their eyes were looking, and movements of their lips. This was despite the low resolution of the face textures, suggesting that the facial video feeds, even with limited detail, do have the ability to successfully to allow non-verbal communication.

The other aspect of non-verbal communication, body language, was also addressed in VMX. It was in this area that the abilities of the Kinect proved most useful to the project. The avatars in VMX were animated using the Kinect’s 3D skeleton position data of the users controlling them. This meant that a large part of the position of an avatars body was taken directly from the position of the user. The result was that avatars can emulate the body language of the user with no special effort on the user’s part. In the usability trial participants reported that they were successfully able to see certain aspects of body language, particularly when that language involved large movements. The sometimes erratic movements in the positions of skeleton joints as determined by the Kinect did limit the ability of users to see finer details of other users’ movements. From this, it is clear that animating avatars using this method improved the ability of users to communicate non-verbally, though there is still room for improvement in handling inaccuracies in the Kinect’s data.

The use of Kinect data to control avatar’s movements also solved the problem of the clumsy mechanisms for controlling avatars that were available in Second Life. One of the participants of the usability trial for this project had previously used Second Life to participate in a virtual meeting. They characterised the experience of trying to control Second Life avatar as “struggling with the interface to a puppet theatre”. That participant and others reported that they found controlling VMX’s avatars with the Kinect straightforward; no participants suggested that they encountered any trouble whatsoever in getting their avatar to do what they wanted. This evaluation indicates that this application of the Kinect to control avatar movements was successful in overcoming the existing difficulties with avatar control in virtual meetings. This use of the Kinect is perhaps the most successful of all of the uses explored in this project.

One aspect of an avatar’s movement was not (directly) determined from the Kinect’s skeleton position data: the direction in which an avatar’s head faces at any given time. Instead this is set from the current direction of an avatar’s user’s view of the virtual environment. The primary reason for doing this was to add an additional element to the avatar’s body language: an indication of what a user was looking at in the virtual environment. In existing virtual meeting software, and also in video conferencing this is not always clear. The results of the usability trial showed that this feature was reasonably successful, with all users reporting that they were able to tell where another user was looking in most situations. Significantly, one user cited this feature as improving their sense of immersion in the meeting. The sense of a consistent spatial relationships between participants and the ability to directly evaluate where participants were looking within the space, gave a sense of ‘being there’.

Overall the implementation of avatars in VMX is probably its most successful element. The data that the Kinect device provides was very well suited to improving avatars and this is shown in the positive response that the user avatars received from the participants of the usability trial. The avatars in VMX were able to address all of the key problems that were identified earlier to a degree. The largest criticism of the avatars was their erratic movements, caused by jitter in the skeleton joint position data from the Kinect. While work could be done on VMX to reduce the impact of this jitter, it is likely that future iterations of the Kinect hardware and software will work to improve the accuracy of the data the Kinect provides. Indeed improvements were made to the skeletal tracking system in both the update from the Beta 1 to the Beta 2 version of the Kinect SDK (Microsoft, 2011) and the update from the Beta 2 version to the official release (Microsoft, 2012).

Beyond avatars, this project incorporated another means of simplifying the control mechanisms of virtual meeting software using Kinect. This was in the form of the automatic camera controls. These controls were designed with the intention of reducing the need of the user to interact with their mouse and keyboard during the meeting, freeing them to pay attention to the events of the meeting. The controls directed the user’s view of the virtual environment based on skeleton position data. The user was able to look left and right in the environment (from their avatar’s point of view) and zoom in and out using only subtle body movements. As was seen in Chapter 6, there were mixed feelings among participants about the usefulness of this feature. The feedback that was given indicated that the feature was partially successful in allowing hands free control of where the user was looking, but that there is still room for improvement. The periodic instability of the camera’s view was of particular concern to the participants. Part of the cause of this instability is the aforementioned jitter in the Kinect skeleton position data, so this feature is also likely to be improved by future updates to the Kinect hardware and software.

Overall the automatic camera controls in VMX were reasonably successful in reducing the attention that users needed to give to the programs controls. Most users appeared to find it easy to use during the trial. The fact that during the usability trial some participants were able to happily use the automatic camera controls through the entire meeting and never needed to resort to using the manual controls is an encouraging sign. It suggested that with the right improvements it may be possible to get the automatic controls to the point where they are useful enough to remove the need to include manual controls at all. The success of the automatic camera controls can be embodied by one participant's description of them as being "easier to use, even when not thinking about it".

This project also aimed to explore ways to incorporate the Kinect to improve the experience of giving a presentation to an audience in a virtual meeting. Areas where there appeared to be room for improvement included the ability of a presenter to gesticulate to their audience and the limited options where it came to utilising visual aids.

The user's ability to directly control their avatars movements with their body as discussed earlier immediately offered a way for a presenter to gesticulate. This functionality was as successful at addressing this issue as it was at allowing participants' body language to be reflected by their avatars, and was subject to the same limitations (jitter in the Kinect's reported joint positions) as earlier described.

The key component for providing a presenter with the ability to use visual aids is the virtual display screen. The virtual display screen has a number of features that extend its versatility. These include: the capability of displaying an image to all of the participants in a meeting; the capability for the user to zoom in and pan across the image on the display screen with hand gestures; and the facility to be used with a real world counterpart which the user can interact with by pointing, or drawing with their fingers as if it were a whiteboard.

During the usability trial a participant performed a presentation. They used the display screen to give a slide show to the audience. From this it was possible to see that for the purpose of allowing visual aids to be used, the display screen was successful. The presenter was able to speak and gesture to the slides in the virtual environment, and the audience was able to watch what was going on, as if they were seeing the presentation in person.

The gesture controls for manipulating the image shown on the display screen proved to be partially successful for helping the user when giving a presentation. During the usability trial the presenter took advantage of the functionality several times. The feedback from the presenter did reveal a problem however; the presenter found it difficult to use the gesture accurately. While they did also say that it would likely become easier with time and practice, it does still suggest that there is room for improvement in making the gestures easier to use and more intuitive. The difficulties with the gestures could have arisen from the similarity of the different gestures for zooming and panning, or possibly the lack of feedback that was available to the user about what gesture they had activated (in the usability trial the presenter only had a small monitor depicting the changes they were making to the display screen image available, and were unable to see the colour coded avatar hands described in Chapter 5).

The ability for the user to point at a position on the real world screen and for that to be reflected by their avatar pointing at the same position on the virtual screen was one of the more successful areas of the project. The feature was used extensively in the usability trial by the presenter, and no problems were reported with it by any participant (though further enhancements relating to it were suggested). Those participants that did comment on it were pleased with it as a presentation tool. From a technical stand point, the feature was successful in solving a limitation of the real world to virtual screen relationship (as described in Section 5.6.3).

One feature that was never truly successful was implementing a way for a user to treat the presentation screen as a whiteboard, using the Kinect to track their finger movements across the screen to determine where they were ‘drawing’. The idea for the feature was suggested in Virtual Worlds as Meeting Places as a possible enhancement to virtual meetings in Second Life. In the end the technical difficulties of getting the Kinect to report the position of the end of a user’s finger with consistent accuracy proved too difficult to overcome and made it impossible to draw anything accurately. This highlights one of the key limitations with the current Kinect technology: the limited degree of accuracy in the skeleton tracking system.

Ultimately, most of the problems with existing virtual meeting applications that were targeted by this project were solved, though with varying levels of success. It was almost invariably the aforementioned lack of accuracy in the skeleton tracking system that led to trouble in completely eliminating the targeted issues. The limited accuracy prevented subtle body language from being picked up by users; it led to periodic breaks in users’ immersion through unusual, inconsistent and improbable joint position estimation; and it prevented the whiteboard capability of the virtual display screen from being usable. Despite the problems, the lack of accuracy was not significant enough to prevent the use of the Kinect in many areas. The success of many of the feature that have been covered in this chapter show that there is most certainly potential for motion controllers such as the Kinect to be used to improve virtual conferencing.

Part of the questionnaire in the usability trial sought feedback from the participants as to how they felt VMX compared to other forms of meeting. The comments that were received give the clearest indicators of the how successful the use of the Kinect is in making virtual meetings a good way to meet with others.

When asked about how virtual meetings in VMX compared to video conferencing and teleconferencing the participants frequently described the experience of being involved in the meeting as feeling more ‘fun’, ‘relaxed’ or ‘informal’. This suggests that the participants felt more comfortable in the meeting, focusing on the meeting itself more than the software being used to facilitate it. This idea is further supported with participants, throughout their questionnaires, commonly reporting feeling immersed when using the software and certain features. The feedback that was received in this regard validates the idea that virtual meetings are worth holding at all, it indicates that there are real advantages to virtual meetings when compared against other forms of remote meetings.

Information was also sought in the usability trial about how participants felt VMX compared to real life meetings. The feedback received showed that while apparently an improvement on remote meetings, VMX still has a long way to go before it can be as good as a meeting in person. This clearly reveals that the experience of virtual meetings in VMX is not totally immersive. There are still many avenues of exploration for VMX however (which will be discussed in Chapter 8) so there is still potential for virtual meetings to become more competitively matched against real life meetings.

One of the valuable aspects of the some of the participants in the usability trial is that they were also present at the experiments held in Virtual Worlds as Meeting Places. This meant that they were able to give clear feedback about how the experiences between virtual meetings in VMX and in Second Life compared. The feedback that was given was universally in favour of VMX. The Kinect enabled controls were described as being less clumsy and easier to use than their counterparts in Second Life. It is this that gives the clearest indicator of success in this project. Most of the objectives that led to the features that were chosen for VMX stemmed from problems and limitations that were encountered in Second Life. VMX's favourable comparison with Second Life for holding virtual meetings clearly demonstrates that VMX was successful in developing ways to address the issues that Second Life presented.

Using Motion Controllers in Virtual Conferencing set out to apply the newly accessible technology that was made available by the Kinect to improve the experience of holding a virtual meeting between participants across the Internet. Over the course of the project VMX was developed and tested to find out if the Kinect could be successfully used in this way.

The work that was done over the course of the development of VMX clearly shows that it is possible to apply the motion controllers to create new features for virtual meetings, and the informal testing and usability trial have demonstrated that these new features can provide profound improvements to the experience of virtual meetings. It has been demonstrated that the ability of the Kinect's skeleton tracking system to allow the user to control their avatar with their own body movements is both intuitive and functional. Freeing the user from needing to use the keyboard and mouse in the control of their avatar, and in the control of their view of the virtual environment has successfully increased the sense of immersion that users experience while participating in a virtual meeting, allowing them to focus more attention on the events of the meeting and less on manipulating the software they are using to facilitate it. VMX has utilised the Kinect to allow users to give a presentation in the virtual world by directly performing that presentation in the real world in front of the device. VMX has also demonstrated that the Kinect can be used to give users the ability to control the virtual environment using gestures.

All of these new applications clearly demonstrate that the core goal of this project: to improve the experience of virtual meetings with the Kinect device, has been successfully achieved and that the use of motion controllers can most definitely improve virtual meeting software.

Chapter 8: Future Work

Many uses for the Kinect device in the context of virtual meetings have been investigated throughout this project, however there are still many more potential avenues of exploration. This chapter will discuss some of these possible lines of future investigation. The features and improvements that are discussed come in two forms: potential tweaks and improvements that were identified from problems that arose during user test experiment; and areas of exploration that are natural progressions from what has already being done in this project.

8.1 Screen Depth

Currently, the distance from the Kinect device to the real world presentation screen is loaded into VMX through the configuration file (where it must be set manually by the user). It is not straightforward to reliably retrieve this information automatically at runtime due to the fact that the location of the screen is only determined in Kinect colour image space (for reasons discussed in Chapter 5). A system that could detect the depth of the screen would increase the robustness of the system against movement of the Kinect device or of the screen. It would also allow for more precise detection of the closeness of the user's hand to the screen. This could be potentially be used to enhance the whiteboard functionality of the display screen, only drawing when the user physically touches the screen, or perhaps changing the weight of the lines drawn based on how close their finger was to the screen.

There are a few possible ways that could be explored towards finding a solution to this problem. The simplest of these would be to enforce restrictions on what can be used as a real world screen. Requiring that some specific part of the screen (e.g. the border) must be non-reflective would allow the development of an algorithm that could analyse the Kinect depth image to find the screen instead of analysing the colour image data. The depth of the screen could then be found directly from its position data.

Another solution would be to exploit the fact that even on a reflective screen, the Kinect is able to detect the depth of the parts of the screen that face the camera head on (see Figure 31 and Section 5.6.2). An algorithm could be developed that would attempt to estimate where a accurate depth value for the screen was on the depth image, and then retrieve that value. The greatest difficulty with this approach would be verifying the correctness of the result.

8.2 Avatar Improvements

The user avatars in VMX provide a wide range of areas for further work, particularly with respect to their appearance, and the information they convey through that appearance. An avatar's head and hands are especially interesting as they are the areas where the Kinect is most limited in picking out important details (facial expressions, finger positions etc.).

8.2.1 Avatar Hands

One suggestion that was made by a participant in the experiment was to improve the hands of the avatars in VMX. Currently hands are represented by spheres and give no information about a user's fingers. As discussed in Section 5.5.2 the Kinect SDK does not provide finger positions, and current techniques for acquiring this information are computationally expensive.

One solution to this would be to apply the same algorithm that picks out a users face on the Kinect colour video image to pick out their hands. The idea would be to show live video of a users hands on their avatar. This would allow all participants to see what a user was doing with their fingers.

There would also be a potential secondary benefit to doing this in that would allow a user to show other participants real world objects that they were holding. These objects could serve as visual aids in a discussion or presentation.

8.2.2 Avatar Heads

The current heads used by avatars have a number of drawbacks. One drawback is that they are quite large and can obstruct users' views of things in the virtual environment. Another drawback is that it is not possible to see a user's face from a side on view. Both of these drawbacks could be addressed by incorporating depth data from the Kinect when generating an avatar’s face.

The current avatar heads are large and circular. This contributes to the problem of view obstruction. As it stands the heads show more data than they need to. The simple algorithm for extracting user face textures picks up a lot of the background behind the user, which is then displayed on the avatar’s head. The Kinect depth data can be used to determine the precise boundaries of a user’s head on the colour video image. This information could be used to give an accurate “cardboard cut out” of the user’s head and face. If this “cardboard cut out” was used in place of the current circular head, the head would be smaller, and less prone to causing view obstructions.

This “cardboard cut out” approach would not address the fact that an avatar’s face is not visible from the side. To solve this, the depth data from a user’s head could be used to extrude the flat “cardboard cut out” into a realistic face shape. This would give an avatar’s head the appearance of being a 3D floating mask that reflects the appearance and facial expression of the user it represents. Because the face would then have a 3D shape, it would be visible from the side. This approach essentially makes use of the Kinect as a 3D scanner; such a use has already been demonstrated as being feasible (University of California, 2011).

8.3 Further Exploitation of Virtual Reality

Further work could be done in finding ways of utilising the virtual world to make it possible to do things that could not be done in a real world meeting. As it stands only a small number of features of VMX make use of this fact. There are many other features that were considered for inclusion that ultimately were not implemented in this project.

8.3.1 Personal Display Screens

One such feature would allow any user to make use of the display screen, without standing up. The idea would have been to have a user 'call' the display screen to them. The screen be scaled down and would fly over to sit behind that user at the table, the user could put what they liked on the screen and gesture to it (with or without a real world counterpart for reference). Other users at the table would see the small display screen behind the person using it at the table so would simultaneously be able to see the user and their screen at the same time without looking at the location of the main display screen.

There is of course no requirement that the main display screen be used for this purpose. A smaller display screen could simply appear behind any user who needed it, leaving the main display screen where it normally is. Under this model there could be as many of these screens at one time as there are users. This would allow users to display whatever they wished next to themselves throughout a meeting. This could include things like illustrative diagrams of what they might be talking about, a website that they were referencing, or anything else they wished.

8.3.2 Always Visible Faces

Another possible feature could give the user the option to force all of the avatars in their instance of the virtual environment to look directly at the virtual camera. This would mean that a user could guarantee a clear view of all of the faces of all of the participants in the meeting, regardless of where participants were actually looking. This feature could be further expanded to force avatars heads into different locations if one head was obstructing the view of other heads, for example if one user's avatar was sitting between a second user's camera and a third user's avatar, then the head of the third user's avatar could be repositioned so that it appeared above the first user's avatar. The result would be the appearance of a slightly disembodied head floating some distance above its body.

8.3.3 Meeting Table Shape

In the experiment, the meeting table had rectangular shape with participants sitting along two sides, as shown in Figure 52. This shape serves as a compromise between being suitable for seeing and talking to other participants around the table, and for viewing a presentation. In practice the compromise does mean that the table shape was not perfect in either circumstance. During presentations, participants sitting the furthest away from the presentation screen can have their view of the presentation obscured by the participants in front of them. During discussions participants who are sitting on the same side of the table but at opposite ends can have trouble seeing each other if there is someone else sitting between them.

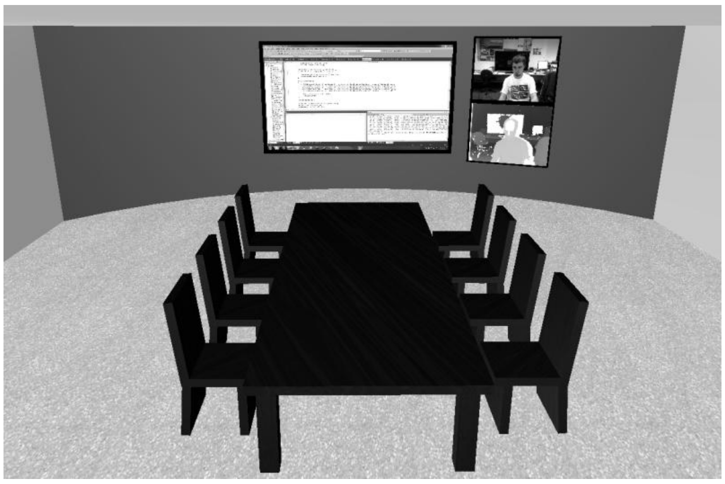


Figure 52: Current meeting table shape

Because the meeting is in a virtual environment, there is no reason why the shape of the table must stay the same between discussion phases and presentation phases. This provides the basic idea for a system that could solve the problems with the current table shape. Figure 53 shows what might be considered an ideal table for discussions. The table is circular meaning that all participants should be able to get a good view of all other participants. This circular shape is not very good for presentations however, as participants nearer the screen are facing in the wrong direction and can also obscure the presenter and screen from the view of the participants on the far side of the table.

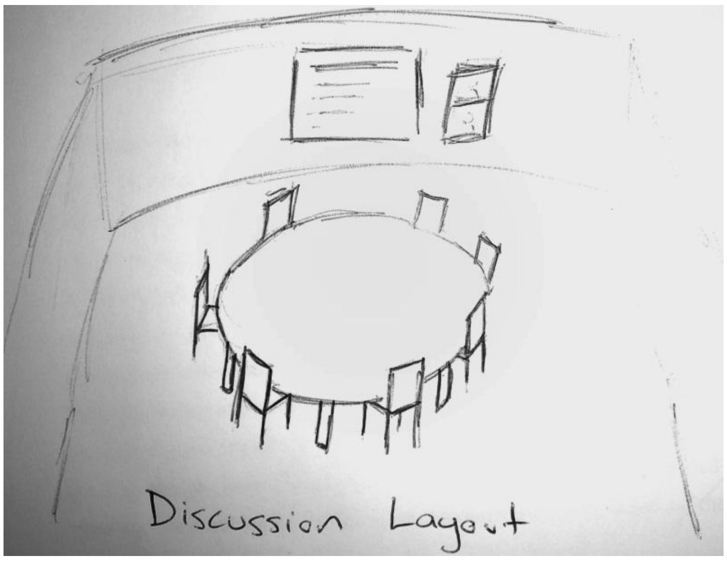


Figure 53: Circular table design

Figure 54 shows how the table shape could be changed during a presentation. This shape allows all of the participants to directly face the presentation screen, and also prevents any participant from obscuring the view of any other participants.

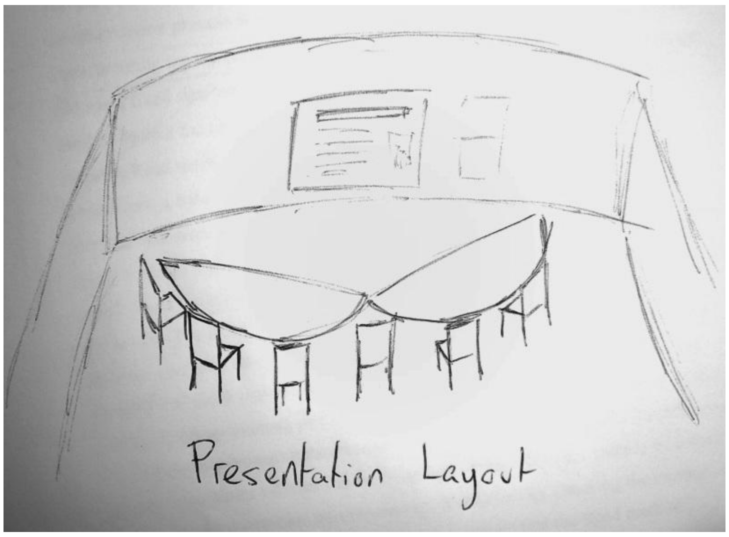


Figure 54: Spilt table design

Animating the change between table shapes could help ensure that a user's sense of immersion in the environment is not lost.

8.4 Audio

This project makes no use of the Kinect's microphone array or audio output. In practical use during the experiments, a third party application was used to send voice data between participants in the meeting. However, if audio data was captured by and sent through the VMX application then a host of new features could be explored.

One such feature could be like that seen in Kinected Conference as described in Chapter 2. The feature in question tracked the length of time for which user talked during the meeting. Whenever the user was talking a timer appeared above their head in the video feed. A similar feature could be implemented in VMX, with the difference that the timer would appear above the head of the virtual avatar, not on the video feed from that user (i.e. not on their face texture).

Another simple feature could be to emphasize who was talking with some form of indicator. This could be something that appeared over a user's avatar's head when they were talking, or maybe their avatar could change colour; this feature was explicitly requested by two participants in the usability trial. Another way to emphasize who was talking would be to use 3D sound positioning within the virtual environment, this would mean that if someone in the meeting was talking and that person was sitting to the left of another participant in the meeting, that other participant would here the speaker’s voice as coming from their left (using stereo sound). This would be a natural way of indicating to a user what direction they should look in to see who was talking.

8.5 Large Conferences

This project only looked at small scale meetings between groups of less than ten people. In Chapter 2 the use of Second Life to hold large scale conferences was discussed. Further expansion of VMX to allow meetings of this scale would make it possible to investigate the value of Kinect control in that kind of situation.

The biggest likely hurdles that would be encountered in a large virtual conference are the technical problems of limited network bandwidth and limited CPU time for decoding face textures. Even the current small scale meetings in VMX are taxing on system resources. Some improvement to compression and encoding/decoding of textures would help, but there are other elements distinctive to large conferences that provide opportunities to reduce system resource requirements. Specifically, in a large conference it can be expected that at any given time, the majority of the participants would only be watching what is going on. There would probably be an individual or a small group doing a presentation in front of a large audience. There would be little need for the audience to be sending video of their faces to other conference participants. Instead, audience members could have simple 3D modelled heads. Those heads could still be oriented to show where each audience member was looking. Only those who were addressing the audience would transmit face textures. Kinect skeleton data does not use much data, so it would likely be possible to have the entire audience of a conference fully animated, just like an ordinary meeting. Another alternative would be to keep face textures for audience members, but greatly reduce the frequency at which they update.

8.6 Interactive Objects in the Virtual Environment

A final area of interest to future work on using Kinect in virtual meeting software is the integration of interactive objects into the virtual environment. This would allow users to collaboratively work with objects in the virtual world. There would also be potential applications in terms of making the experience of using the software more immersive. There would be a few hurdles in the implementation of such a system however.

An example of how this could be used would be a meeting of people planning an event like a concert and needing to decide how to lay out various amenities in the space they have. The participants in such a meeting could have simple models of various objects such as a stage, boundary fences, amenities etc. and they would all be able to interact with these models and arrange them on the virtual meeting table however they see fit. Another example could be a teamwork exercise where the participants of the meeting are tasked with cooperating to build a tower out of blocks as high as possible without it falling down.

It would be necessary to come up with a way for user to pick up objects in the virtual environment. If the virtual objects were subject to a physics simulation within the virtual environment, then users could simply be given the ability to knock objects around with their avatars. In this system they could potentially pick objects up by squeezing them between two hands. There are a few problems with this however. The first lies in the way physics simulations typically work when it comes to preventing objects from colliding with each other. When the simulation detects that two objects are colliding, it will apply a restoring force that works to push the objects apart (Dean, 2010). This power of this force tends to depend on the degree to which the objects overlap. If a user wasn't careful about how close together they brought objects there could be the potential for them to be launched out of their grasp as high speed. While amusing, this would not be very productive.

Another other potential problem lies in the accidental movement of objects in the virtual environment. With data from the Kinect having the exhibiting significant anomalies when the position of joints is poorly estimated, there is the chance that a user might inadvertently send an object flying.

For the reasons above it would probably be desirable to incorporate some form of mechanism for allowing users to only manipulate object in the environment when they demonstrate intent to do so, such as by closing and opening their hand in a grasping motion to grab and release objects. As discussed earlier in this document, there is no support within the Kinect SDK for detecting gestures, or for reporting the positions of individual fingers. Consequently, it would be necessary to develop an algorithm that could detect this kind of hand gesture directly from the Kinect's depth stream data.

Aside from allowing users to pick up objects it would also be desirable to provide them with a way to rotate these objects. Seemingly, the most natural way to do this would be to have the user rotate the object as they would in the real world, i.e. turn the objects by turning their hand while they were holding it. This would be subject to the same problem as above however, in that there is no way to detect this kind of motion using the default capabilities of the Kinect SDK.

The virtual nature of the objects that would be being manipulated opens an opportunity for object manipulation that does not exist in the real world. Users would be able to do things such as scale objects up and down (perhaps using a gesture where the user grabs the object between two hands and pulls them apart or pushes them together in order to make the object bigger or smaller respectively). There would also be the opportunity to do things such as duplicate already existing objects in the virtual space, or objects that represented scale models of real world things (such as buildings) could be given physical properties matching the full size object, opening the door for realistic simulations to be performed in the virtual environment.

As mentioned earlier, a potential use of virtual objects is to make the virtual meeting more immersive. To have the experience be more immersive means to make the way a user interacts with the software when performing a certain action in the virtual world seem more like the way they would perform the same action in real life. To give an example, say in a meeting someone wished to distribute a document like a memo or report to the other participants in the meeting. Instead of transmitting a digital copy of the document to the other participant via email or something similar, they could upload a copy into the virtual meeting program and it could be applied as a texture to a virtual object shaped like a piece of paper. The person who is distributing this document could then pass out this bit of paper, perhaps using the object duplication feature mentioned above to hand out copies of the document to the other participants in the meeting. The participants could then use a grasping gesture to pick the object up of the table and have their avatar look down so they could see the virtual document on their screen and read it. If the text on the paper was too small for a user to read clearly, they could use the ability to scale objects to make the paper bigger and the text on it easier to read.

One major consideration that would need to be made with regard to any virtual object manipulation using real world body movements revolves around matters of scale; the question of whether real world movements should translate into virtual movements on a 1:1 scale. To illustrate why this needs to be considered, imagine a situation where a user wanted to have their avatar pick up an object that was on the other side of the virtual table. Ideally the user would perform a gesture where they act as if they are really reaching over the table to grab the object, their avatar would do the same movement and the object would be picked up. But a problem arises if the user's real world surroundings don't permit such an action; for instance, what if the object is one metre in front of the user's avatar in the virtual environment, but in the real world the user has a wall eighty centimetres in front of them? It can even be that the same problem could stem not from there being an physical object in the way, but from the user needing to move outside the Kinect's effective range for skeletal tracking. In this case there would need to be some mechanism for allowing the user to pick up this object without performing a full one metre reach forward. There are a number of ways to approach this problem. One is to provide a way for users to perform motions that will be exaggerated by their avatars, thus allowing a user to do a small reach forward while having their avatar to a large reach forward. This approach would raise the question of whether the user's actions should always be exaggerated. If they were, this could make it difficult for user's to perform precise movements; and if they weren't there would need to be a mechanism for allowing the user to indicate when they wished for a movement to be exaggerated, and by how much.

Another interesting approach to the reach and grab problem would be to take advantage of the virtual nature of the environment to give the users' avatars 'telekinetic' powers. A user could look an object they want to manipulate in the virtual world, and perhaps perform a beckoning gesture with their hand to draw it closer. Similarly, a flick of the hand could send the object flying away. Other more pedestrian methods of solving this problem might involve forgoing gestures entirely and having users make use of the keyboard and mouse to select and manipulate out-of-reach objects.

Appendix I

This appendix lists the questions that were asked in the questionnaire given to participants of the usability trial that was carried out as a part of this project.

Section 1: Kinect Controls

Did you adjust the size of avatars' heads during the meeting?

If yes why did you choose/need to do it? Is there anything that you could think of to improve this ability?

Did you prefer using the automatic camera controls or the manual camera controls?

What were the reasons for your preference (if any)?

Did your preference of camera control (Automatic/Manual) change between when somebody was presenting at the front of the room and when everyone was sitting around the table?

If yes, was there a particular reason for this?

Do you have any other comments about the camera controls?

Were you sitting in a chair that can be swivelled to the left and right easily (e.g. an office chair)?

Section 2: Presenting

Only answer the questions in this section if at some point during the meeting you made a presentation in front of the display screen. If you did not, go to Section 3.

Did you utilise gestures to control the display screen?

Do you have any comments on the gesture controls (e.g. were there any particular difficulties? Would you have preferred a different form of control?)

Were you able to see the faces of your audience in the meeting clearly while presenting?

If no, what was the reason that prevented you from having a clear view of other participant’s faces?

Were you able to see the body language or facial expressions of the other participants (e.g. could you tell if they were looking at you or other participants, or performing actions like applauding etc.)?

If yes, what kinds of things did you see?
If no, what (if anything) could you see the other participants doing?

Section 3: Meeting

During the meeting were you able to tell where other people were looking in the virtual meeting room? Could you tell when they were looking at you?

Could you see who was speaking by looking at their avatar and face?

If so, what things could you see that made it clear who was speaking?

Was it possible to tell when a speaker was using body language and gesturing to other participants?

If yes, what kinds of actions did you see?
If no, what (if anything) could you see their avatar doing?

During the meeting, were you able to notice if other participants were not paying attention to the meeting and instead doing something else in the real world (e.g. reading, playing a game, browsing the internet etc.)?

If so what did you notice that made it apparent they were doing this?

Did you do any real world activities besides watching and participating in the meeting during the experiment?

Section 4: Other Kinds of Meetings

Have you ever participated in a meeting where everyone was not in the same location before, like a video conference, a teleconference, or a virtual meeting in a different piece of software (e.g. Second Life)?

If you have tried any of these, did you notice any advantages or disadvantages when compared to using the software in this experiment?

Have you been involved in face to face meetings in real life?

If so, how do the compare to using the software in this experiment?

Section 5: Final Questions

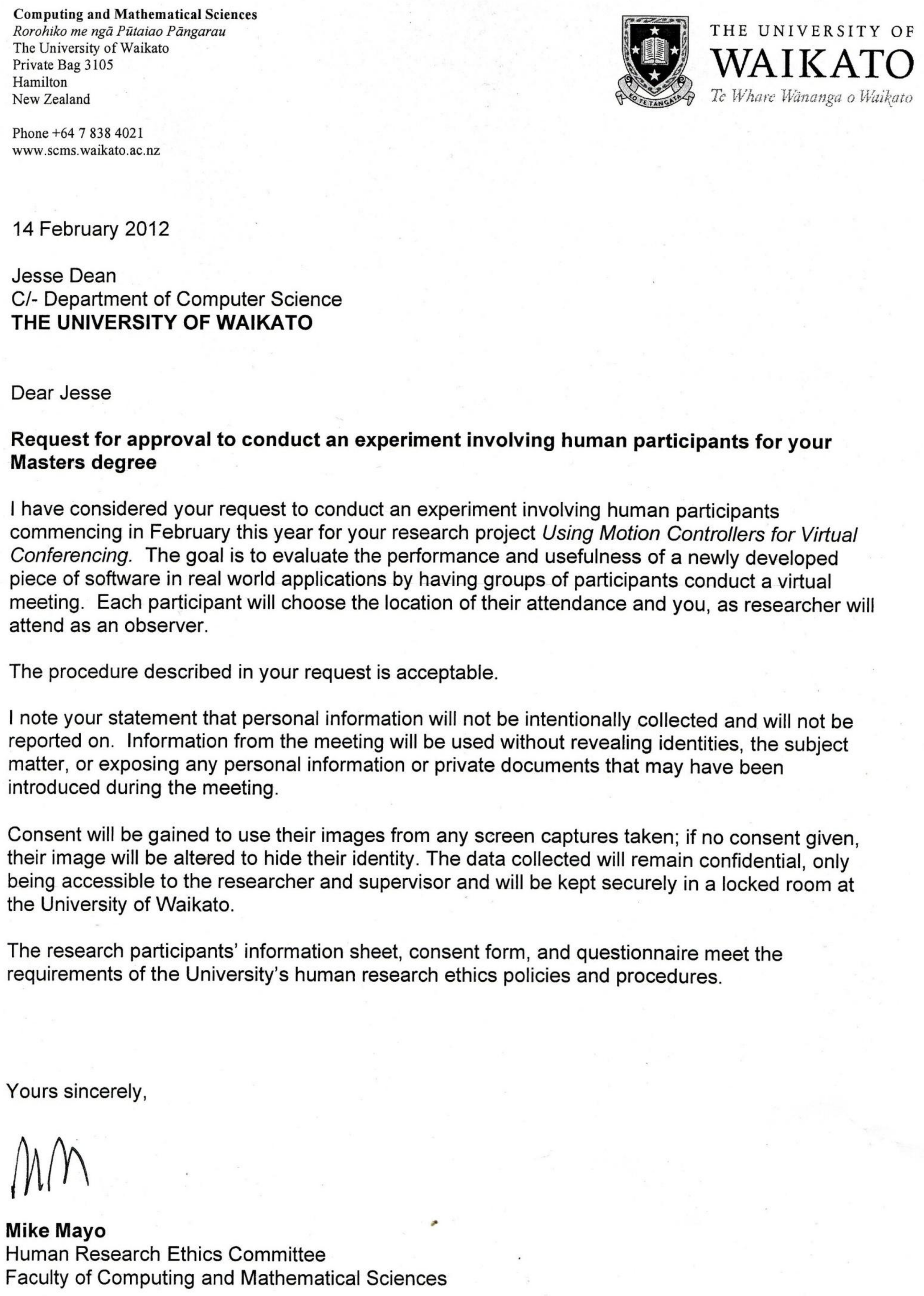
Do you have any other comments about how the software was to use (e.g. any difficulties, suggestions, silly or unhelpful controls etc.)

Would you be willing to be contacted by email by the researcher for additional clarification of your answers to this questionnaire if necessary?

If so, please provide your preferred contact email address:

Appendix II

A copy of the letter giving ethical consent for the usability trial that was conducted as a part of this project can be found on the following page



Computing and Mathematical Sciences
Rorohiko me ngā Pūtaiao Pāngarau
The University of Waikato
Private Bag 3105
Hamilton
New Zealand

Phone +64 7 838 4021
www.scms.waikato.ac.nz

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

14 February 2012

Jesse Dean
C/- Department of Computer Science
THE UNIVERSITY OF WAIKATO

Dear Jesse

Request for approval to conduct an experiment involving human participants for your Masters degree

I have considered your request to conduct an experiment involving human participants commencing in February this year for your research project *Using Motion Controllers for Virtual Conferencing.* The goal is to evaluate the performance and usefulness of a newly developed piece of software in real world applications by having groups of participants conduct a virtual meeting. Each participant will choose the location of their attendance and you, as researcher will attend as an observer.

The procedure described in your request is acceptable.

I note your statement that personal information will not be intentionally collected and will not be reported on. Information from the meeting will be used without revealing identities, the subject matter, or exposing any personal information or private documents that may have been introduced during the meeting.

Consent will be gained to use their images from any screen captures taken; if no consent given, their image will be altered to hide their identity. The data collected will remain confidential, only being accessible to the researcher and supervisor and will be kept securely in a locked room at the University of Waikato.

The research participants' information sheet, consent form, and questionnaire meet the requirements of the University's human research ethics policies and procedures.

Yours sincerely,

[illegible]

Mike Mayo
Human Research Ethics Committee
Faculty of Computing and Mathematical Sciences

References

Al Qahtani, S. H. (2010). *Virtual Worlds as Meeting Places*. Hamilton: The University of Waikato.

Ashby, A. (2009, 02 27). *IBM Saves $320,000 With Second Life Meeting*. Retrieved 02 15, 2012, from Engage Digital: http://www.engagedigital.com/blog/2009/02/27/ibm-saves-320000-with-second-life-meeting/

Cangeloso, S. (2012, 02 01). *Kinect for Windows SDK is released, gesture apps on the way*. Retrieved 02 16, 2012, from Geek.com: http://www.pcworld.com/article/230445/kinect_for_windows_sdk_beta_available_now_to_download.html

Clayton, S. (2011, 11 04). *Technet*. Retrieved 02 16, 2012, from Beta 2 of Kinect for Windows SDK released: http://blogs.technet.com/b/next/archive/2011/11/04/beta-2-of-kinect-for-windows-sdk-released.aspx

Dean, J. (2010). *Live Construction in Computer Games*. Hamilton: University of Waikato: Department of Computer Science.

DeVincenzi, A., Yao, L., Ishii, H., & Raskar, R. (n.d.). *Kinected Conference | MIT Media Lab*. Retrieved 01 31, 2012, from http://kinectedconference.media.mit.edu/

DeVincenzi, A., Yao, L., Ishii, H., & Raskar, R. (2011). Kinected conference: augmenting video imaging with calibrated depth and audio. New York: ACM.

Foxlin, E., Harrington, M., & Pfeifer, G. (1998). Constellation: a wide-range wireless motion-tracking system for augmented reality and virtual set applications. *SIGGRAPH '98*. New York: ACM.

Gohring, N. (2010, 07 30). *Mundie: Microsoft's Research Depth Enabled Kinect*. Retrieved 02 14, 2012, from http://www.pcworld.com/businesscenter/article/202184/mundie_microsofts_research_depth_enabled_kinect.html

Grootjans, R. (2009). *XNA 3.0 Game Programming Recipes: A Problem-Solution Approach.* Apress.

Hinchman, W. (2011, 06 20). *Kinect for Windows SDK beta vs. OpenNI*. Retrieved 02 14, 2012, from http://labs.vectorform.com/2011/06/windows-kinect-sdk-vs-openni-2/

Joystiq. (2010, 12 10). *PrimeSense releases open source drivers, middleware that work with Kinect*. Retrieved 02 14, 2012, from http://www.joystiq.com/2010/12/10/primesense-releases-open-source-drivers-middleware-for-kinect/

Li, C. (2009). *HLSL Introduction*. Retrieved 02 16, 2012, from Neatware: http://www.neatware.com/lbstudio/web/hlsl.html

Linden Labs. (n.d.). *Downloads | Second Life*. Retrieved 02 15, 2012, from Second Life: http://secondlife.com/support/downloads/

Linden Labs. (2003, 6 23). Second Life. San Francisco, California, USA.

Microsoft. (n.d.). *Getting Started with XNA Game Studio Development*. Retrieved 02 15, 2012, from MSDN: http://msdn.microsoft.com/en-us/library/bb203894.aspx

Microsoft. (2011). Kinect for Windows SDK Documentation.

Microsoft Kinect SDK vs PrimeSense OpenNI. (n.d.). Retrieved 02 14, 2012, from http://www.brekel.com/?page_id=671

Microsoft. (2011). *Kinect Sensor.* Retrieved 01 10, 2012, from MSDN: http://msdn.microsoft.com/en-us/library/hh438998.aspx

Microsoft. (2012, 01 31). *Microsoft Kinect For Windows SDK - V1.0 Release Notes*. Retrieved 03 12, 2012, from Microsoft.com: http://www.microsoft.com/en-us/kinectforwindows/develop/release-notes.aspx

Microsoft. (2004, 3 24). *Microsoft: Next Generation of Games Starts With XNA.* Retrieved 02 15, 2012, from Microsoft.com: https://www.microsoft.com/presspass/press/2004/mar04/03-24xnalaunchpr.mspx

Microsoft. (2009, June). *Project Natal 101.* Retrieved 01 18, 2012, from Microsoft: http://download.microsoft.com/download/A/4/A/A4A457B3-DF5D-4BF2-AD4E-963454BA0BCC/ProjectNatalFactSheetMay09.zip

Microsoft. (2011, 11). *Readme for Kinect for Windows SDK - Beta 2 release.* Retrieved 03 12, 2012, from Microsoft.com: http://www.microsoft.com/en-us/kinectforwindows/develop/readme.htm

Microsoft. (2010). XNA Game Studio Documantation.

Murray, D., & Basu, A. (1994). Motion Tracking with an Active Camera. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 16 Issue 5* , 449-459.

Nykamp, D. Q. (n.d.). *The cross product*. Retrieved 02 15, 2012, from Math Insight: http://mathinsight.org/cross_product

Ogre. (2012, 1 18). *Free Resources*. Retrieved 2 14, 2012, from http://www.ogre3d.org/tikiwiki/Free+Resources

Ogre. (n.d.). *Licensing*. Retrieved 14 02, 2012, from http://www.ogre3d.org/licensing

Oikonomidis, I., Kyriazis, N., & Antonis, A. A. (2011). *Efficient Model-based 3D Tracking of Hand Articulations using Kinect.*

OpenNI. (2011). *OpenNI/SampleAppSinbad.* Retrieved 02 14, 2012, from GitHub: https://github.com/OpenNI/SampleAppSinbad

OpenNI.org. (2010, 12 21). *PrimeSense™ Establishes the OpenNI™ Standard and Developers' Initiative to Bring the World of Natural Interaction™ to Life*. Retrieved 02 14, 2012, from http://www.openni.org/News/PrimeSenseEstablishestheOpenNIStandardandD.aspx

Peckham, M. (2011, 06 16). *Kinect for Windows SDK Beta Available Now to Download*. Retrieved 02 16, 2012, from PCWorld: http://www.pcworld.com/article/230445/kinect_for_windows_sdk_beta_available_now_to_download.html

Plunkett, L. (2008, 09 19). *So How Many People Actually PLAY Second Life?* Retrieved 02 18, 2012, from Kotaku: http://kotaku.com/5052067/so-how-many-people-actually-play-second-life

PrimeSense. (2011). *PrimeSense Natural Interaction*. Retrieved 02 14, 2012, from http://www.primesense.com/technology/nite3

Schroeder, R. (2002). *The Social Life of Avatars.* London: Springer.

Sidenbladh, H., Black, M., & Fleet, D. (2000). Stochastic Tracking of 3D Human Figures Using 2D Image Motion. *Computer Vision - ECCV 2000* (pp. 702-718). Springer.

Stott, L. (2011, 11 04). *Kinect SDK Beta 2 Release*. Retrieved 02 16, 2012, from MSDN: http://blogs.msdn.com/b/uk_faculty_connection/archive/2011/11/04/kinect-sdk-beta-2-release.aspx

University of California. (2011, 08 01). *Researchers turn Kinect game into a 3D scanner*. Retrieved 03 06, 2012, from Physorg.com: http://www.physorg.com/news/2011-08-kinect-game-3d-scanner.html

Valve Software. (2007). *Source - Programming*. Retrieved 02 14, 2012, from http://source.valvesoftware.com/programming.php

Zafrulla, Z., Brashear, H., Starner, T., Hamilton, H., & Presti, P. (2001). American sign language recognition with the kinect. *ICMI '11 Proceedings of the 13th international conference on multimodal interfaces* (pp. 279-286). New York: ACM.

Zhu, R., & Zhou, Z. (2004). A Real-Time Articulated Human Motion Tracking Using Tri-Axis Inertial/Magnetic Sensors Package. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* , 295-302.

Zhu, Y., Dariush, B., & Fujimura, K. (2008). Controlled human pose estimation from depth image streams. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08.* (pp. 1-8). Anchorage, AK: IEEE.