# Self-Avatars and IR-based Position Tracking in Virtual Environments using Microsoft Kinects

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Srivishnu Kaushik Satyavolu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Peter J Willemsen

June 2012

© Srivishnu Kaushik Satyavolu 2012

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Peter J Willemsen, whose constant motivation, guidance, moral suppport and help are the reasons that this work is a reality. I would also like to thank him for being a great teacher and a mentor and especially for keeping me focused towards the *big picture* right from the beginning of this work.

Next, I would like to thank Dr. Gerd Bruder from University of Wrzburg, Germany for his idea, motivation and all the support in designing the multiple Kinect based IR position tracking system. I would also like to thank his advisor Dr. Frank Steinicke for his support and invaluable feedback on this work.

Next, I would like to thank my lab mates and Subbu for all the moral support and encouragement they have been.

A special thanks to Prof. Murthy Ganapathibhotla, without whom I wouldn't be pursuing my Masters degree.

Also, I would like to thank the excellent support and funding provided by The National Science Foundation, without which the work wouldn't have been a reality. In particular, this work is supported under Grant No. 0828206.

And, I would like to thank all my teachers, staff, friends and wellwishers for all their support and encouragement.

Last but not least, I would like to thank my family again for all the support and encouragement ever since I can remember.

## Dedication

I would like to dedicate this thesis to my mom & dad without whose unconditional love and support none of this would have been possible. I would also like to dedicate this to my late grandmother who always believed in me and my abilities.

Abstract

The purpose of this work is to develop a means for providing users with real time visual body feedback when interacting in Virtual Environments using Microsoft Kinect sensors. The advent of the Microsoft Kinect provided the research community with an inexpensive but versatile piece of equipment that can be used to obtain real time 3D information of the physical world. Our setup uses multiple Microsoft Kinects to capture a 3D point cloud of a participant in the Virtual Environment, constructs a 3D mesh out of the point cloud, and reprojects the 3D data into a Virtual Environment so that a user can experience his/her own body moving when looking through an immersive display device. Previous studies have indicated that experiencing a self avatar in a Virtual Environment enhanced user presence. In other words, users felt the Virtual Environments to be more realistic in a variety of measures. This work also proposes an InfraRed-based position tracking system using multiple Kinects for tracking user's head position when moving in Virtual Environments. A pilot study was conducted to analyze the effects of IR interference among multiple Kinects on the overall performance of the tracking system. The results show that our self avatar representation combined with the proposed tracking system should help support natural interaction of users in 3D spaces.

# Contents

# List of Figures

# 1 Introduction

A virtual environment (VE) is typically a 3D model or simulation rendered generally on stereoscopic displays such as Head Mounted Displays (HMDs) that are worn by users. These HMDs are equipped with position and orientation trackers that allow a user to look into the VE based on his/her real world head position and orientations. However, the user does not experience any visual body feedback in the VE (for example, being able to look at one's own hands and feet in the VE while looking down in the real world) because the user's real body does not exist in the rendered 3D model.

The holy grail of virtual reality (VR) applications has always been to make the users perceive virtual environments (VEs) as real environments and interact naturally as one would if the VEs were truly *real*. Though there might be numerous features that need to be incorporated into a VE to fully achieve this, visual body feedback is considered a major contributor to the user's level of presence in a VE. Previous studies have indicated that the visual body feedback in VEs has great potential in enhancing the user's sense of presence in computer generated virtual scenes [6], in causing more natural interaction with virtual content [45], and in improving ego-centric distance judgments [32]. This thesis focusses on using the visual body feedback for enhancing user's level of presence in a VE.

The first way that comes to mind when talking about visual body feedback is to mount a camera on the user's head, capture the video of his/her motions and process each 2D image/frame that is captured to separate or segment out user pixels (the pixels that make up user's hands and feet, etc.) from the rest of the background. This user information can then be merged onto the VE images rendered on the HMD

to provide a user his/her own visual body feedback. This process of separating foreground user pixels from the rest of the background is often referred to as *segmentation* or *background subtraction*. This type of image-based user segmentation is trivial if the background is static, but becomes quite complex if the background is in motion relative to the user's head (see [42] for a review). In the context of a HMD-based VE, the camera needs to be mounted on the user's head along with the HMD. As a result, the background is no longer static relative to the user's head. Therefore, static background subtraction techniques cannot be directly applied to separate user pixels. One way to overcome this would be to employ skin-color-based segmentation techniques (as in [6]) that separate pixels corresponding to the user based on user's skin color. However, this method is not at all scalable due to the large variation of skin color from user to user and varying lighting conditions.

Another way to provide visual body feedback is to load a human-like avatar model into the VE and change its position in the VE based on the user's real world position. The issue then would be to replicate user's physical motion in the VE. The solution for this is to animate the 3D avatar model by means of full body tracking of the user. Full body tracking is usually achieved by means of highly expensive full body motion capture suits or at the cost of using multiple IR markers attached to each joint that needs to be tracked (as in [5]). Even with that, the 3D avatar does not necessarily resemble the user. In order to distinguish the 3D model-based self avatars from the actual 3D user representations, this thesis uses *true* self representation to refer to the actual 3D user representations. Although there is no study done to directly suggest that *true* self representation of the user gives better sense of presence than a random 3D avatar, using a *true* self representation of the user can be important especially when working with social virtual environments. An alternative method to get a possibly better *true* representation of the user is to capture the *true* 3D representation of the user, reproject it into the VE and update it in real time based

on user's physical movements.

The introduction of the Microsoft Kinect sensor for Xbox 360 has inspired much interest across the virtual reality (VR) community, primarily due to the rich array of sensors it has for capturing 3D scene information, and its very low cost of approximately U.S.$150. The Kinect has already shown its applicability as a low-cost means of full body tracking for gaming applications that require full body tracking for animating avatar models. Microsoft Kinects are 3D sensors that acquire depth data (in addition to the color data from the RGB camera) by projecting an IR (InfraRed) pattern into the scene and then estimating the depth of the objects in the scene based on the captured IR image (see Chapter 2 for more information on Kinects). This thesis proposes that multiple Kinects can be used for providing *true* visual body feedback for users in a VE. As a single Kinect gets only partial view of the scene (it can be referred to as 2.5D information [20]), in order to get a 360° (3D) representation of the scene, a multiple Kinect setup has been proposed in this thesis.

Another pertinent issue with VEs is that the position tracking system used for tracking a user's head position is quite expensive and requires large multi-camera setups that are also extremely difficult to relocate. For example, the WorldViz PPTH/X tracking system [38] is one such position tracking system consisting of 8 optical cameras that communicate with one another over a physical network using a protocol called VRPN and use IR based stereo triangulation to track a marker (typically attached to a HMD). Though this system is more robust and accurate, it is highly expensive, costing roughly around $25k just for the multi-camera setup.

For this purpose, an IR-based (Infrared) position tracking system using multiple Kinects is proposed as a low-cost alternative for tracking user head position in VR applications where high tracking accuracies are not essential. This document also addresses the other issues that arise when using multiple Kinects (such as IR interference) and how much effect they can have on the proposed visual body feedback and

the position tracking approaches. To this order, a pilot study has been conducted to analyze the effect of interference on the multiple Kinect based position tracking system.

The rest of the document discusses in detail how each of the visual body feedback approaches and the position tracking systems are implemented. Chapter 2 describes the complete background information that is necessary to understand the implementation details discussed in the consequent chapters. It also consists of a Related work section 2.2 that discusses the related work that has been done in the areas of visual body feedback and position tracking. Chapters 3 and 4 give the implementation details of the visual body feedback approach and the position tracking systems, respectively. They also explain in different sections, the details of the issues that had to be overcome, the final results, the limitations and the future scope of the corresponding systems. Chapter 5 describes in detail about the software implementation and introduces *libkinect* API, the library that has been used for interfacing with Kinects. Chapter 6 concludes the document with a summary of the proposed approaches, results and the future scope. Finally, Appendix A contains miscellaneous implementation details that do not directly fit into the original parts of the document.

# 2 Background

This chapter describes in detail all the background information that is necessary to follow the rest of the paper. Section 2.1 discusses in detail everything about Kinect, its parts, how it works, why is it used, need for calibrating it and issues with using multiple Kinects. Section 2.2 discusses the related research that has been done on Kinect.

## 2.1 Background Information

### 2.1.1 What is Kinect?

Microsoft Kinect [2, 1] is a hands-free gesture based motion sensor released by Microsoft for the Xbox 360 game console and later for the Windows Operating Systems. Figure 2.1 shows the Kinect and its major parts namely, a RGB (Red-Green-Blue) Camera, an IR (Infrared) projector, an IR Camera, a motorized tilt and an array of microphones. The motorized tilt allows the Kinect to pan and tilt vertically about its base. The array of microphones are capable of locating the voice and also cancelling out any ambient noise in a room.

The RGB camera of the Kinect is just a color camera equipped with a Bayer color filter [2] that is capable of capturing 1280×1024 resolution images at 15 Hz or 640×480 resolution images (see Figure 2.2(a)) at 30 Hz. The IR camera is a monochrome CMOS sensor [2] that can capture IR video data under any lighting conditions. Just like the RGB camera, the IR camera is also capable of capturing 1280×1024 resolution images at 15 Hz or 640×480 resolution images (see Figure 2.2(b)) at 30 Hz. The IR

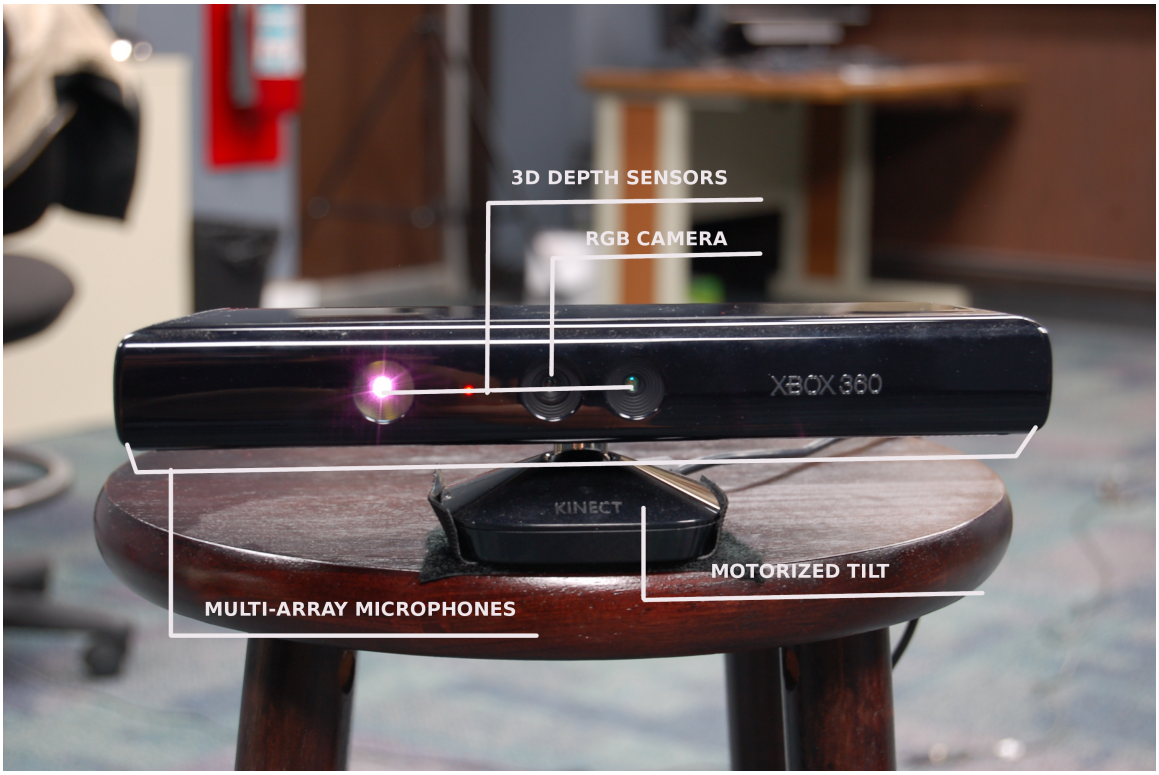
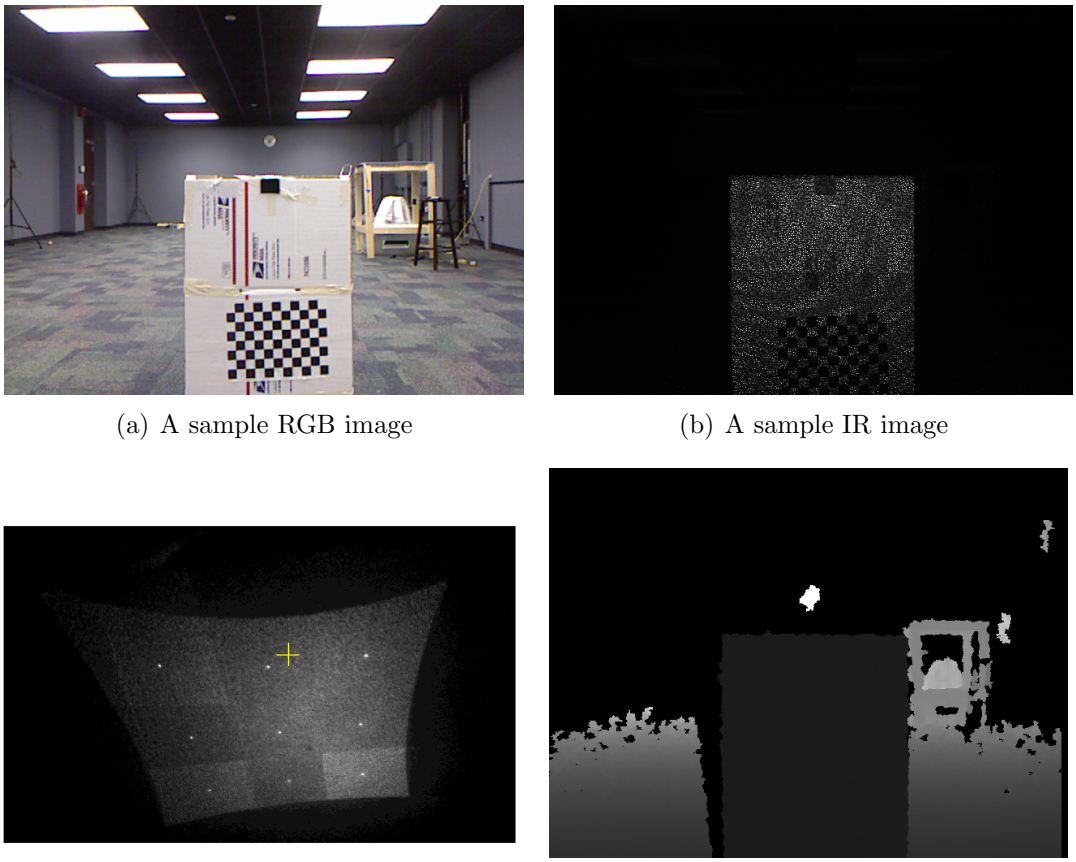



Figure 2.1: Figure shows Xbox 360 Kinect and its components, namely; a RGB camera, depth sensors, multi-array microphones and a motorized tilt.

projector is a IR laser projector that projects patterned IR light (see Figure 2.2(c)) into the scene. The power of the Kinect comes from the IR projector/sensor pair (see Figure 2.1) which, when combined, acts as a single 3D depth sensor that is capable of capturing 640×480 resolution depth images(see Figure 2.2(d)) at 30 Hz. The mechanism of the depth sensor (also known as the depth camera) is described in the following subsection.

### Depth Camera/Sensor

Kinect's depth sensor is driven by a microchip designed by PrimeSense. It uses a technique called *Light Coding* for obtaining the 3D depth information of the scene in its field of view [2]. Basically, the Kinect IR projector projects an invisible IR light pattern (or code) made up of dots (see Figure 2.2(c)) of varying light intensities at



(a) A sample RGB image

(b) A sample IR image

(c) A sample image showing the projected IR pattern

(d) A sample depth image

Figure 2.2: Figure shows 640×480 resolution a) RGB, b) IR and d) depth images obtained from the Kinect. c) shows a sample IR pattern projected into the scene by the IR projector.

30 Hz. The IR camera then captures these IR patterns every frame as IR images (see Figure 2.2(b)) and compares the captured IR pattern of dots with the projected IR pattern. The IR pattern that the Kinect projects is actually hardwired into the Kinect.

In general, the Kinect's hardware compares the distance between any two consecutive IR dots with their actual distance in the projected pattern to estimate the actual depth at the corresponding pixel position. This depth estimate is represented as a 11-bit depth disparity value in the range of [0, 2047) in the depth image. This process is done for each pair of captured dots to obtain depth values at that pixel

position. The pixels for which the captured information is insufficient to estimate a depth value are given an invalid depth value of 2047. The result is a depth image as shown in Figure 2.2(d), which is color-coded based on mapping the actual depth range [0, 2047] to the inverted grayscale color range [0, 255] such that a 2047 (invalid depth value in the depth image) corresponds to a black (0 grayscale) color in the depth image.

**A few applications of Kinect**

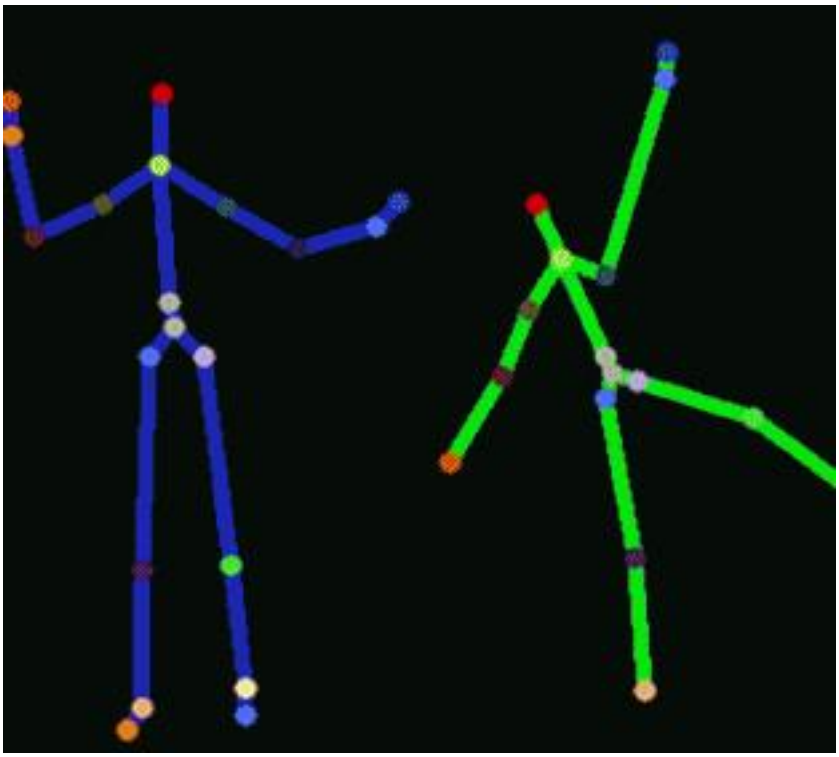


Figure 2.3: Skeleton tracking using Microsoft SDK for Kinect showing all the 20 joints of a user

Right from its launch as a novel low-cost motion-sensing input device [2], Kinect has seen wide-spread adoption well outside the traditional game console market. It has been used in numerous projects integrated with either the Microsoft-provided Kinect SDK [11], Primesense's OpenNI SDK [35], or through open-source PC drivers that provide access to the RGB camera and depth-related information. Using this

information, the existing software technology [11, 35, 13] can also be used to track the motion of up to six users by tracking positions and orientations of up to 20 joints per user. See Figure 2.3 see Appendix A.3.1 for a full list of skeletal joints that can be tracked using a Kinect. Tracking these joint positions across time enables advanced gesture recognition, facial recognition and voice recognition.

### 2.1.2 What is Camera Calibration?

Like any hardware device, the images captured by a camera are also susceptible to errors/noise that arise due to several hardware factors such as variations in magnitude and phase in the measurement setup, etc. Camera calibration is the systematic process of removing those errors in the estimated image values based on an error model. The error model consists of a set of parameters (typically called *intrinsic* parameters) that vary from device to device. *Intrinsic* parameters of a 2D camera constitute the x,y focal lengths (*fx,fy*) of the camera and the principal point (*cx,cy*) of the captured images [10]. the Calibration is in essence the process of estimating these parameters by capturing a series of known patterns. See [33] for more details on camera calibration and the error model. Thus, any camera needs to be calibrated for obtaining more accurate results.

Camera calibration can also refer to the process of aligning one or more cameras into the same space, aka *extrinsic* calibration. This is necessary when more than one camera is viewing the same scene from different positions and orientations and the values estimated by each camera need to be transformed into a global coordinate system. In general, if multiple cameras are viewing the same object, and the local position and orientations of the object with respect to each camera are known, then it is possible to determine the rotation (R) and translation (T) transformations that need to be applied on one camera to get to the coordinate system of another camera. Alternatively, if the global positions of the object are known, the local position esti-

mates by each camera can be used to determine the global position and orientation of the camera. The process of determining the parameters forming these transformations (known as *extrinsic* parameters) is referred to as *extrinsic* calibration, or *stereo* calibration. Thus, camera calibration in general may refer to both *intrinsic* and *extrinsic* calibrations.

**Why does Kinect need calibration?**

As the Kinect (see Figure 2.1) contains RGB, IR and Depth cameras, each of these needs to be calibrated before being used (for the reasons described in the above section 2.1.2). Also, determining the color of a depth pixel involves determining the corresponding pixel in the RGB image. As the RGB and the depth cameras are physically separated (again see Figure 2.1), they need to be calibrated into same space before being able to map depth image pixels to the corresponding RGB image pixels. Similarly, using multiple Kinects requires them to be calibrated into the same space in order to get the advantage of using all of them. The next section describes the advantages of using multiple Kinects and the potential issues that might arise when using them.

### 2.1.3 Necessity for multiple Kinects

Multiple Kinects are useful when the 360° tracking of a scene is required. This is because a single Kinect offers information only about the points of an object that are in its field of view (also referred to as 2.5D information in [20]). Therefore, to get a full 360° representation of the object in the scene, a multiple Kinect setup with each Kinect viewing the objects in the scene from a different position and orientation can be used. For position tracking purposes, having 3D coverage would guarantee that the marker be within the view of at least one Kinect at any point of time. The Kinects will typically be arranged in some sort of a circular fashion such that each

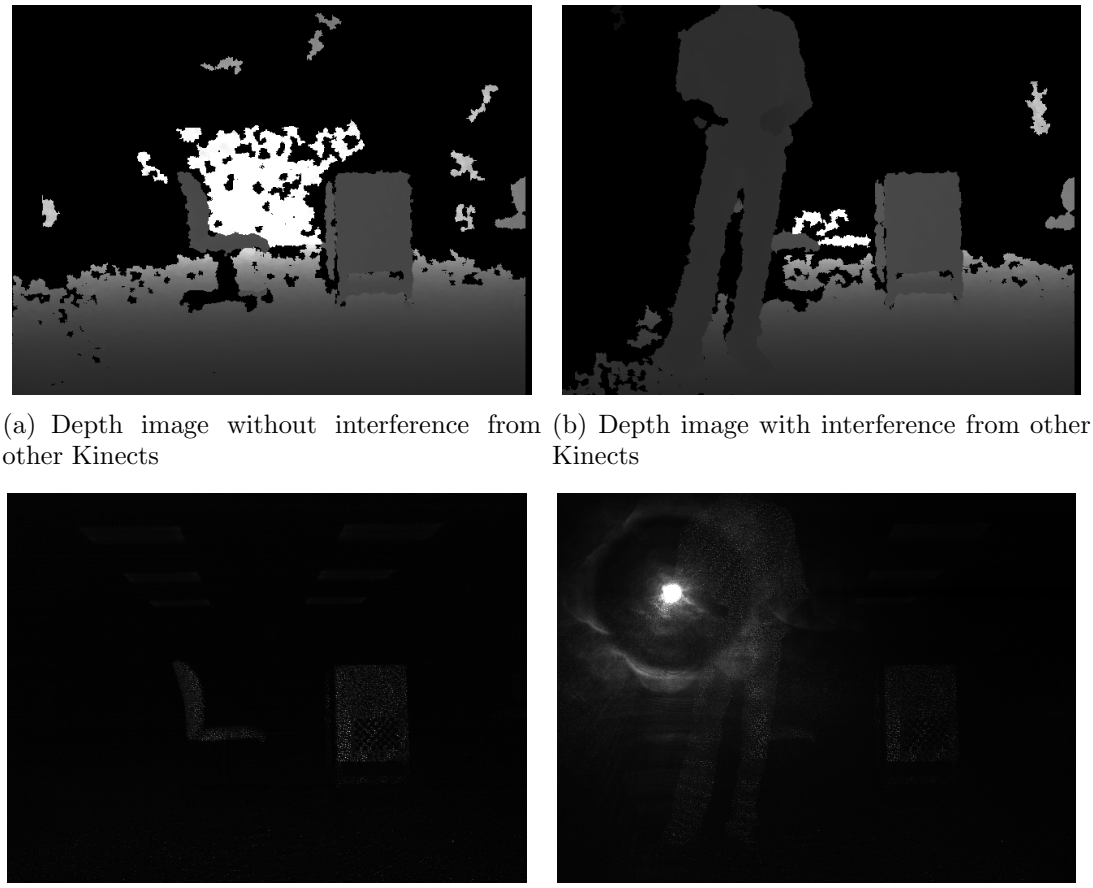
Kinect contributes to a portion of the full 360° representation of the objects in the scene. The following section offers detailed explanation of the various advantages and disadvantages a multiple Kinect setup can offer.

### Issues with multiple Kinects

The major limitation of using multiple Kinects though is the interference of the IR lights projected from each of the Kinects in the regions with overlapping Kinect views. Yet another disadvantage is that the huge bandwidth requirements of the Kinect allow at most one Kinect to be connected to a single USB controller. So, in order to connect N Kinects to the same system, one would require N different USB controllers present on the same machine. Then, it becomes the responsibility of the system hardware resources to switch between each Kinect stream while not increasing the latency. Alternatively, multiple Kinects can be connected over the network to operate in a collaborative fashion (see Section 4.2.3 for details). The disadvantage of this approach is that huge network bandwidths are required for collaborating the large image frames acquired by each Kinect at 30 fps.

### What exactly is interference?

*Interference* occurs when the IR dot patterns from one Kinect falling on an object intersperse with the IR dot patterns from another Kinect, confusing both the depth cameras and thus resulting in more invalid depth values in both sets of depth images. Figure 2.4 demonstrates the effects of interference on captured depth images. The effects of interference are further analyzed and discussed in detail in both the visual body approach (see Chapter 3 and the position tracking system implementation (see Chapter 4) chapters.



(a) Depth image without interference from other Kinects

(b) Depth image with interference from other Kinects

(c) IR image without interference from other Kinects

(d) IR image with interference from other Kinects

Figure 2.4: Example figures show depth/IR images obtained by a Kinect with and without interference from another Kinect. a) shows the color-coded depth image in the absence of interference. A user brings another Kinect into the view, and the resultant interference causes b) to get many invalid (black) depth pixels (see the cart, for instance). c) shows another IR image without interference and d) shows the bright IR light emerging from the other Kinect in an otherwise IR-free scene.

## 2.2 Related Work

There are a few previous studies that analyzed the effects of visual body feedback on user's perception of HMD-based VEs. The studies conducted by Mohler et al. [5, 32] suggest that providing visual body feedback in the form of an animated self-avatar model can improve distance perception in VEs. Another study conducted by Bruder et al. [6] shows that providing visual body feedback as camera-captured realistic self-avatars can also improve user's sense of presence in VEs. These two results support the

claim that visual body feedback improves user perception of VEs. However, Streuber et al. [45] found that a brief exposure to self-avatars does not have significant impact on a variety of tasks such as locomotion, object interaction and social interaction. However, it can be argued that this is because the visual body feedback was not available during the actual tasks. Garau et al. [16] studied the effect of avatar realism and eye gaze in the perceived quality of communication in immersive VEs. They suggested that in general the avatar-realism did not have a significant difference on the conducted study. But the avatars they tested were 3D models rather than the *true* representations of the involved users. Based on these previous studies we believe that visual body feedback can improve the user perception in VEs, provided that the right type of visual body feedback is used for the task at hand. Especially in an immersive social VE, we claim that using a *true* self representation will make communication between various users natural. Although, there is no study directly supporting this claim, the goal of this is to provide such a *true* visual body feedback using reconstructed 3D self representations of users that are captured by multiple Kinects.

Different approaches have been proposed to generate 3D representations using the Kinect hardware [3]. For example, Izadi et al. [20] have demonstrated that integrating 2.5D information from a moving Kinect can be used to generate a robust virtual 3D reconstruction of physical objects in the scene with the help of standard image registration techniques. This approach has the objects in the scene relatively stable, while the Kinect is in motion. In contrast, our implementation has the user in motion with the Kinects stable in a precalibrated setup. The same techniques used in the above cited paper can be used for generating a better 3D representation of the user in the VE.

Kinects have been used for a wide variety of purposes, including motion sensing and skeletal tracking [4, 11, 35, 13], implementation of real time virtual fixtures [14],

3D environment modeling [17], object interaction and manipulation [28] and even for in-home fall risk assessment for older people [44]. Most approaches in the field of VR try to reconstruct physical objects in real time by integrating the data provided by multiple Kinects [4, 41, 5]. While in general this appears to be a viable solution, it is still not fully understood how Kinects can be placed in a VR lab for accurate 3D tracking of users without interference between Kinects. Although the Kinect appears to be reasonably robust to interferences caused by ambient IR light, researchers noted potential problems caused by overlapping structured IR light from multiple Kinects [4, 41]. Due to the emitted IR light pattern, it is possible that one Kinect can affect depth measurements of other Kinects in a shared tracking space. Maimone et al. [31] proposed an approach to mitigate the effects of interference with the help of sensor motion. Though it seems to be a promising solution to the interference problem, their study did not test the effects of sensor motion on the resultant 3D data. Schroder et al. [41] suggested another way of reducing interference by use of rotating disks for creating a time division multiple access of the sensors in the scene. However, this approach requires additional hardware modifications which might not be feasible every time.

Several studies [43, 22, 23] have been conducted to analyze the accuracy of Kinect depth data and effects of resolution on it. However, none of these were carried out in the presence of interference from other Kinects. Hence, a pilot study has been conducted in this thesis to analyze the effects of interference on the proposed position tracking system and thus the Kinect's tracking abilities.

Lastly, though many methods were suggested for calibrating the Kinects [7, 24, 18], none of them are fully automated. The current thesis uses a portion of Oliver Kreylos' [24] calibration method and builds a semi-automated extrinsic calibration method that we believe is an easier, scalable method than existing methods.

# 3 Real Time Visual Body Feedback using Multiple Kinects

This chapter describes in detail how visual body feedback can be provided to the users by merging information from multiple Kinects. The idea is to capture the 3D point cloud of a scene using multiple Kinects (see Section 2.1.3), apply segmentation techniques to extract the points corresponding just to the user, construct a 3D triangle mesh out of the point cloud and then project the point cloud into the VE so that the user wearing a HMD can experience visual body feedback while looking into the VE.

For the proposed visual body feedback approach, precise and robust head tracking is necessary to get the user a stable visual body feedback in realtime. Hence, the current implementation uses the WorldViz PPTH/X tracking system [38] for tracking the position and the InterSense InertiaCube [19] for tracking orientation of user's head. WorldViz PPTH/X tracks the position marker attached to user's HMD, acquires the orientation data from the InterSense InertiaCube which is also attached to the user's HMD. The obtained position and orientation data are then streamed over a network by the WorldViz PPTH/X tracking system using VRPN protocol [39]. Together, WorldViz PPTH/X and InterSense InertiaCube constitute a robust and a precise head tracking system and hence are chosen for the current approach. Alternatively, the proposed Kinect-based IR tracking system (described in Chapter 4) can also be used instead of WorldViz PPTH/X tracking system for user head position tracking. Algorithm 1 lists the steps that need to be followed in order to achieve visual body feedback.

**Algorithm 1** Steps for acquiring visual body feedback

```
// Step - 1: Intrinsic Calibration
for each Kinect K do
    Determine the intrinsic parameters of RGB, IR and Depth cameras
    Determine the stereo transformation that needs to be applied between RGB and
    the depth cameras
end for
//Step - 2: Extrinsic Calibration
for each Kinect K do
    Determine the extrinsic parameters necessary to transform the current Kinect's
    coordinate system to the global coordinate system used by WorldViz PPTH/X
    tracking system
end for
for each new time frame T do
    // Step - 3: 3D Reconstruction of User
    for each Kinect K do
        Acquire the raw RGB, and depth images
        // Step - 4 Segmentation of User pixels
        Apply depth based segmentation techniques
        for each valid pixel p in the depth image do
            // Step - 5
            Reconstruct the corresponding 3D position
            Using the stereo transformation between RGB and the depth images, find
            the corresponding RGB pixel
        end for
        // Step - 6
        Generate 3D Triangle Mesh for the extracted/segmented User Point Cloud
        Using extrinsic parameters of the current Kinect, transform the 3D mesh into
        the global space used by the WorldViz PPTH/X tracking system
        Apply face culling on the resultant point cloud to enable view-dependent
        Kinect priority for the current Kinect when rendering this view of the global
        user mesh
    end for
    // Provide visual body feedback for a user wearing the HMD
end for
```

The following sections describe each of the above steps in detail. Sections 3.1 and 3.2 discuss the process of calibrating the Kinects. Section 3.3 describes the process of reconstructing/representing user in the VE as a 3D triangle mesh obtained by merging the point clouds captured by all the Kinects.

## 3.1 Intrinsic Calibration of the Kinects

This Section describes ***Step - 1*** of Algorithm 1. To acquire the intrinsic parameters of the RGB/Depth/IR cameras of the Kinect described in Section 2.1.2, Nicholas Burrus' [7] RGBDemoV0.4 [9] is used. RGBDemoV0.4 uses standard chessboard recognition techniques [10] for extracting the intrinsic parameters for each of the RGB/IR/depth cameras of the Kinect. Section 3.1.1 describes in general what a chessboard recognition mechanism is and how it helps for calculating the intrinsic parameters of a camera.

### 3.1.1 Chessboard Recognition

The concept is quite simple. A known pattern comprised of points separated at fixed distances is captured by the camera, and the captured image is used for determining the positions of these points. The captured positions and the known positions can then be used together to estimate the intrinsic parameters that form the error model [10].

A chessboard is an excellent candidate for the pattern discussed above because it has fixed number of corners separated at fixed distances that depend only on the size of the chessboard. Thus, in a standard chessboard based calibration method, a chessboard like the one shown in Figure 3.1 is used for determining the positions of its corners (typically internal corners as shown in Figure 3.2) with the help of OpenCV's [10] edge detection methods such as Hough line detection, Canny edge detection, etc. These positions are then used for determining the intrinsic parameters of the camera.

As described in OpenCV's Wiki page [10], the following is an exhaustive list of the intrinsic parameters that are calculated for each camera.

`fx, fy -focal lengths of the camera in pixel-related units`

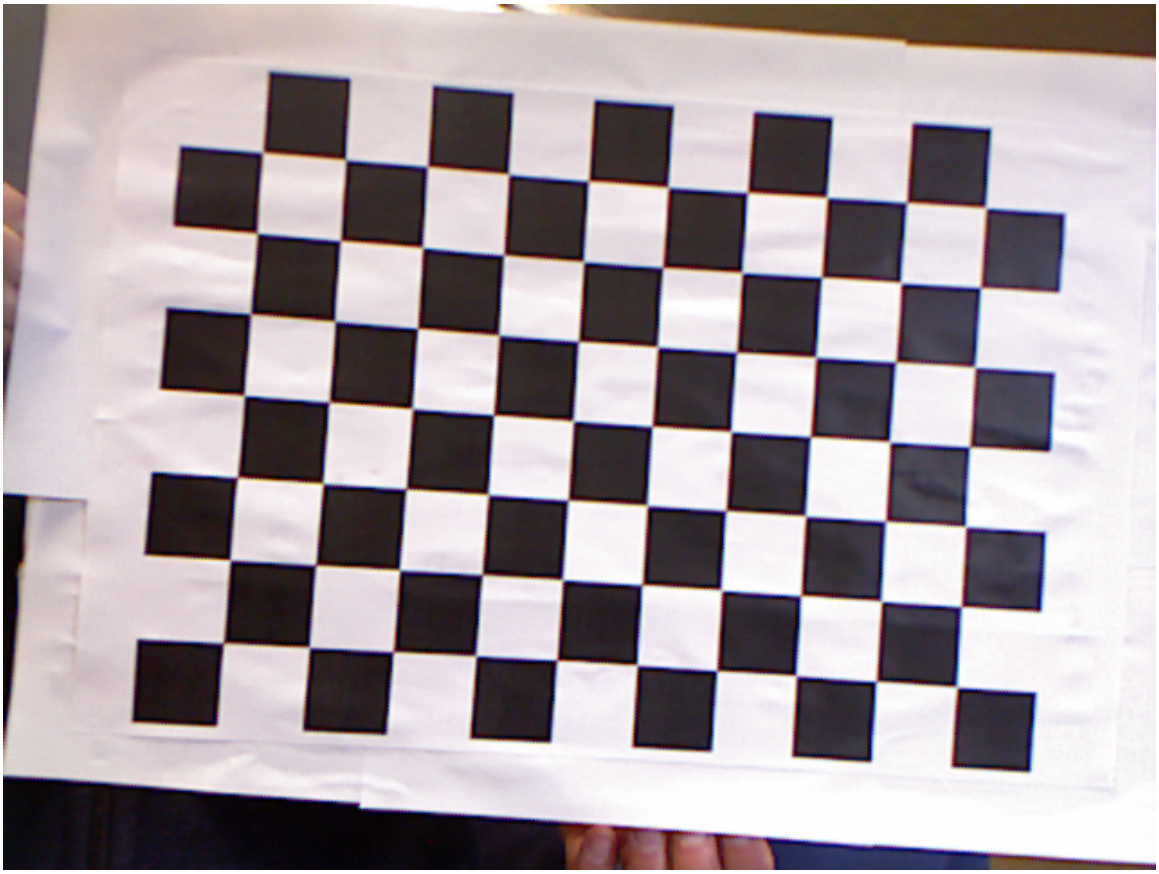


Figure 3.1: Figure shows a screenshot of the chessboard used for obtaining intrinsic parameters of the Kinects

```
(cx, cy) -is a principal point which is usually the image center
```

See Appendix A.1.1 for a detailed procedure for obtaining the above intrinsic parameters for each of the Kinect cameras using Nicholas Burrus' RGBDemoV0.4.

## 3.2 Extrinsic Calibration of the Kinects and the WorldViz PPTH/X Tracking System

This section describes ***Step - 2*** of Algorithm 1. In a multiple Kinect setup, each Kinect has its own coordinate system in which it reports positions (e.g., 3D points making up the body mesh, tracked head position, etc.). Also, WorldViz PPTX itself

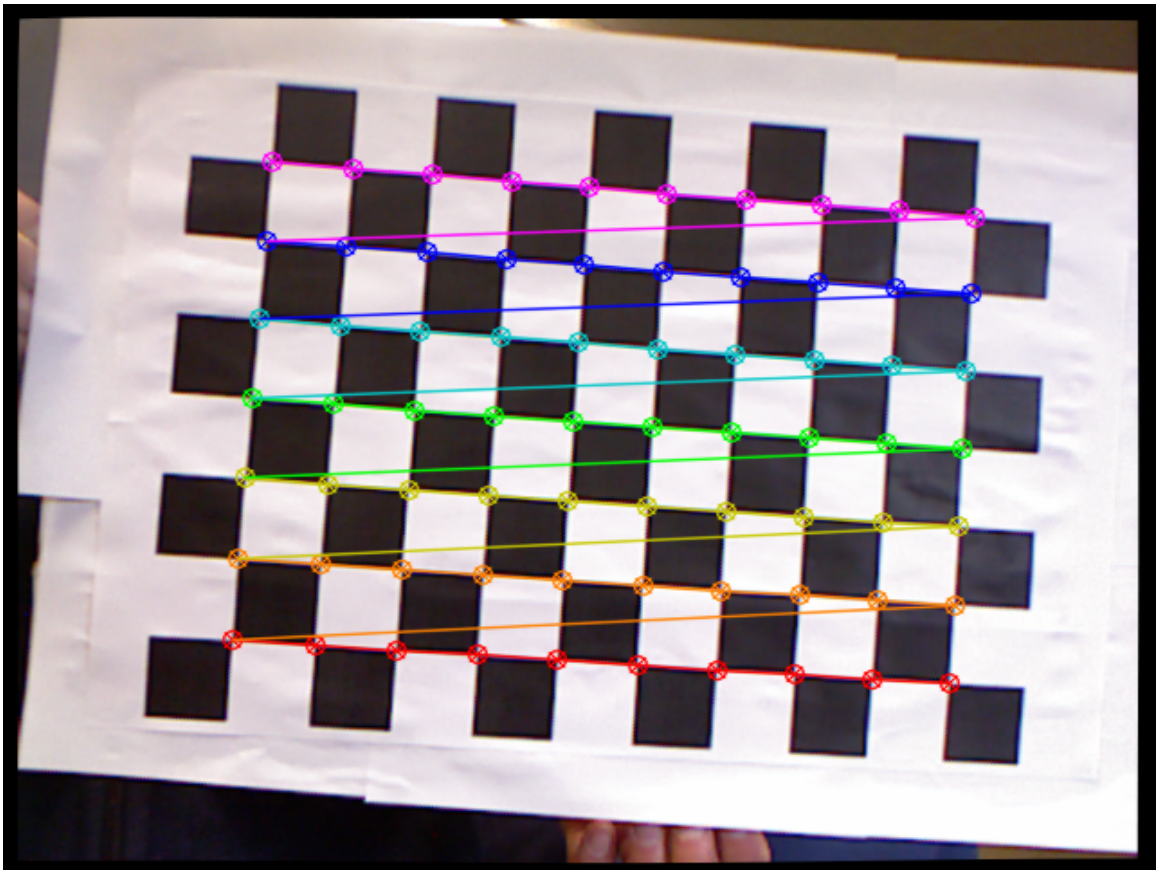


Figure 3.2: Processed chessboard Color image

reports positions in its own coordinate system based on the location and orientation of the WorldViz IR calibration target (see Figure 3.3) during the calibration process. In order to have a multi-tracker setup with trackers collaborating with one another, it is necessary that all the trackers report points in a unified global coordinate system. For example, multiple Kinects (see Figure 4.3) arranged in a scene to get a 3D avatar and/or 3D gesture recognition and/or 3D position tracking are expected to report data in the same coordinate system so that the data from different Kinects can be merged into the global coordinate system. Moreover, if WorldViz PPTH/X is used for position tracking, then it has to operate in the same coordinate system as the Kinects. Otherwise, the head positions reported by WorldViz do not map to the actual head position according to the user mesh generated by the Kinect, and vice-a-versa. As a

Figure 3.3: A figure of WorldViz PPTH/PPTX Calibration Rig that shows the *origin* marker glowing

result, the user would not be able to experience any visual body feedback. Therefore, all Kinects need to be calibrated together for the desired functionality.

In order to get all the Kinects and the WorldViz tracking systems into a unified global coordinate system, an extrinsic calibration matrix $M$ needs to be calculated. For this purpose, our position tracking system (described in detail in Chapter 4) is used to capture the marker positions on the WorldViz Calibration Target (See Figure 3.3) whose world positions (as reported by the WorldViz PPTH/X tracking system) are then used for estimating the final transformation matrix $M$ for each of the Kinects. For alternative procedures see Nicholas Burrus' [7] RGBDemoV0.4 [9]

website. The detailed procedure is given in Appendix A.1.3.

## 3.3 3D User Reconstruction

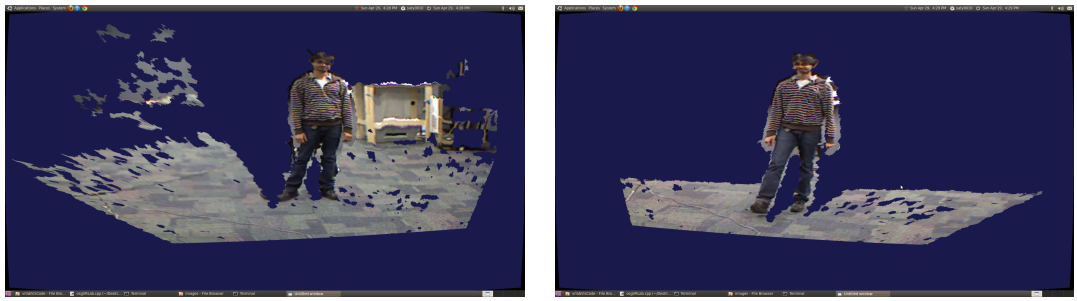
This Section describes ***Step - 3*** of Algorithm 1. Specifically, it describes how multiple Kinects that are already calibrated using the procedures described in Sections 3.1 and 3.2 can be used for capturing the 2.5D [20] point clouds, segmenting and generating a 3D representation of the user, which can then be used to provide realtime visual body feedback for users in HMD-based VEs. Specifically, Section 3.3.1 describes how the pixels corresponding to the user are identified and segmented from the depth image. Section 3.3.1 gives the equations that use the 2D pixel positions, the intrinsic and extrinsic calibration parameters of the Kinect (calculated using procedures described in Sections 3.1 and 3.2), respectively, to recover the corresponding 3D (2.5D) points that make up the user. Section 3.3.2 then describes how the user is then represented as a 3D (2.5D) mesh of triangles.

### 3.3.1 Segmentation of User Pixels

This section describes ***Step - 4*** of Algorithm 1. The RGB and depth images from the Kinect contain information about the entire scene whereas only the information corresponding to the actual objects in the scene is required for visual body feedback. To segment/separate out user pixels from the rest of the background pixels, two types of segmentation techniques are applied on the depth images namely, *Depth Thresholding* and *Static Background Subtraction*.

#### Depth Thresholding

*Depth Thresholding* is analogous to applying a Band Pass Filter on the depth image. That is, a depth value in the depth image is considered valid only if it lies



(a) RGB image before applying Depth Thresholding on the depth image

(b) Depth image after applying Depth Thresholding on the depth image

Figure 3.4: Figure shows depth image before and after applying Depth Thresholding. You can see that b) does not fully remove all the background pixels (the ones on the floor), but removes most of the distant background pixels.

within a predefined *range*. Every *invalid* depth value (even if it is not the default invalid value, 2047) is then replaced by an INVALID_DEPTH value (the default invalid value, 2047). The pseudo code is given below:

**Algorithm 2** Depth Thresholding Pseudo Code

```
// load current depth image from the Kinect
depth ← current depth image from the Kinect

for all pixel p ∈ ℕ² in depth do
  if depth( p ) ≥ MAX_DEPTH or depth( p ) ≤ MIN_DEPTH then
    depth( p ) ← INVALID_DEPTH
  end if
end for
```

For the purposes of figures, the obtained depth image is used as a mask for removing corresponding background pixels from the corresponding RGB image. The threshold depth range in this algorithm is practically determined based on the application. The main purpose of this process is to remove the far background and to reduce the number of pixels for the Static Background Subtraction technique (the next step of the segmentation process) to deal with. Figure 3.4 illustrates the effects of Depth Thresholding on the depth image. The current implementation though, performs Depth Thresholding as a part of 3D Mesh Generation, described later in Section 3.3.2.

### Static Background Subtraction

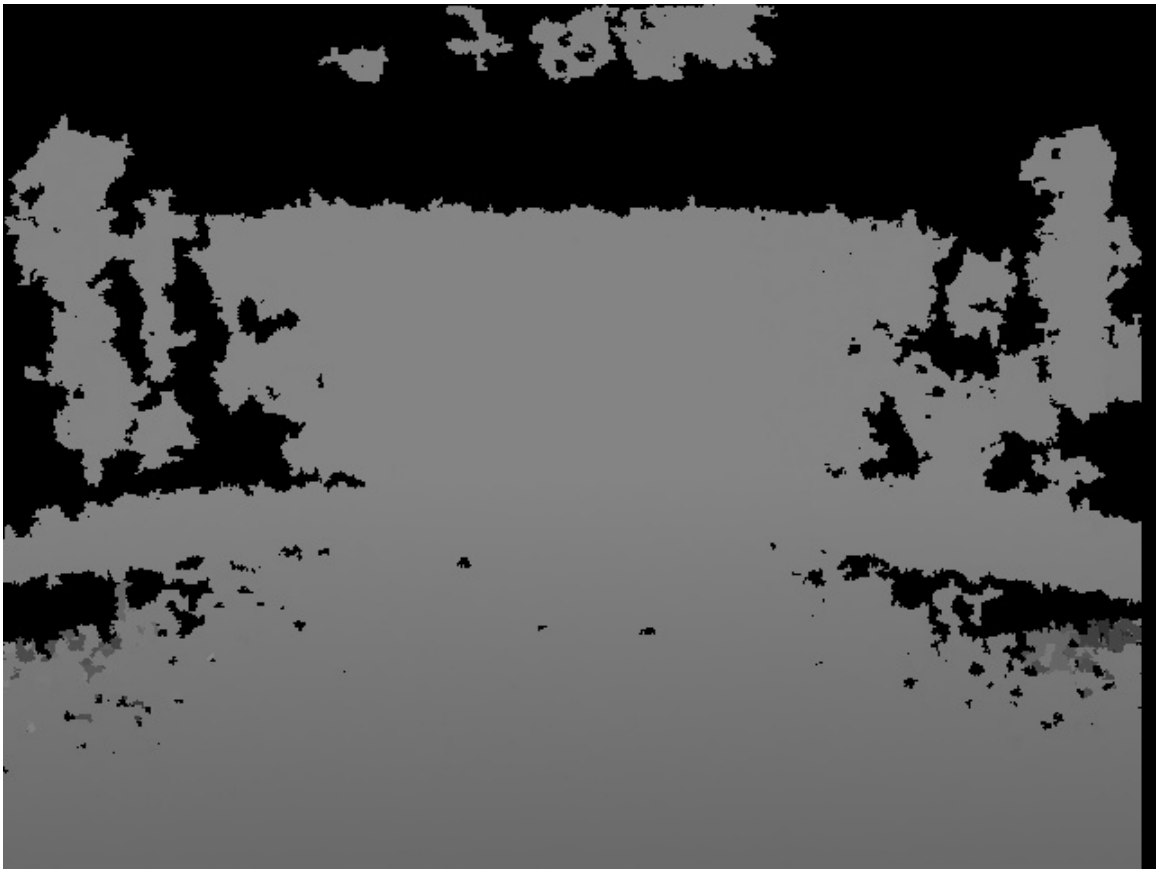


Figure 3.5: A sample background image constructed using the Appendix Code Snippet 3.

As the name indicates, *Static Background Subtraction* is the process of removing static background pixels based on a static background depth image. That is, initially, a background depth image is constructed (see the Algorithm 3) with just the scene background in the Kinect's view. This background image is later used for isolating (see Algorithm 4) foreground pixels in the captured depth images since the full scene now has user and other foreground objects.

Figure 3.5 shows a background image constructed using the above code snippet for Background Construction. Algorithm 4 shows how this background depth image is used for performing Background Subtraction on depth image.

**Algorithm 3** Background Construction Pseudo Code

```
/* tempDepth will store current depth information and bgDepth will hold the constructed background depth. */
// Initialize bgDepth with current depth image from the Kinect
if !getAsyncRawDepth( &bgDepth ) then
  displayKinectMessage( "Not detected!!!" );
  return false;
end if
/* updateCount (initialized to zero here) indicates the number of new background depth values identified during the current iteration and changeCount (assigned zero at this step) indicates how many consecutive frames have passed without a single update to the bgDepth */
// Loop until we find no significant changes to the bgDepth
repeat
  // get the current depth image
  if !getAsyncRawDepth( &tempDepth ) then
    displayKinectMessage( "Not detected!!!" );
    return false;
  end if
  // reset updateCount for this iteration
  updateCount = 0;
  /* check if any pixel in tempDepth is a valid pixel that is not identified during the previous frames in the bgDepth */
  for k = 0 to 640 * 480 do
    // Check if the current pixel is a newly identified valid depth value
    if bgDepth[ k ] == INVALID_DEPTH && tempDepth[ k ] != INVALID_DEPTH then
      // Update the current depth pixel with the new value
      bgDepth[ k ] = tempDepth[ k ];
      updateCount ++;
    end if
  end for
  // If there is no update during the current iteration, increment changeCount
  if !updateCount then
    changeCount++;
  else
    // There was an update in the current iteration. So, reset changeCount
    changeCount = 0;
  end if
  /* If a fixed number of frames have passed without any modification to the bgDepth then we assume that the bgDepth has converged and stop updating it */
until changeCount < MIN_NUMBER_OF_FRAMES
// The resultant bgDepth is the constructed background image
```

**Algorithm 4** Background Subtraction Pseudo Code

```
// load current depth image from the Kinect
depth ← current depth image from the Kinect

for all pixel p ∈ ℕ² in the depth do

    // Check if this is a valid background pixel
    if backgroundDepth( p ) ≠ INVALID_DEPTH then

        // Check if the depth of this pixel is similar to that of the background image
        if |depth(p) − backgroundDepth(p)| ≤ DEPTH_THRESHOLD then

            // This pixel is also part of the static background. So remove it.
            depth( p ) ← INVALID_DEPTH

        end if

    end if

end for
```

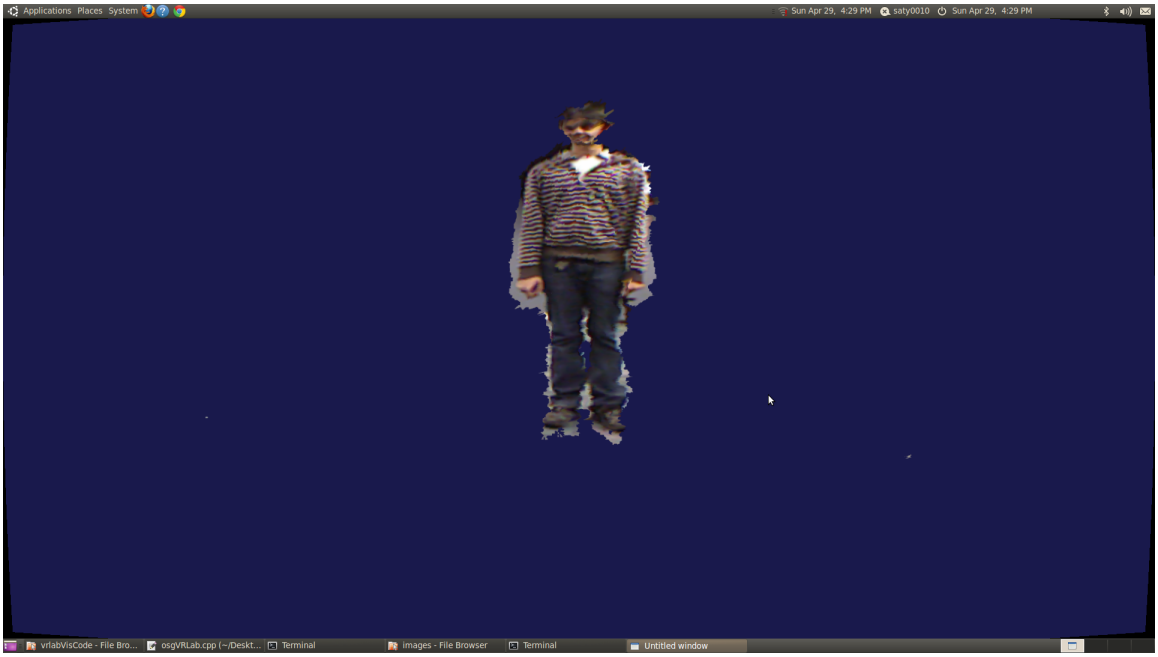



Figure 3.6: Figure shows the RGB image obtained after applying Depth Thresholding and Background Subtraction on the depth image. Observe that most of the background pixels are removed and only the user pixels remain.

Though other types of segmentation techniques (like motion segmentation, etc.) can also be applied, for the current purposes, the above two segmentation techniques were sufficient to get the desired results. Figure 3.6 shows the resultant user pixels that are segmented from the rest of the background pixels. The following section shows how these 2D user pixel positions are used to obtain the 3D point cloud corresponding to the user. It also describes how the color at each point is recovered from the corresponding RGB image.

### Equations for Recovering 3D Positions

This section describes ***Step - 5*** of Algorithm 1. That is, it gives the equations that are necessary to convert the 2D depth pixels into 3D physical positions. The 2D depth pixel coordinate ($x_{image}$, $y_{image}$) and the 11-bit raw depth disparity value $d_{image}$ are used to obtain the corresponding 3D position world $P_{world}(x_{world}, y_{world}, z_{world})$. The following equations obtained from Nicholas Burrus' [7, 8] and Stephane Magnenat' [29]s online posts can be used for this purpose:

$$z_{world} = 0.1236 \cdot tan(d_{image}/2842.5 + 1.1863) - (1)$$

$$x_{world} = (x_{image} - cx_d) \cdot z_{world}/fx_d$$

$$y_{world} = (y_{image} - cy_d) \cdot z_{world}/fy_d$$

where $fx_d$, $fy_d$, $cx_d$ and $cy_d$ are the intrinsic parameters of the depth image. Equation (1) is Stephane Magnenat's depth conversion function that converts a 11-bit raw depth disparity value into actual depth in meters. The rest of the equations are obtained from Nicholas Burrus' Kinect calibration page [8]. The points thus obtained will form the user point cloud and will be in the Kinect depth camera's local coordinate system.

Now, in order to retrieve the RGB information of each of the above point (or of those nearby) from the corresponding RGB image, the $P_{world}$ is transformed into $P'_{world}$, a point in the RGB camera space (using extrinsic/stereo calibration parameters of the RGB and depth cameras, see Sections 3.1 and 3.2 for details) and then mapped to the corresponding RGB pixel (using intrinsic parameters of the RGB camera) that contains the desired color information. The following equations obtained again from Nicholas Burrus' [7, 8] Kinect calibration page demonstrate the above steps:

$$P'_{world} = R \cdot P_{world} + T$$

$$x_{rgb} = (x'_{world} \cdot fx_{rgb}/z'_{world}) + cx_{rgb}$$

$$y_{rgb} = (y'_{world} \cdot fy_{rgb}/z'_{world}) + cy_{rgb}$$

where $fx_{rgb}$, $fy_{rgb}$, $cx_{rgb}$ and $cy_{rgb}$ are the intrinsics of the RGB camera. R and T are the rotation and translation parameters for the RGB/depth camera pair estimated during the calibration of the Kinect (see Section 3.1). The 2D RGB pixel position ($x_{rgb}$, $y_{rgb}$) holds the desired color information for a 3D point $P_{world}$. Note that the $P_{world}$ should be multiplied with the extrinsic transformation matrix $M$ for transforming the Kinects into the WorldViz PPTH/X tracking system space.

### 3.3.2 3D Triangle Mesh Generation

This section describes ***Step - 6*** of Algorithm 1. The resultant 3D points (see the above section) can now be used to obtain a 3D mesh of triangles that represent the user in the scene. Each of these triangles is generated by following a simple triangulation scheme, in which one vertex of the triangle corresponds to a depth image pixel while the other two are the nearest valid pixels to the right and bottom of it. Algorithm 5 illustrates the process as a pseudo code. Various improvements

to this simple mesh generation have been proposed that can be applied to remove outliers in the 3D surface [30].

**Algorithm 5** Mesh Generation Pseudo Code

```
for all pixel p_1 ∈ ℕ^2 in the depth image do
    // Get triangle pixels:
    p_2 ∈ ℕ^2 ← find the nearest valid pixel to the right of p_1
    p_3 ∈ ℕ^2 ← find the nearest valid pixel to the bottom of p_1

    // Compute 3D points using equations listed in Section 3.3.1:
    p'_1 ∈ ℝ^3 ← compute 3D point for pixel p_1
    p'_2 ∈ ℝ^3 ← compute 3D point for pixel p_2
    p'_3 ∈ ℝ^3 ← compute 3D point for pixel p_3

    // Add triangle to the list of triangles
end for
```

For complete details on this stage of processing, see Appendix A.2.1.

## 3.4 Results and Explanation

In order to obtain a 360° view of the user, the 2.5D point clouds from multiple Kinects are merged together into the same coordinate system, the one used by the WorldViz PPTH/X tracking system. Figure 3.7 illustrates how different views of a user surrounded by multiple Kinects are used to generate a 3D representation in the VE. In the figure, two Kinects are used to show a frontal and a side view of a VE user. Merging multiple views is done with the help of the extrinsic camera calibration parameters of each Kinect (see Section 3.2). This calibration between Kinects can also be achieved using any of the various calibration programs as supported by Nicholas Burrus' [7] RGBDemo [9], Oliver Kreylos' Kinect-1.2 [24, 25] and/or manual calibration. See [25, 9] for more details. None of these methods are fully automated though, as they all require some sort of manual intervention.

Finally, by using the resultant 3D VE, visual body feedback is experienced by the user. Figures 3.8 and 3.9 demonstrate how a user experiences visual body feedback

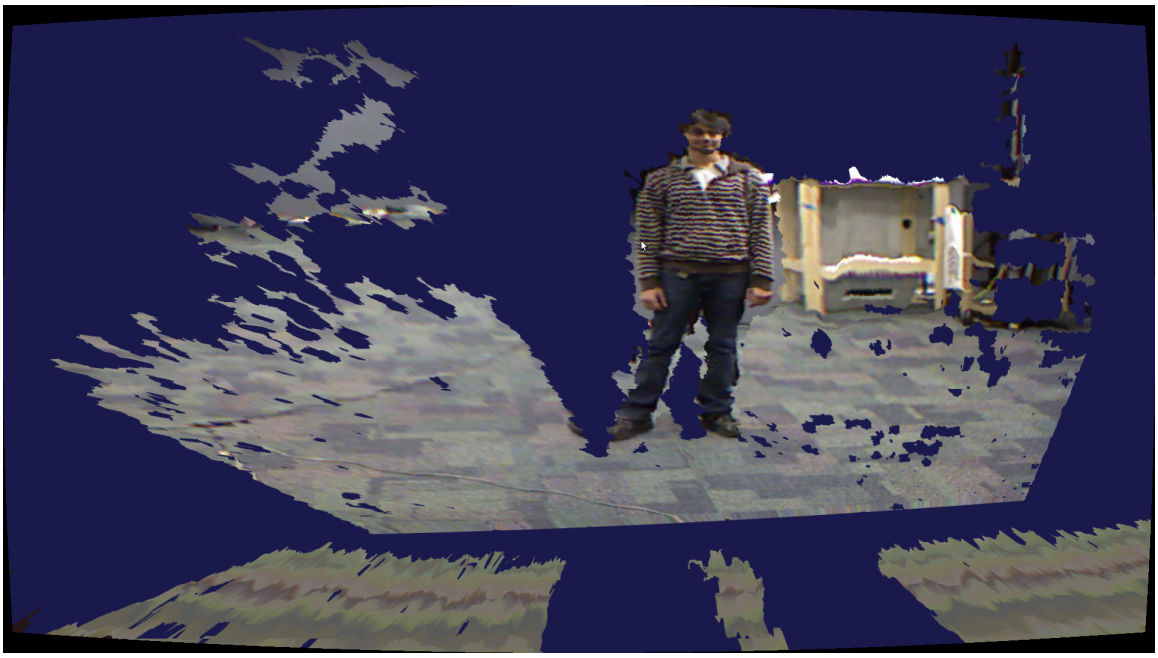


Figure 3.7: Merged 3D views from two different Kinects

when looking through the HMD.

The above figures demonstrate that the 3D user point cloud captured using multiple Kinects can be reprojected into a VE, and that by representing the point cloud and the reported tracked head position in the same coordinate system, the user can experience a real time visual body feedback. Even though the visual body feedback experienced by the user is very noisy (see Section 3.5) and by no means is a perfect representation of the user, we believe that this rough representation is a positive step towards obtaining a better visual body feedback, and can be improved with advancements in Kinect hardware and interference mitigation techniques [31]. Also, this can be considered a significant step towards obtaining a fully Kinect-based VE (see Section 3.6).

Figure 3.8: A figure of visual body feedback that shows user feet

## 3.5 Limitations

The visual body feedback experienced here by the user is noisy due to several reasons, namely: incorrect camera calibration (see Section A.1.1), interference between multiple Kinects, lower depth image resolution, short visible range of Kinects, and approximated user representation. Each of these are explained in the following with suggested ways to overcome them.

*Camera calibration*, especially intrinsic calibration, (see Section 3.1) is prone to manual errors that occur while capturing different orientations of the chessboard. Improper lighting conditions/insufficient number of chessboard samples can also result in an incorrect camera calibration. By automating the calibration process, most of the manual errors can be avoided. Moreover, by using methods (like the one described in [18]), calibration can be done independently of the lighting conditions, thus fully eliminating the calibration issues.

*Interference* (see Section 2.1.3) between Kinects is the major contributor of noise

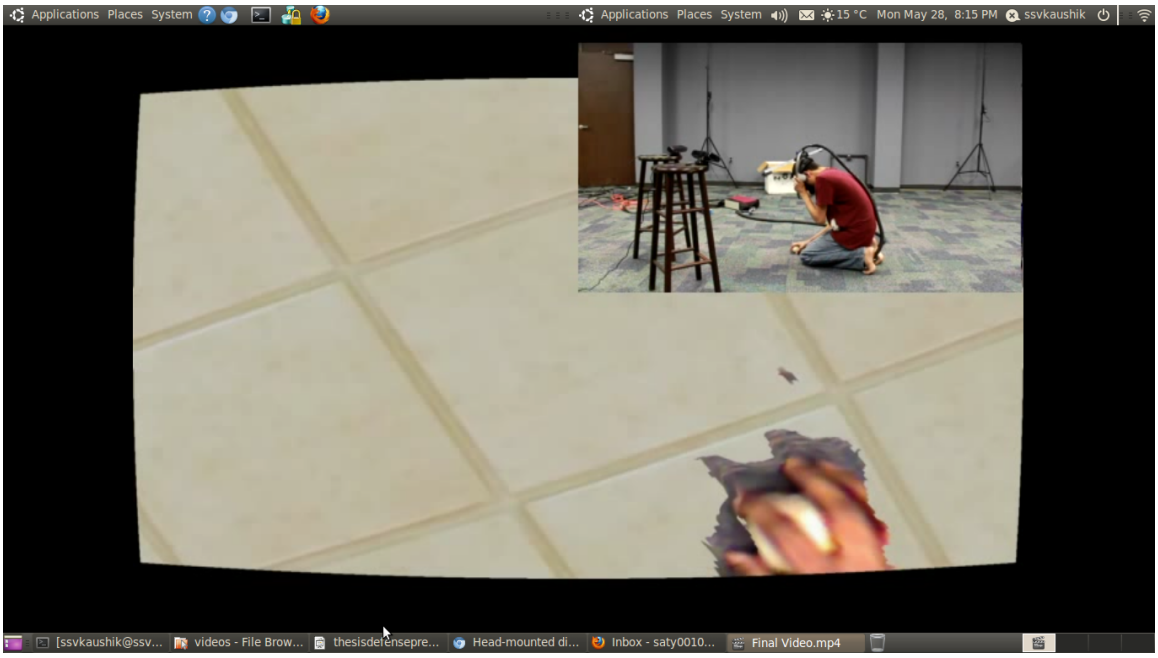



Figure 3.9: A figure of visual body feedback that shows user hands

in the user representation. Interference between the projected IR patterns in the areas of overlapping Field of Views of the Kinects results in a large number of invalid depth values in the respective depth images captured by each of the Kinects. This would cause the resultant 3D user mesh (see Algorithm 5) to be heavily distorted and thus the observed low quality visual body feedback (see Figures 3.8 and 3.9). Maimone A. [31] suggests that interference can be significantly reduced by vibrating the Kinects. Though it has to be verified that this method provides correct depth values, it does seem to be a promising solution.

Due to the limitations in the Kinect hardware, the lower *resolution* depth images do not provide a sufficient number of points to get a good representation of the user. To overcome this issue, one can employ spatial/temporal coherence methods as in [20]. Improved Kinect hardware that provides higher resolution depth images at faster frame rates can also be a feasible solution. The *short visible range* of Kinects (roughly 1.2 to 2.5 m.) provides another issue for user representation since the user

points get less accurate with increased distance from the Kinect. Again, to overcome this problem, improved Kinect hardware providing greater Kinect visibility range is necessary.

Another profound limitation of the existing Kinect hardware is that more Kinects (i.e. more Kinect views) are necessary to get a good 3D representation of the user. But, due to the high USB bandwidths required for streaming the RGB/Depth/IR streams of the Kinect, a separate USB controller is necessary for each Kinect. This limits the number of Kinects that can be hosted on a single computer to the number of PCI controllers connected to the computer at that point. This limitation can be overcome if the Kinects can be connected over a network of systems (instead of having to connect all of them to a single system), thus allowing a large number of Kinects to be connected into a VE at the same time. But, this has its own limitations on the available network bandwidths due to the large packet sizes required by the RGB/depth/IR frames from each Kinect.

Algorithm 5 describes a naive way of representing/constructing user mesh and as a result gives only an approximate representation of the user points. This can be improved using much more complex mesh generation algorithms like in [30].

## 3.6 Future Scope

With improved Kinect hardware and enhanced calibration techniques, it is quite possible to get an accurate representation of the user, thus giving a *true* visual body feedback. A *true* visual body feedback will not only enhance user presence in VEs, but also help in more natural user interactions with the VEs. A true visual body feedback will also be useful in social based VEs especially when a user is trying to communicate with friends. Previous work [12] has been done for understanding the potential of 3D telepresence for collaboration between users at remote locations and

the preliminary results are somewhat promising. Overall, it is believed that a true visual body feedback will make the communication natural for both of them rather than when using random 3D model-avatars.

# 4 IR-based Position Tracking System using Multiple Kinects

This chapter describes how an IR-based position tracking system using multiple Kinects can be built for VEs that do not require highly accurate position tracking. The idea is that the IR image of a Kinect can be used to retrieve the 2D position of an IR marker in its view, which can then be converted into a 3D position using the equations given in Section 3.3.1. Ideally, the IR marker can be attached to the user's head enabling the Kinects to track user head position. By using multiple Kinects for this purpose (see Figure 4.3 and Section 4.2.3 for a detailed discussion of such a system), an IR marker can be tracked across the entire 3D space. A pilot study was also conducted to analyze the effects of potential interference on such a tracking system. Figure 4.3 shows a typical multiple Kinect setup that can be used for the proposed system.

Combining multiple views can strengthen skeletal tracking mechanisms used by APIs such as Microsoft's Kinect SDK [11] or the OpenNI framework [35]. The skeletal tracking in these systems is quite limited in range (up to 3-3.5 meters), most likely because of the *depth resolution* 4.2.2 of the Kinect, which decreases with increasing distance from the Kinect. The proposed tracking system exploits the fact that even though it is very difficult to estimate the entire set of skeletal positions after a certain range, it is still possible to get adequately reliable depth values to get a reasonable estimate of the user's head position within a certain extended range (like 5 meters). The following sections tackle these statements in detail.

Section 4.1 describes the implementation of the IR tracking system in detail. It describes one by one about the issues that need to be overcome before an IR marker can be tracked by a Kinect. Section 4.2 introduces the interference pilot study, discusses the experiment setup and then analyzes the results. Section 4.3 discusses the current limitations of the tracking system and ways to overcome it while Section 4.4 discusses the future scope of the proposed tracking system.

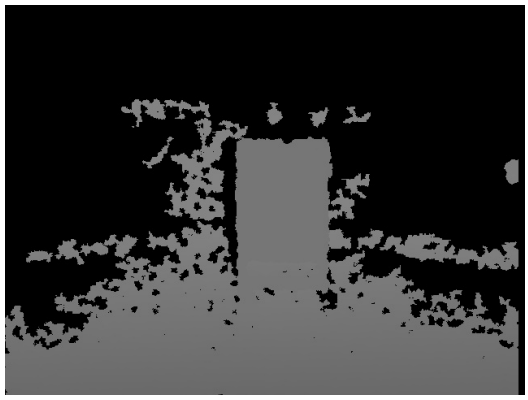
## 4.1 Tracking the IR Marker

Tracking is here based on the assumption that there are no other moving IR light sources in the scene other than the marker itself. The goal is to track an IR marker position across the lab space in the presence of IR interference from other Kinects. This is typically done by tracking the brightest pixel in the 2D IR image. But, any interference from other sources like other Kinects can result in multiple bright spots in the IR image (see Figure 4.1(c)), confusing the tracking system. To remove this direct IR interference from other static IR sources like Kinects, a simple threshold-based static background subtraction technique (as described in Section 3.3.1) is employed on the depth image to nullify the pixels that correspond to the static background and hence, the pixels corresponding to the IR light sources such as Kinect (see Figure 4.1). A series of OpenCV morphological operations [10] are also performed on the depth and IR images to remove any static noise present in them. The resultant depth image(see Figure 4.1(b)) is used as a mask for the IR image so that only the pixels that belong to the foreground will be considered by the tracking system for tracking the 2D marker position. Note that the interference is no longer a concern because the pixels corresponding to Kinect (which is a part of the static background) were removed during the background subtraction process. As a result, the IR tracking system does not get confused while searching for the brightest pixel (as it searches only among the

foreground pixels) based on the depth image mask. For the background subtraction process, the acquisition of the background depth image (see Figure 4.2) is done just once upon starting the tracker system (similar to that done for user segmentation in Section 3.3.1). Thereafter, the IR marker is identified as the brightest pixel region in the resultant foreground (see Figure 4.1(d)).

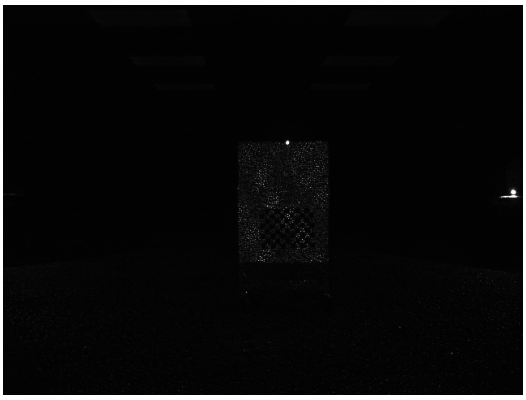
Now that the IR marker position is identified, the corresponding depth value should be recovered from the depth image. A problem here is that the IR marker creates its own small circle of IR interference around it that makes it difficult to precisely find its corresponding depth from the depth image (see Figures 4.1(a) and 4.1(b)). The tiny black hole in the depth images at the pixels where the IR marker is mounted on the cart (see Figure 4.1) illustrates this. To overcome this issue, a *locality of neighborhood* technique is used to estimate its depth. That is, a small neighborhood of pixels around the tracked 2D marker position is considered to estimate the depth at the marker position. The current implementation considers a 7x7 neighborhood of pixels around the 2D marker position and uses the mean of those depths to get the estimated depth value. This *locality of neighborhood* method imposes an additional restriction on the tracking system in that the surface surrounding the marker has to be relatively flat (see Section 4.3 for a detailed discussion on this limitation and proposed solutions).

Once the 2D marker position and the raw depth disparity value are identified, the correponding real world IR marker position is computed based on the equations listed in Section 3.3.1.

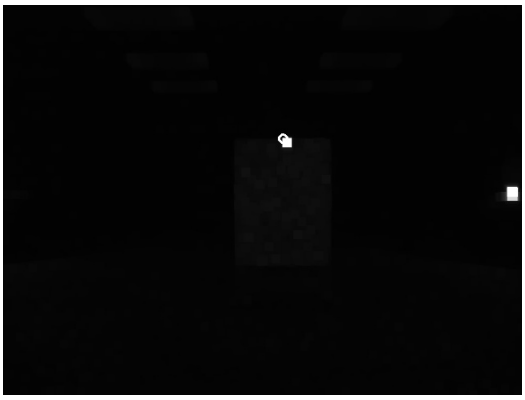


(a) Depth image before background subtraction

(b) Depth image after background subtraction



(c) Unprocessed IR Image shows interference from another Kinect (see the light on the right edge).



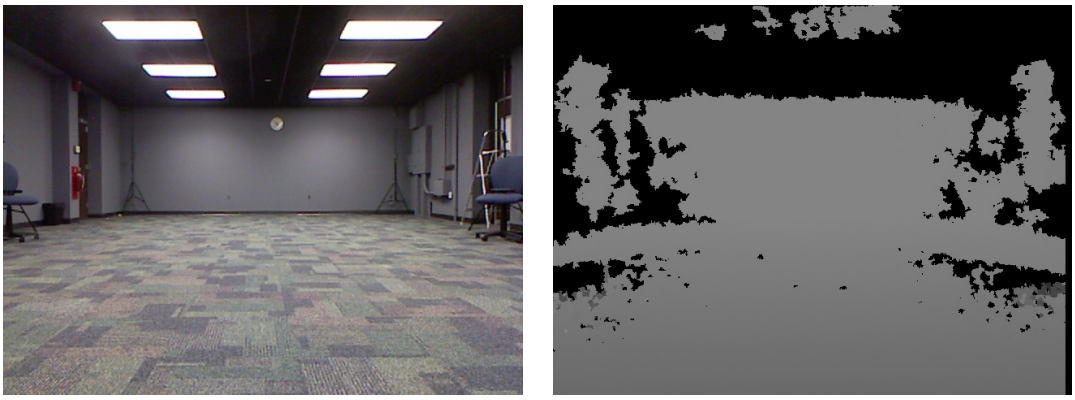
(d) Processed IR image shows the tracked marker position. Notice that the interference from the Kinect no longer confuses the tracking system

Figure 4.1: Figure illustrates the interference issue and the use of background subtraction technique for mitigate its effect on the tracking sytem.

## 4.2 Results and Explanation

### 4.2.1 Introduction to the Pilot Study

The overall objective of this work is to understand the efficacy of using multiple Microsoft Kinects as tracking devices for monitoring a user's position and a user's skeletal system in a VR application. For this purpose, this pilot study uses a Kinect for tracking an IR marker across the space of a 7m×10m lab, focusing on the jitter of position data acquired over distance and in the presence of multiple Kinect sensors



(a) RGB image of a sample scene background (b) Depth image of the constructed background

Figure 4.2: Figure shows a) RGB and b) constructed depth images of the scene background.

(as shown in Figure 4.3) that can cause severe IR interference [4, 41].

### 4.2.2 Experiment 1 - Interference Analysis

In this experiment, the effects of IR interference on the proposed tracking system are evaluated using a pilot setup consisting of five Kinects mounted on chairs of the same height, with one being the primary Kinect (see Figure 4.3), while the remaining ones are used to introduce interference. The Kinects are arranged in the lab room to provide tracking over a 7m×7m area. A WorldViz IR marker mounted on a cart is used as the marker that is tracked. The Kinects are calibrated as given in Section 3.1. As the other Kinects are just passive no explicit extrinsic calibration is performed for this experiment.

To evaluate the effects of IR interference, 30 manually marked positions (prone to manual errors) on the floor were used for placing the cart before being tracked. These positions are marked approximately one meter apart from each other (see Figure 4.4).

At each position, 1000 samples of the IR marker positions are collected, which are then averaged to get the final estimate of the tracked position. This is done both with and without interference from the other Kinects. The entire step is repeated for

Figure 4.3: Figure shows multiple Kinects arranged in a typical VR Lab setup for a 360° coverage of a circular space of radius 3.5m. The solid green circle shows the active *primary* Kinect, while the dashed green circles show the passive *secondary* Kinects.

each of the 30 positions. The jitter in these IR marker positions is then analyzed over all the 1000 samples with and without interference.

Figure 4.5 shows a comparison of the tracked positions with and without interference. The green circles represent the estimates of the positions measured without interference, while the red crosses represent the estimates of the positions measured with interference from other Kinects. Standard deviations of reported pixel positions were nearly zero in all cases, except for the points shown in Figure 4.5 at which the interference was maximum. In particular, the mean and standard deviations of a single point with maximized interference was a mean depth value of 1016.0 (SD=89.6723). Without interference at this location, the mean depth value was 1039.6 (SD=1.0484).

These results suggest that the effects caused by IR interference are almost negligi-

Figure 4.4: A portion of the floor that shows the manually placed marker positions roughly 1m. apart.

ble except for those points that are very close to the Kinects (see Figures 4.3 and 4.5). That is, for a Kinect-based position tracking system, the effects of interference are insignificant enough to be ignored. As each Kinect is roughly $150, the entire tracking system (see Figure 4.3) costs less than $1000 in comparision with the much more expensive (though more precise) WorldViz PPTH/X tracking systems [38]. Thus, we believe that our tracking system can be used as a low cost alternative to existing tracking systems. Our system is also quite portable. Another advantage the tracking system provides is that, a fully Kinect-based VR system will be possible (see Section 4.4).

### Depth Function and Depth Resolution Plots

For further analyzing the reliability of the depth values measured by the Kinect over a distance, Stephane Magnenat's (see Section 3.3.1) depth conversion function f(see Figure 4.6), and what we call the *depth resolution* of the Kinect are plotted (as shown in the Figure 4.7) using MATLAB. Here, we define *depth resolution* of the Kinect $\rho$ as:

$$\rho = 1/(f(rawD) - f(rawD - 1))$$

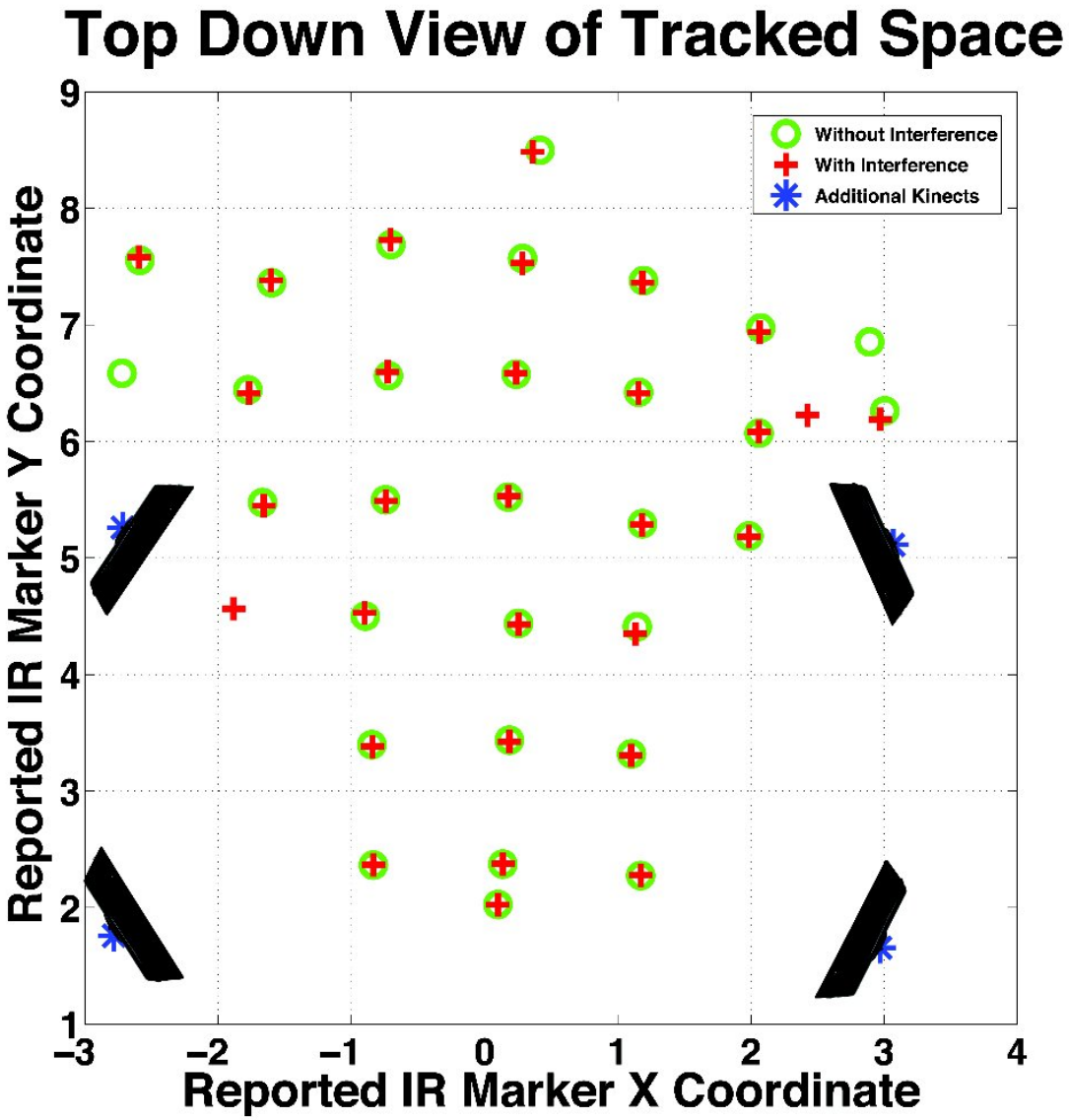



Figure 4.5: Comparision of the Kinect-tracked marker positions with and without interference from other Kinects. The green circles represent marker positions tracked without interference from other Kinects, while the red plus represent positions obtained in the presence of interference from other Kinects. The blue asteriks indicate the approximate placement of the Kinects.

where rawD is the 11-bit raw depth disparity value reported by the Kinect. At a higher level, depth resolution helps in understanding how good the Kinect can be at

distinguishing depth values with increasing distance.

The graph 4.7 shows that with increased distance the depth resolution decreases drastically, making the depth resolution almost insignificant after around 5-6m. Thus, depth resolution quantifies the number of distinguishable physical values that can be reported by the Kinect between any two depth values. For instance, one can observe from the graph that between depth values 6 and 7 m. the distinct raw Disparity values reported by the Kinect are very much less (around 5 to 10 from the graph). A corollary from this would be that distinct marker depths can be measured only in depth steps of (roughly) 0.1 to 0.2 m. These depth steps get larger with increased depths as seen from the graph, making the Kinect completely unreliable for identifying/distinguishing larger depths.

Thus, for larger distances, due to the depth resolution limitation of the Kinect, our tracking system becomes inoperable. This can be overcome by improved Kinect hardware or by using multiple Kinects (see Section 4.3).

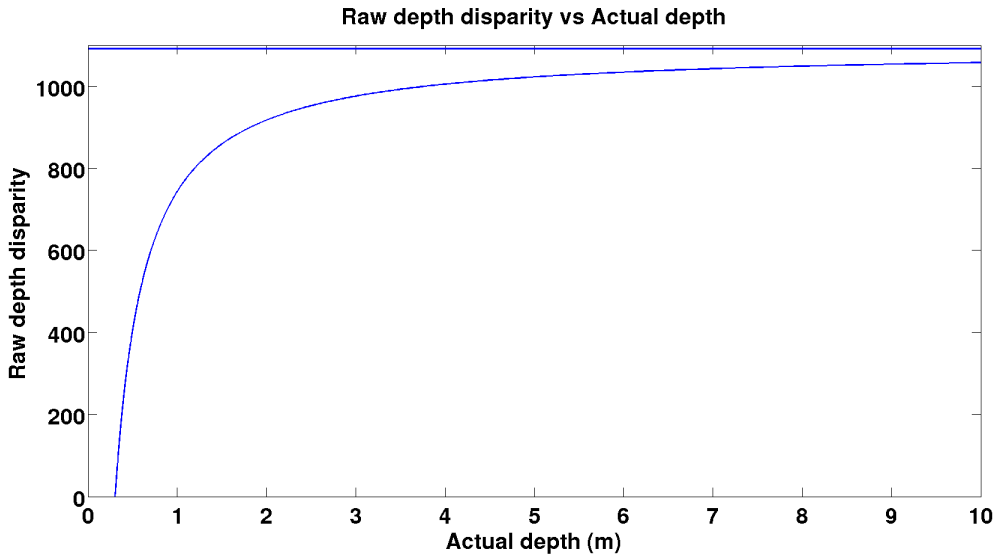



Figure 4.6: Figure shows Stephane Magnenat's depth function on the X-axis plotted against 11-bit raw depth disparity values obtained from the Kinect.

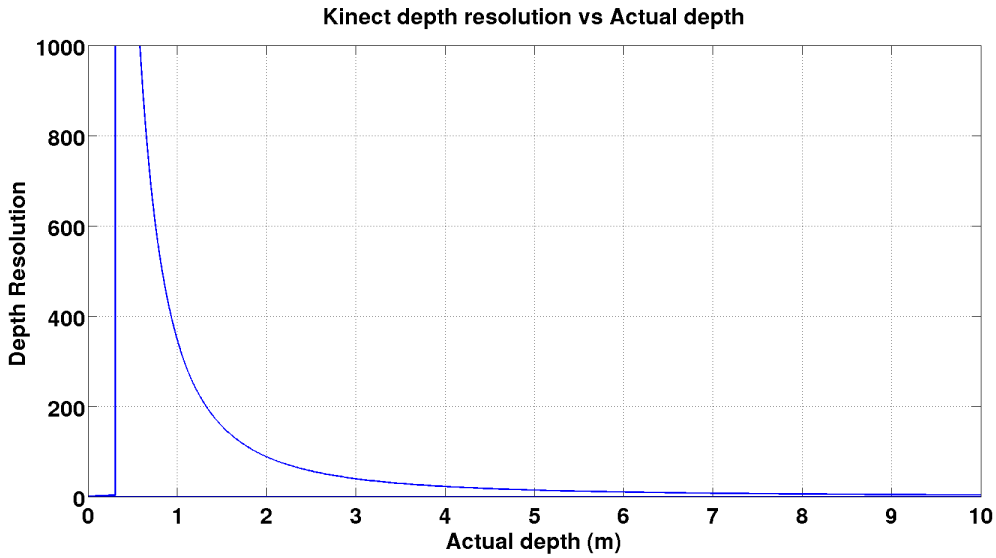



Figure 4.7: Figure shows depth resolution on the Y-axis plotted against actual distance/depth in meters from the Kinect.

### 4.2.3 Collaborative Position Tracking Using Multiple Kinects

Despite the potential issues involved (as discussed in 2.1.3), multiple Kinects can be used conveniently for position tracking not only because it's quite easy to have multiple Kinects connected to different physical systems spread across a network, but also because the actual data transfer required for collaboration is very small (on the order of a few bytes!). Such a system can also improve the overall peformance of the tracking system by using network based optimizations and also filtering approaches like Kalman filters [21] to robustly estimate the marker positions at any point of time.

Our initial attempt at creating such a position tracking system uses 5 Kinects (see Figure 4.3) communicating over the network with one another. One of them is considered *server* while the rest of them are *clients*. The *server* is responsible for *merging* the data obtained from the *clients*. The *merging* is necessary because, at any instant of time, one or more of the *client* Kinects can see the same marker and report the respective tracked positions to the *server* and the *server* has to decide which one to use. To do that, the *server* first validates the data obtained from different

Kinects, *weighs* them based on the *confidences* reported by the *client* Kinects and then merges them based on the weights and various filtering approaches like Kalman Filter [21] in order to get a better and a robust estimate. In our implementation, we used the *depth resolution* (see 4.2.2) as the confidence with which the *client* Kinect reports its tracking data to the server. This is because the depth resolution gets worse with distance of the marker and serves as a reliable confidence measure of the value reported by the Kinect.

Our implementation suffered from high latency as we did not implement any of the optimizations suggested above. One should note that there is a tradeoff between latency and robustness in this system. Using a smoothing filter typically increases the robustness of the value reported by the system, but increases the overall latency of the system. On the other hand, using a very basic filter (or none) will result in a very good overall performance of the system, but the value reported might be very noisy due to the inconsistencies in the tracked values reported by each Kinect. The tracking incosistencies might depend on the distance of the marker from the Kinect, orientation of the surface surrounding the marker, amount of interference from other Kinects, etc. Though we implemented a relatively simple smoothing filter (based on weighted average of the values reported), the smoothing operation resulted in high latency. We believe this can be improved by using more complex filters or more intelligent decision making on the *server* side as to which Kinect to trust more at any given instant.

## 4.3 Limitations

The pilot evaluations revealed a few limitations of our current system. For instance, it appears quite challenging to extend the tracking to more than one IR marker (especially when using multiple Kinects) and thus to multiple users or rigid bodies.

One possible way of overcoming this limitation is to use spatio-temporal coherence information of multiple markers to distinguish between IR markers on different users. Even with that, it is still challenging to uniquely distinguish and identify a specific marker from other markers with multiple Kinects at use. As a result, scaling the tracking system into a full-fledged tracking system is very difficult because, getting the position and orientation data of a user head, one would require at least 4 marker points (often situated on a rigid body) to be tracked across multiple frames. All these limitations can be mostly overcome by using skeletal joint positions and orientations reported by OpenNI [35] or Microsoft SDK [11] in conjunction with our position tracking system. This will not only allow the realtime position and orientation of the user to be determined, but also makes it possible to use these joint positions and orientations for realtime natural gesture recognition of the user across 3D space. A problem with skeletal tracking though, is that it is limited by the operating range of Kinect (roughly 1.2 to 3.5 m) [14] and can only be overcome using improved Kinect hardware or by using multiple Kinects spread across the space with partially overlapping fields of view to accomodate the shift in movement from one Kinect space to the other.

Another limitation is that the current Kinect hardware interface can only stream either RGB images or IR images over the USB connection. Since our head tracking approach uses IR images to compute IR marker positions, whereas our visual body reconstruction uses RGB images for colored visual body feedback, it is not practically feasible to combine both approaches using a single Kinect (with the current Kinect hardware). This limitation can be overcome by improved Kinect hardware and/or allocating some Kinects just for position tracking and the rest for obtaining visual body feedback.

One advantage of having Kinects spread across the network is that the data packet required to be communicated is essentially very small because it needs to contain

just the position and orientation data (6 values). Depending on the implementation, the packet data might also contain weights that indicate the confidence (see Section 4.2.3) with which the Kinect sensor can report the corresponding marker position. Nevertheless, the packet size is small, making it very feasible to relocate the Kinects across remote (physically distant) networks for tracking/communicating user positions across remote VEs (see Section 4.4).

Yet another limitation is that the current implementation of the tracking system uses the principle of locality of neighborhood (see Section 4.1) on depth images. Due to this, the implementation requires a relatively flat surface (a few pixels) below the physical IR marker. The only practically feasible solution for this seems to be using spatio/temporal techniques to identify the depth observed at that position during the immediate previous frames. This is because, due to the local circle of IR interference around the IR marker, it is not possible for the Kinect to estimate depth at the exact marker position.

As the Kinects provide images with a refresh rate of 30Hz, with correct implementation, the latency of the overall system should be realtime. Latency refers to the delay in reflecting a single physical movement of the user inside the VE. Even though, the latency of the proposed position tracking system has not been tested, using network based optimizations (see Section 4.2.3) and improved tracking algorithms is bound to improve the overall performance of the tracking system.

## 4.4 Future Scope

With improved Kinect hardware and by employing the techniques described in Section 4.3, it is possible to build a fully Kinect based IR tracking system that can track the joint positions and orientations of multiple users in realtime across large VR lab spaces or across remote VEs. Also, combining the tracking system discussed in

this Chapter 4 with the visual body feedback approach described in Chapter 3, it is possible to implement a fully Kinect based VE that not only provides realtime visual body feedback for the users but also allows them to naturally interact in realtime using 3D gestures. This will also allow users situated across remote VEs to communicate with one another and interact in a fully social-based VE.

# 5 Software Design and Implementation

The previous chapters have discussed the design and implementation details of the proposed multiple Kinect based visual body feedback and the IR position tracking systems. This chapter introduces the software implementation that binds the Kinects, the WorldViz PPTH/X tracking system, the 3D models together and responsible for making the proposed implementations possible. It briefly describes the overall software hierarchy, and the contribution of all the major APIs in providing what the user application needs.

An Object Oriented approach was followed when designing the entire hierarchy of classes/libraries on top of the Kinect hardware. Section 5.1 describes various open source API that are used by our application. Section 5.2 discusses our software hierarchy in detail, and Section 5.3 describes the basic functionality of each of the classes in *libkinect* (see Section 5.2), the API we developed for Kinect.

## 5.1 Introduction To The Various APIs Used

The following open source API are directly used by our VE application.

**OpenSceneGraph:**

OpenSceneGraph (OSG) [36] is an open source 3D graphics API that operates over OpenGL to render complex simulations/3D models for games and other

applications. It represents elements in a scene as nodes in a graph and provides flexibility to modify the state of each of the nodes in real time with the help of various callback functions. We use OSG to render our VR lab model and also the 3D user mesh obtained using multiple Kinects (see Chapter 3).

**libfreenect:**

Libfreenect API [34] is an open source library that interfaces with the Kinect hardware via USB drivers. It is also built using OpenGL for rendering the visual images. We use libfreenect API for almost every access to the raw RGB/Depth/IR streams from the Kinect.

**OpenCV:**

OpenCV API [10] is an open source image processing library that is used for real time computer vision. It can be considered the ultimate tool kit for any computer vision/image processing tasks. We use OpenCV mostly for removing noise from the depth/IR images with the help of a series of morphological operations provided by the OpenCV API such as the morphological erode, dilate, open and close operations.

**yaml-cpp:**

yaml-cpp [46] is a C++ parser for the YAML [37] format (stands for YAML Ain't Markup Language). We use yaml-cpp API to parse and extract information from the calibration files for each Kinect. The calibration files use YAML 1.0 format for representing the intrinsic/extrinsic and the stereo calibration parameters for the Kinect cameras.

**OpenNI:**

OpenNI [35] is an open source natural interaction API that allows users to interact with applications using natural gestures and voice. It operates on the

top of a variety of hardware sensors (like Kinects) for obtaining user inputs. Though we don't fully support OpenNI in our software as of now, we do have support for extending our API to use OpenNI drivers instead of libfreenect to interface with the Kinect hardware. This will enable us to acquire skeletal tracking data in addition to the RGB/Depth/IR streams.

The following APIs are used indirectly (mostly) for calibration purposes.

**RGBDemoV0.4:**

Nicholas Burrus' RGBDemoV0.4 is an open source API built on top of the libfreenect and OpenCV APIs and provides many features like people detection, calibration, etc. that can be used by the applications built on top of it. We used this API for the chessboard based intrinsic calibration of the Kinects (see Section 3.1). The rgbd-viewer is used to grab the chessboard samples while the calibrate_kinect_ir is used to generate YAML [37] calibration files (see Section 3.1 and A.1.4) that are used by libkinect API for calibrating the Kinects.

**Kinect-1.2:**

Kinect-1.2 [25] is another open source API built on the top of VRUI toolkit [26] for incorporating Kinect data into VR applications. This does provide alternative calibration mechanishms (though extremely laborious) for Kinects. The AlignPoints application of this library estimates the best transformation that can be applied between any two ordered sets of points. We use alignpoints for extrinsic calibration (see Section 3.2) of the Kinects/World Viz tracking systems.

## 5.2 Software Hierarchy

This section describes the interconnection of the components that make up our software/hardware and the interactions between them. Figure 5.1 illustrates the entire software hierarchy. We describe the role played by each of the components (see Section 5.1) in the software hierarchy as follows (bottom to top):

**Kinect Hardware:**

Kinect hardware comprises of the RGB/Depth/IR cameras that continuously stream data over the USB to the host computer. One should note that the RGB/IR cameras both use the same USB bus and require very large bandwidths, and hence only one of them will be streamed at any one point of time.

**libfreenect:**

libfreenect API interfaces with the Kinect hardware using the host's USB drivers and makes the data available to be used by higher level API (libkinect, in our case).

**OpenNI:**

OpenNI is also be used for the same purposes as libfreenect except that OpenNI also provides skeletal tracking data to the libkinect. The current OpenNI implementation does not support multiple Kinects, though.

**OpenCV:**

OpenCV API is used for image morphological operations by the libkinect API for removing noise in the images.

**libvrlab:**

libvrlab API manages the entire VE and its components. It is responsible for everything in the VE right from updating the camera position and orientation to deciding what the user sees in his viewing frustum. It interacts with the worldviz tracking system using the VRPN protocol [39] and updates the camera position and orientation based on the user head positions/orientations tracked by WorldViz tracking system. It also take cares how the virtual objects interact with user actions.

**libkinect:**

libkinect API is a highler level API that operates over middle level API like libfreenect/OpenNI to interact with Kinect hardware. It uses OpenCV for image processing, yaml-cpp for parsing calibration files. It passes on the parsed Kinect information to higher level applications like OpenSceneGraph applications, for instance.

**yaml-cpp:**

yaml-cpp parser is used by libkinect API for parsing and extract data from the YAML based calibration files.

**Keyboard:**

A set of preassigned keys are used to turn on/off various features provided by the libkinect and also to shift between various modes of operation of the libvrlab and libkinect.

**OpenSceneGraph Application:**

OpenSceneGraph application takes care of building and managing the OpenSceneGraph node hierarchy using various dynamic callbacks associated with each of the nodes in the hierarchy. It takes care of loading object models into

the scene and placing them in the VE. It uses libvrlab API for updating the viewing frustum and uses libkinect API to update the merged 3D user point cloud obtained from multiple Kinects.

**Calibration Files:**

These files are named after the serial number of the corresponding kinect and represent calibration data for that particular Kinect in YAML 1.0 format.

**User:**

A user looking at the VE through a HMD will experience visual body feedback and can possibly interact with the objects in the VE in realtime. His head position and orientation can also be tracked dynamically and used by the libvrlab API to decide which images have to be rendered in the left and right HMD *eye* screens.

## 5.3 libkinect API

This section briefly describes the role played by each of the classes in the libkinect API.

**KinectManager:**

This class is responsible for managing all the Kinects right from their initiation to termination. On initiation, it allocates a separate thread for each Kinect which does synchronous updates on the desired data streams of the corresponding Kinect. This class also provides support for streaming position tracking data over the network for a multiple Kinect based position tracking system. It is capable of running in two different modes viz. *server* mode, in which it

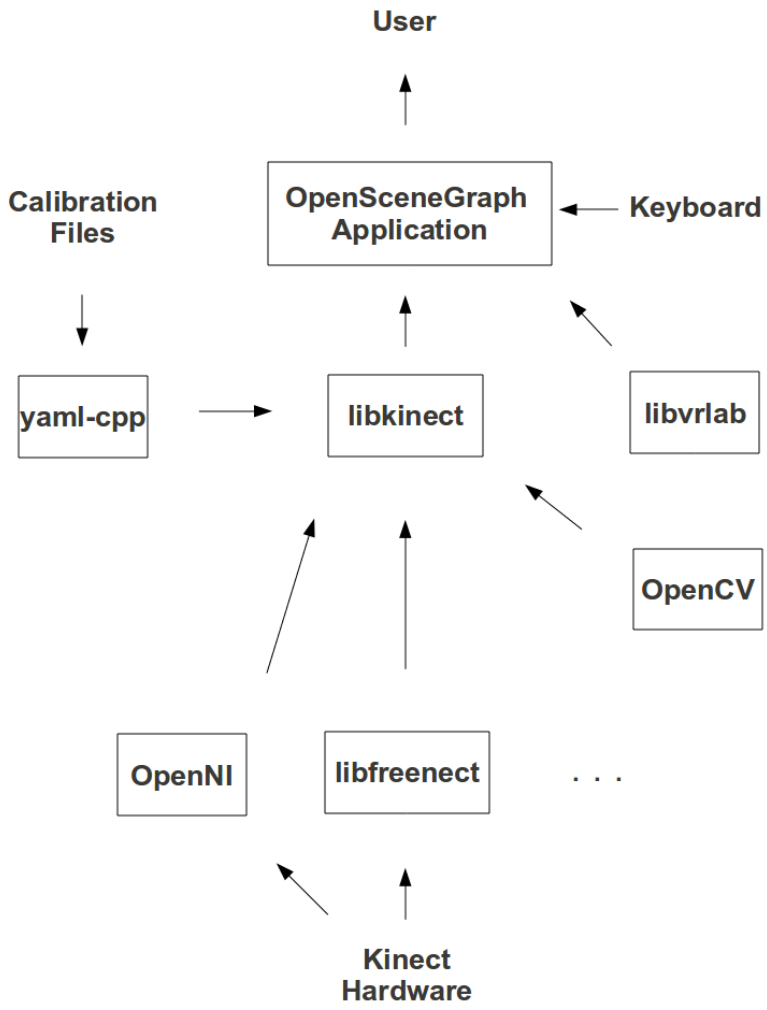



Figure 5.1: Figure shows the entire software hierarchy that the raw information from the Kinect hardware goes through before reaching the user. The "...' represents APIs such as libfreenect, OpenNI that interface with the Kinect hardware

can acquire marker postions tracked and reported by the client Kinects spread across the network and *client* mode, in which it is responsible for tracking the marker positions and reporting them across the network to the server Kinect. The server thread is also responsible for merging/validating the position tracking data acquired from the client Kinects (see Section 4.2.3).

**OSGKinectClassWrapper:**

This class acts as an interface between OpenSceneGraph application and KinectClass. It processes the data obtained from the KinectClass and converts it into OpenSceneGraph compatible vertex arrays and matrices. When running in *tracker* mode, this class implements the IR-based position tracking and maintains its state to reflect the recently tracked positions of the marker.

**KinectClass:**

This class forms the heart of the libkinect API and is reponsible for interfacing directly with libfreenect or OpenNI drivers. This class mainly represents all the calibration, data streams, serial number that correspond to a particular Kinect. This class also processes the raw data obtained from the Kinect into usable information (like indices of a 3D triangle mesh). Most importantly, this class provides functionality to toggle between various modes of operation like depth segmentation, motion segmentation and the triangulation scheme (see Algorithm 5). This class also provides support for background subtraction, depth conversion mechanisms etc.

**KinectCalibrationData:**

This class acts as an ADT representing the entire calibration data of all the Kinect RGB/Depth/IR cameras and the stereo transformations among them.

**CameraCalibrationData:**

This class acts as an ADT representing the calibration data that is specific to a specific Kinect camera. It typically contains the intrinsic, distortion parameters that correspond to specific Kinect camera (RGB/Depth/IR). It also contains functions that are capable of extracting each of the parameter values from the YAML [37] calibration files.

**KinectKeyboardHandler:**

This class acts as an interface between the keyboard and the users application. It maps each key press to a corresponding action. For example, key "d" would toggle the depth segmentation feature of the libkinect API ON/OFF.

**KinectGeometryUpdateCallback:**

This class hosts the OpenSceneGraph geometry node update callback function and is responsible for updating the geometry of each of the Kinects every frame.

**KinectTransformUpdateCallback:**

This class hosts the OpenScenegraph transform node update callback function and is responsible for updating the transformation of each of the Kinects every frame.

**OpenNIDriver:**

This class is not fully implemented at the time of writing this thesis. This class is meant to be a wrapper between KinectClass and the OpenNI API to interface with the Kinect hardware. At the time of the writing this thesis, OpenNI support for handling streams from multiple Kinects is extremely limited and is one of the primary reasons why this driver is not fully implemented.

# 6 Conclusions

In this thesis, a real time visual body feedback approach for HMD-based VEs is introduced that generates *true* self-avatars based on the user point clouds obtained from multiple Kinects. Though there have been previous attempts at providing visual body feedback to the users with the help of in-animated avatar models, animated avatar models and *true* self-avatars, none of them is as general as the kinect-based *true* self-avatars obtained using multiple Kinects. There are several issues that had to be overcome for acquiring the proposed visual body feedback approach. All of these were explained in detail and were overcome one by one in a methodogical approach. While the resultant visual body feedback produced by the current implementation is very noisy and needs improvement, better looking self-avatars are possible with improved Kinect hardware, automated calibration methods, interference mitigation techniques, and better meshing algorithms.

This thesis also proposes a low-cost IR-based position tracking system for providing user head tracking in HMD-based VEs. Its implementation was discussed in detail explaining the several issues that were overcome. A pilot study was also conducted to analyze the behavior of the resultant tracking system in the presence of interference from multiple other Kinects. The results show that the position tracking capabilities of Kinect are not significantly altered by the effect of interference. That is, interference from other Kinects is not a major concern for Kinect IR-based position tracking system at least for the typical VR lab setups. The Kinect depth measurement is also analyzed in general to understand how it scales across depth. Our results show that Kinect depth measurement grows significantly less reliable for distances more than

5-6 m.

The proposed tracking system can work as an alternative to the existing, highly expensive, tracking systems such as WorldViz PPTH/X for the VR applications that can not afford such high end tracking systems and/or do not require super accuracy for position tracking. The proposed tracking system can also be scaled across multiple systems spread over a network. The current implementation does not allow multiple markers to be tracked. It also does not support rigid body tracking and/or orientation tracking of the markers. But, with better tracking techniques and with better network optimization techniques, it is possible to not only design a full tracking system using Kinects, but also a system that can track user skeletal joint positions and orientations and thus the 3D natural gestures of user. For simple purposes, the existing system can also be used in conjunction with APIs like Microsoft SDK and OpenNI to accommodate skeletal tracking, in addition to head position tracking. Ideally, a multiple Kinect-based tracking sytem can allow full body position and orientation tracking to support 3D gesture based natural interaction by the user in HMD-based VEs.

Another thing that this thesis introduces is a semi-automated extrinsic calibration method that uses the proposed IR-based Kinect tracking system for tracking marker of a calibration target. The calibration target used was the WorldViz PPTH/X Calibration target and makes it a lot easier to calibrate any type of IR tracking systems into the same coordinate system with least manual intervention. We believe that this is the most flexible calibration method at the moment that allows extrinsic calibration across a variety of IR tracking systems.

For each proposed system, the implementation, the issues that had to be overcome, the results, limitations and the future scope are discussed in detail in this thesis. The software implementation hierarchy was also discussed in detail emphasizing on the behaviour of each hardware/software component and interactions that are carried

out between one component and the other. In summary, this work is a significant step towards developing a fully-Kinect based social VE in which users across multiple VEs can not only experience visual body feedback and see one another, but can also interact with objects in the VEs and also with one another using gesture based 3D natural interaction.

# Bibliography


[1] Kinect official website. http://www.xbox.com/en-US/kinect.

[2] Kinect wikipedia. http://en.wikipedia.org/wiki/Kinect.

[3] L. Almeida, P. Menezes, L. D. Seneviratne, and J. Dias. Incremental 3D body reconstruction framework for robotic telepresence applications. In *Proceedings of the IASTED International Conference on Robotics*, pages 286–293, 2011.

[4] K. Berger, K. Ruhl, C. Brümmer, Y. Schröder, A. Scholz, and M. Magnor. Markerless motion capture using multiple color-depth sensors. In *Proceedings of Vision, Modeling and Visualization (VMV)*, pages 317–324, 2011.

[5] William B. Thompson Betty J. Mohler, Heinrich H. Bulthoff and Sarah H. Creem-Regehr. A full-body avatar improves distance judgments in virtual environments. In *ACM/SIGGRAPH Symposium on Applied Perception for Graphics and Visualization*, August 2008.

[6] G. Bruder, F. Steinicke, K. Rothaus, and K. Hinrichs. Enhancing presence in head-mounted display environments by visual body feedback using head-mounted cameras. In *Proceedings of the International Conference on Cyber-Worlds (CW)*, pages 43–50. IEEE Press, 2009.

[7] Nicholas Burrus. http://nicolas.burrus.name/index.php/AboutMe/Main.

[8] Nicholas Burrus. Rgbdemov0.4 kinect calibration page. http://nicolas.burrus.name/index.php/Research/KinectCalibration, 2011.

[9] Nicholas Burrus. Rgbdemov0.4 website. http://nicolas.burrus.name/index.php/Research/KinectRgbDemoV4?from=Research.KinectRgbDemo, January 2011.

[10] Intel Corporation. Opencv wiki. http://opencv.willowgarage.com/wiki/, 2006.

[11] Microsoft Corporation. Microsoft kinect sdk for windows. http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/, September 2011.

[12] James E. Manning Bruce Cairns Greg Welch Henry Fuchs Diane H. Sonnenwald, Hanna M. Sderholm1. Exploring the potential of video technologies for collaboration in emergency medical care: Part i. information sharing. In *Journal of the American Society for Information Science and Technology*, volume 59, pages 2320–2334, December.

[13] A. Rizzo D. Krum E. Suma, B. Lange and M. Bolas. Faast: The flexible action and articulated skeleton toolkit. In *Proceedings of IEEE Virtual Reality*, pages 247–248, 2011.

[14] Sina Nia Kosari Hawkeye King Fredrik Ryden, Howard Jay Chizeck and Blake Hannaford. Using kinecttm and a haptic interface for implementation of real-time virtual fixtures. In *RSS Workshop on RGB-D Cameras*, 2011.

[15] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli. Depth mapping using projected patterns, U.S. Patent 2010/0118123, 13 May 2010.

[16] M. Garau, M. Slater, V. Vinayagamoorthy, A. Brogni, A. Steed, and A. Sasse. The impact of avatar realism and eye gaze control on perceived quality of com-

munication in a shared immersive virtual environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 529–536, 2003.

[17] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments. In *Proceedings of the Symposium on Experimental Robotics (ISER)*, pages 1–15, 2010.

[18] C. D. Herrera, J. Kannala, and J. Heikkilä. Accurate and practical calibration of a depth and color camera pair. In *Proceedings of the International Conference on Computer Analysis of Images and Patterns (CAIP)*, pages 437–445, 2011.

[19] InterSense InertiaCube. Intersense inertiacube website. http://www.intersense.com/categories/18/.

[20] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. KinectFusion: Real-time 3D reconstruction and interaction using a moving depth camera. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 559–568. ACM Press, 2011.

[21] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. In *Transactions of the ASME–journal of Basic Engineering*, volume 82, pages 35–45, 1960.

[22] K. Khoshelham. Accuracy analysis of Kinect depth data. In *Proceedings of the International Society for Photogrammetry and Remote Sensing Workshop on Laser Scanning*, pages 1–6, 2010.

[23] K. Khoshelham and S. O. Elberink. Accuracy and resolution of Kinect depth data for indoor mapping applications. *Sensors*, 12:1437–1454, 2012.

[24] Oliver Kreylos. http://www.idav.ucdavis.edu/~okreylos/.

[25] Oliver Kreylos. Kinect-1.2 website. http://idav.ucdavis.edu/~okreylos/ResDev/Kinect/, 2011.

[26] Oliver Kreylos. Vrui tool kit. http://idav.ucdavis.edu/~okreylos/ResDev/Vrui/MainPage.html, June 2011.

[27] B. Lok. Online model reconstruction for interactive virtual environments. In *Proceedings of the Symposium on Interactive 3D Graphics (I3D)*, pages 69–73. ACM Press, 2001.

[28] Cory Walker Julio Montero Aalap Tripathy Maged N Kamel Boulos, Bryan J Blanchard and Ricardo Gutierrez-Osuna. Web gis in practice x: a microsoft kinect natural user interface for google earth navigation. In *International Journal of Health Geographics*, 2011.

[29] Stephane Magnenat. http://stephane.magnenat.net/.

[30] A. Maimone and H. Fuchs. Encumbrance-free telepresence system with real-time 3D capture and display using commodity depth cameras. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 1–10. IEEE Press, 2011.

[31] Andrew Maimone and Henry Fuchs. Reducing interference between multiple structured light depth sensors using motion. In *Virtual Reality Workshops (VR), 2012 IEEE*, pages 51–54, March 2012.

[32] B. J. Mohler, S. H. Creem-Regehr, W. B. Thompson, and H. H. Bülthoff. The effect of viewing a self-avatar on distance judgments in an HMD-based virtual environment. *Presence*, 19(3):230–242, 2010.

[33] OpenCV. Opencv camera calibration and 3d reconstruction documentation web page. http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html.

[34] OpenKinect. Openkinect community site. http://openkinect.org/, September 2011.

[35] OpenNI. Openni user guide. http://www.openni.org/documentation, September 2011.

[36] OpenSceneGraph. Openscenegraph website. http://www.openscenegraph.org/projects/osg, December 2005.

[37] Clark Evans Oren Ben-Kiki. Yaml technical specification. http://yaml.org/spec/, March 2001.

[38] WorldViz PPT. Worldviz ppt website. http://www.worldviz.com/products/ppt/index.html.

[39] Adam Seeger Hans Weber Jeffrey Juliano Aron T. Helser Russell M. Taylor II, Thomas C. Hudson. Vrpn: A device-independent, network-transparent vr peripheral system. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, November 2001.

[40] D. Scharstein and R. Szeliski. High-accuracy stereo depth maps using structured light. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 195–202, 2003.

[41] Y. Schröder, A. Scholz, K. Berger, K. Ruhl, S. Guthe, and M. Magnor. Multiple Kinect studies. Technical Report 09–15, ICG, TU Braunschweig, 2011.

[42] Yaser Sheikh and Mubarak Shah. Bayesian modeling of dynamic scenes for object detection. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, number 11, pages 1778–1792, November 2005.

[43] J. Smisek, M. Jancosek, and T. Pajdla. 3D with Kinect. Technical report, Czech Technical University Prague, 2011.

[44] E. E. Stone and M. Skubic. Evaluation of an inexpensive depth camera for passive in-home fall risk assessment. In *5th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth) and Workshops*, 2011.

[45] S. Streuber, S. de la Rosa, L.-C. C. Trutoiu, H. H. Bülthoff, and B. Mohler. Does brief exposure to a self-avatar affect common human behaviors in immersive virtual environments? In *Proceedings of Eurographics*, pages 1–4, 2009.

[46] yaml cpp. A yaml parser and emitter for c++. http://code.google.com/p/yaml-cpp/, September 2009.

# A Appendix A

## A.1 Kinect Calibration

### A.1.1 Procedure for calculating intrinsic parameters of the Kinect

The following pseudo code gives the steps to be followed for calculating the intrinsic parameters of RGB/IR/Depth cameras of each Kinect using Nicholas Burrus' [7] implementation of chessboard recognition [10]:

1. With the chessboard (See Figure A.1) as a calibration target, a series of images are grabbed using the rgbd-viewer in Dual IR/RGB mode.
2. The following tips suggested by Nicholas Burrus in his RGBDemoV0.4 website (see [9]) are followed in order to obtain good calibration results:

   - Cover as most image area as possible. Especially check for coverage of the image corners.
   - Try to get the chessboard as close as possible to the camera to get better precision.
   - For depth calibration, you will need some images with IR and depth. But for stereo calibration, the depth information is not required, so feel free to cover the IR projector and get very close to the camera to better estimate IR intrinsics and stereo parameters. The calibration algorithm will

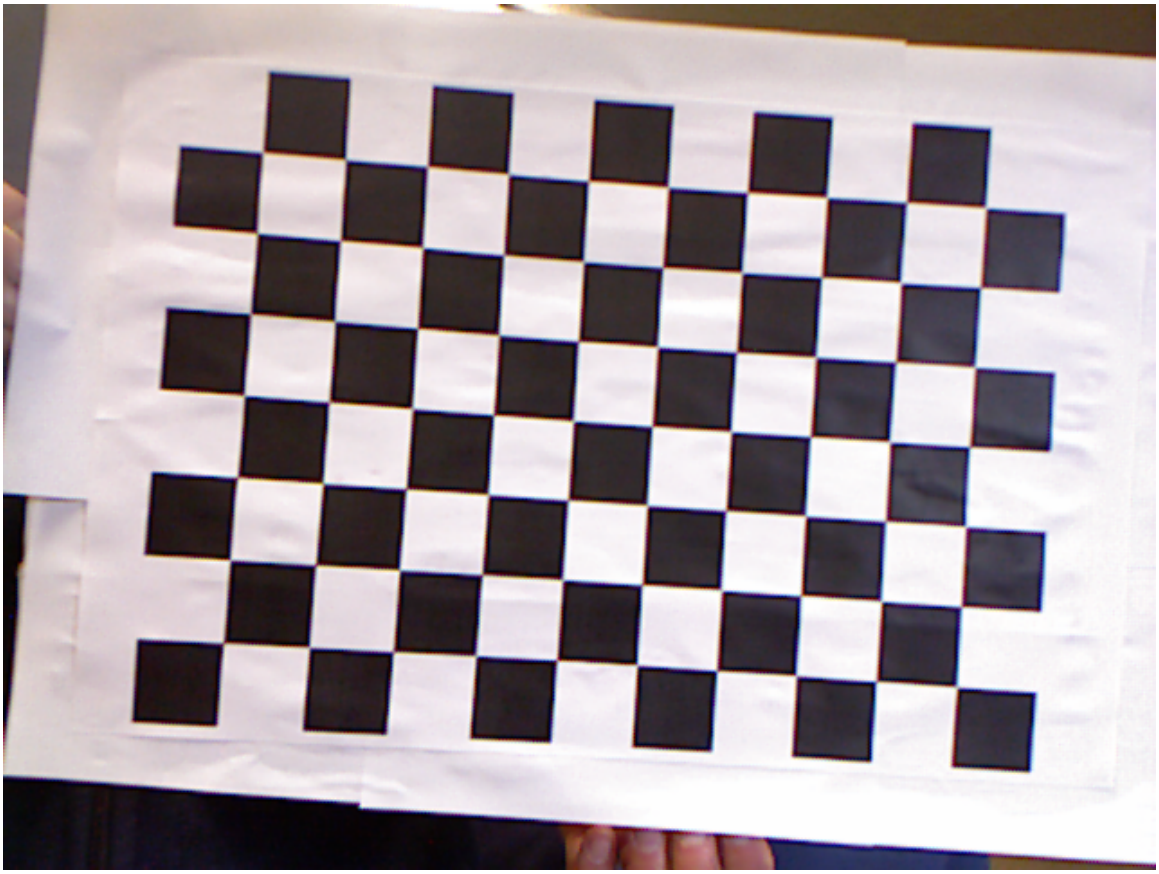


Figure A.1: Figure showing a screenshot of the chessboard used for obtaining intrinsic parameters of the Kinects

automatically determine which grabbed images can be used for depth calibration.

- Move the chessboard with various angles.
- Grab somewhere around 30 images to average the errors.
- Typical reprojection error is $< 1$ pixel. If you get significantly higher values, it means the calibration failed.

3. If the folder containing these images is *grab1*, then the program *calibrate_kinect_ir* is used to generate the calibration file, which is in YAML format [37] explained next.

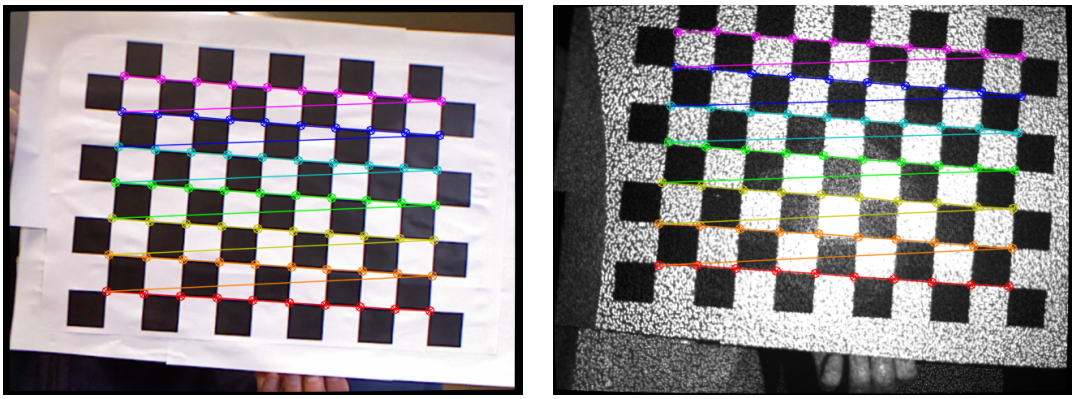
./build/bin/calibrate_kinect_ir –pattern-size *size grab1*

The above command processes each set of chessboard images in the *grab1* folder to extract the internal corners of the chessboard based on OpenCV's chessboard recognition techniques [10]. The corners from all sets/views represent the 2D *image points*, while the actual real world positions of these corners (can be interpretted based on the size of the chessboard) are referred to as *object points*. For each RGB/Depth/IR camera, these two sets of points are used for fitting intrinsic parameters of the corresponding error model [10]. The following images (See Figure A.2) illustrate one such successful set of processed images. The more such successful sets found by the above command, the better the calibration will be.

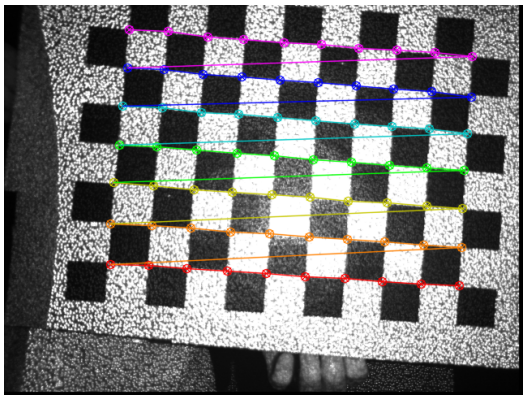
The result of the above operation is a YAML file named *kinect_calibration.yml* that would contain all the intrinsic parameters necessary for calibrating the current Kinect. Note that each Kinect will have its own intrinsic parameters and hence its calibration file. For easier identification, the calibration file names are suffixed with the *serial number* of the current Kinect that is being used. A typical YAML calibration file (see Appendix A.1.4 for the full file) is explained below as follows. The first line would be typically something like:

*%YAML:1.0 !!version-1.0*

This line indicates that the file uses YAML/YML 1.0 format [37]. **!!** indicate comments and in the above case are used to identify each matrix as OpenCV compatible calibration matrices. The next lines contain the actual calibration data in the form of a set of matrices described below. Each *matrix* element indicates the number of rows (*row* field), the number of columns (*column* field), the type of the data (*dt* field) such as float (f), double (d) etc. and the actual data (*data* field). The following lines illustrate this.



(a) (b)



(c)

Figure A.2: Figures showing processed chessboard a) RGB, b) depth and c) IR images for which the internal corners of the chessboard were successfully identified by the calibration program.

*rgb_intrinsics: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 4.538317117309028e+02, 0., 3.4750341100277677e+02, 0.,*

*4.5342394265540486e+02, 2.8733922704001145e+02, 0., 0., 1. ]*

The above lines say that the *rgb_intrinsics* matrix contains 3 *rows*, 3 *columns* of *double* data elements and they all fill up into the typical *rgb_intrinsics* matrix listed

next.

$$\begin{bmatrix} fx_{rgb} & 0 & cx_{rgb} \\ 0 & fy_{rgb} & cy_{rgb} \\ 0 & 0 & 1 \end{bmatrix}$$

where $fx_{rgb}$, $fy_{rgb}$ are focal lengths of the RGB camera expressed in pixels; ($cx_{rgb}$, $cy_{rgb}$) is the principal point of the RGB camera. The matrix when filled up with the data becomes,

$$\begin{bmatrix} 4.5383171173090284e+02 & 0 & 3.4750341100277677e+02 \\ 0 & 4.5342394265540486e+02 & 2.8733922704001145e+02 \\ 0 & 0 & 1 \end{bmatrix}$$

The next few lines contain the *depth_intrinsics* matrix as follows:

*depth_intrinsics: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 5.9315541156816209e+02, 0., 3.2328862557433655e+02, 0.,*

*5.9300201865895576e+02, 2.3509603914688100e+02, 0., 0., 1. ]*

The above lines contain the 3x3 intrinsic parameter matrix of the depth camera in the following format:

$$\begin{bmatrix} fx_d & 0 & cx_d \\ 0 & fy_d & cy_d \\ 0 & 0 & 1 \end{bmatrix}$$

where $fx_d$, $fy_d$ are focal lengths of the depth camera expressed in pixel units; ($cx_d$, $cy_d$) is the principal point of the depth camera. As with *rgb_intrinsics*, the matrix when filled up with the data becomes:

$$\begin{bmatrix} 5.9315541156816209e+02 & 0 & 3.2328862557433655e+02 \\ 0 & 5.9300201865895576e+02 & 2.3509603914688100e+02 \\ 0 & 0 & 1 \end{bmatrix}$$

The next few lines contain the *rgb_distortion* matrix as follows:

*rgb_distortion: !!opencv-matrix*
*rows: 1*
*cols: 5*
*dt: d*
*data: [ 1.5782581266995976e-01, -2.6705966682895449e-01,*
*3.8722500336058627e-04, 2.5853373795836502e-03,*
*1.4748972516351339e-01 ]*

The above lines contain the 1x5 distortion parameter matrix of the RGB camera in the following format:

$$\begin{bmatrix} k1_{rgb} & k2_{rgb} & p1_{rgb} & p2_{rgb} & k3_{rgb} \end{bmatrix}$$

where $k1_{rgb}$, $k2_{rgb}$, $k3_{rgb}$ are radial distortion coefficients, $p1_{rgb}$, $p2_{rgb}$ are tangential distortion coefficients of the RGB camera. Though these parameters are not directly used in Section 3.3.1, these can be used for undistorting the RGB image [10]. Again, the transpose of the resultant matrix *rgb_distortion* when filled up using the above data from the file, looks like (the matrix transpose is displayed because of the constraints on column width in this document):

$$\begin{bmatrix} 1.5782581266995976e-01 \\ -2.6705966682895449e-01 \\ 3.8722500336058627e-04 \\ 2.5853373795836502e-03 \\ 1.4748972516351339e-01 \end{bmatrix}$$

The next few lines contain the *depth_distortion* matrix as follows:

*depth_distortion: !!opencv-matrix*

*rows: 1*

*cols: 5*

*dt: d*

*data: [ -9.3005676375325530e-02, 3.7698605620393016e-01,*

*-1.1632681423432360e-03, -7.8314061322236198e-04,*

*-2.2958257169952864e-01 ]*

The above lines contain the 1x5 distortion parameter matrix of the Depth/IR camera in the following format:

$$\begin{bmatrix} k1_d & k2_d & p1_d & p2_d & k3_d \end{bmatrix}$$

where $k1_d$, $k2_d$, $k3_d$ are radial distortion coefficients, $p1_d$, $p2_d$ are tangential distortion coefficients of the Depth camera. Though these parameters are not directly used in Section 3.3.1, these can be used in the depth calculation formula in place of the constants. The tranpose of the resultant matrix when filled up from the data is given below (the matrix transpose is displayed because of the constraints on column width in this document):

$$\begin{bmatrix} -9.3005676375325530e-02 \\ 3.7698605620393016e-01 \\ -1.1632681423432360e-03 \\ -7.8314061322236198e-04 \\ -2.2958257169952864e-01 \end{bmatrix}$$

The next few lines contain the stereo rotation matrix $R$ and translation matrix $T$ for RGB/depth cameras, respectively:

*R: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 9.9999409789737503e-01, 2.7450204296977669e-03,*

*2.0661638985476362e-03, -2.8751851853318308e-03,*

*9.9782328141468501e-01, 6.5881957901843285e-02,*

*-1.8808191208003091e-03, -6.5887509663598187e-02,*

*9.9782528460134945e-01 ]*

The above lines contain the 3x3 stereo rotation transformation that needs to be applied between RGB and Depth cameras.

*T: !!opencv-matrix*

*rows: 3*

*cols: 1*

*dt: d*

*data: [ 2.4659216039291057e-01, -1.2954316437070665e-01,*

*6.1007098509885627e-01 ]*

The above lines contain the 1x3 stereo translation transformation that needs to be applied between RGB and Depth/IR cameras. The matrices $R$ and $T$ are used in Section 3.3.1 to map a depth pixel to its corresponding RGB/IR pixel.

In summary, the above calibration file contains information about the following matrices:

$$rgb\_intrinsics$$

$$rgb\_distortion$$

$$depth\_intrinsics$$

$$depth\_distortion$$

$$R$$

$$T$$

By default, the RGBDemoV0.4 does not output the matrices R and T, so modifications were made in order to accommodate them into the output. For details on why intrinsic calbiration is required and about Kinect calibration, see Section 2.1.2. For alternative ways of doing intrinsic calibration of Kinect see Nicholas Burrus [7]' RGBDemoV0.4 [9] website.

### A.1.2 Procedure for calibrating multiple Kinects with WorldViz PPTH/X tracking system

In order to calibrate multiple trackers/cameras into a global coordinate system, at least 4 global marker positions are tracked/reported in each tracker's individual coordinate system. So, if there are N trackers/cameras being calibrated, one will have N sets of the 4 global points, each set as reported by each of the N trackers/cameras in corresponding coordinate systems. Unless there's an additional global coordinate system being used, one of the N cameras/trackers is chosen as the reference coordi-

nate system and each of the other N-1 cameras/trackers are transformed into this coordinate system, thus making it the *global* coordinate system. Oliver Kreylos' [24] point alignment program *AlignPoints* from the package Kinect-1.2 [25] is used for this purpose. Its operation is described as follows:

**Input:**

Two sets of points, one from a local coordinate system and the other from a global/reference coordinate system in .csv format.

**Output:**

The closest matrix transformation that can be applied to convert points from individual coordinate systems into the global coordinate system.

**Sample Input:**

```
// ReferencePoints.csv \newline
-0.703702, -0.17017, 0.0851165 \newline
-0.733302, 0.380192, 0.0886507 \newline
-0.184706, 0.409148, 0.090808 \newline
-0.152493, -0.139227, 0.0887336 \newline

// LocalPoints.csv \newline
0.757287, 0.70037, 3.00293 \newline
-0.470119, 0.705546, 3.00293 \newline
-0.47625, 0.672265, 4.13546 \newline
0.729079, 0.658009, 4.13546\newline
```

**Execution:**

*./AlignPoints -OG LocalPoints.csv ReferencePoints.csv*

**Sample Output:**

```
Best distance: 0.000419787
Best transform:
translate (-2.1048444605325, 0.09871119645488, 0.45634821040751) *
rotate (-0.59880372626989, 0.57810722030265, -0.55427983838423),
118.96791377781 * scale 0.46792979730935
```

Here, the command line argument -OG specifies orthogonal transformation. An optional -S can be used if the local and reference points are not measured in the same measuring units. For example, if the reference points are in meters, the local points in cms, then a command line argument -S 0.01 should be specified (since, 1cm=0.01m).

i.e; *./AlignPoints -OG -S 0.01 LocalPoints.csv ReferencePoints.csv*

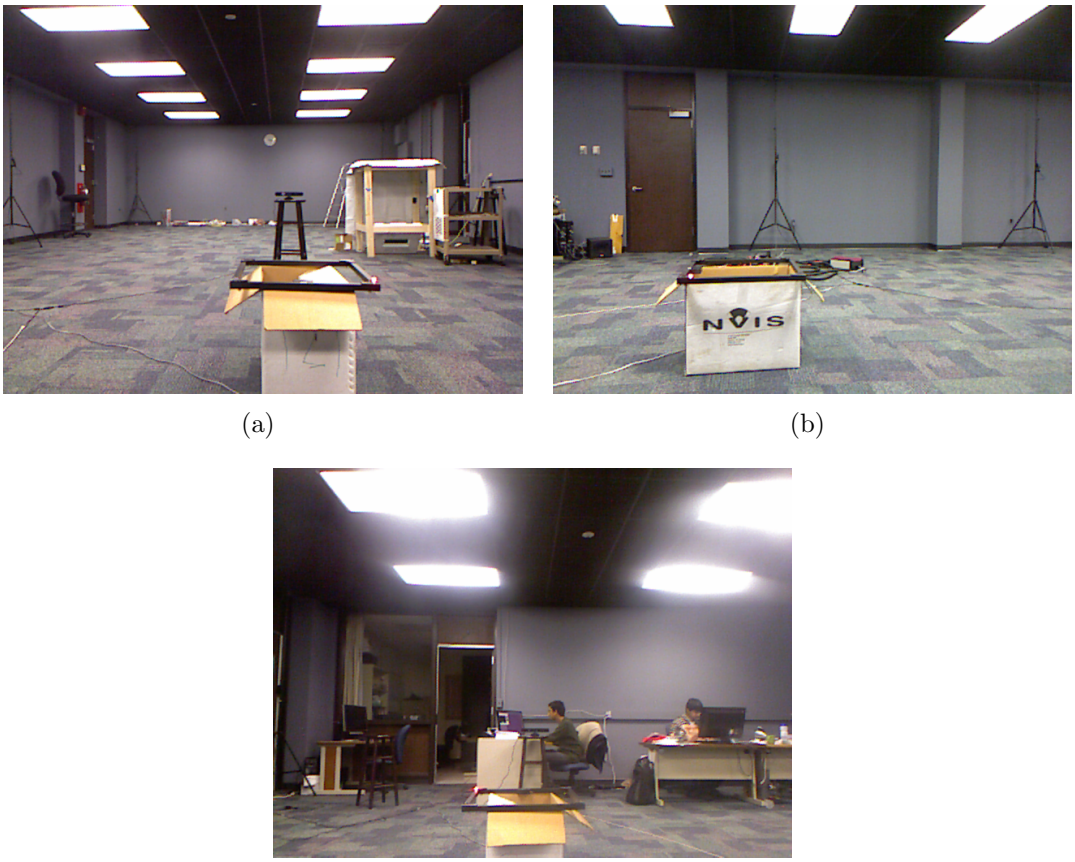
### A.1.3 Calibrating Kinects using WorldViz Calibration Target

The Calibration target contains 4 IR markers which when powered on blink alternatively in a cyclic fashion. The positions of these 4 IR markers with respect to each Kinect/WorldViz position tracking system will make up for each of the *4 point sets* that are required for the calibration procedure described in Section A.1.2.

#### Acquiring Points from the Kinect(s)

The application program *AcquireCalibrationTarget* is used to acquire each of the 4 points on the calibration target. Figure A.3 shows views from 3 different Kinects calibrated in this fashion. Each image shows the calibration rig with the blinking

*origin* marker (see the figure).




(a) (b)

(c)

Figure A.3: Figure giving calibration rig views of three different Kinects.

Having a good (the more valid depth values in the image, the better it is) background image (see Figure 4.2) and a feasible depth threshold (practically determined to remove any IR noise in the scene) is essential to get the necessary 4 points. For best results, the calibration target needs to be placed within the 2-3m range from the Kinect. It should also be made sure that all the 4 markers are in its field of view. As the physical position of the calibration target can not be changed for each Kinect, this limits the setup of the Kinects to a circle of radius 2-3m. This is a pure Kinect limitation and can be resolved only by a future version of the Kinect (hopefully with a better sensing range). Alternatively, a much brighter and possibly bigger IR lights

can be arranged within the extended range (up to 6m) to get reasonable position estimates.

Each 3D point reported by the *AcquireCalibrationTarget* program is computed using equations given in Section 3.3.1. In case a pure position tracking system built just on the Kinects is desired one of the Kinects has to be chosen as the *primary* tracker Kinect (which typically is the Kinect making up the reference coordinate system because WorldViz PPTH/X tracking system will not be used). This Kinect is responsible for giving the head position possibly backed up by a few other *secondary* tracker Kinects. See Section 4.3 for a discussion on how the two approaches can be used together by using a few Kinects for visual body feedback and the rest for the Kinect-based position tracking.

### Acquiring Points from the WorldViz PPTH/X Tracking System

For calibration purposes, the *true/raw* WorldViz 3D positions reported in the WorldViz coordinate system are ignored and the corresponding *processed/transformed* versions that make up the final *eye position* in the simulation are chosen. This is because, ultimately, all the Kinects are made to report in the *eye coordinate system* (which is already used by the WorldViz to report the eye positions).

When using the 4-point sets for calibration, care should be taken that the same *physical* order is used in both the input sets. That is, if the *origin* position of the calibration target (See Figure 3.3) is chosen as the first point in the input set #1 (followed by the other 3 marker positions in the cyclic blinking order of the LEDs on the calibration target), then the same position should be reported first in the input set #2 too!

### A.1.4 YAML Calibration File Example

The following file is a typical calibration file(see Appendix A.1.1 for a detailed explanation of the file contents) used by libkinect API (see 5.1) to extract calibration parameters of the Kinect.

*%YAML:1.0 !!version-1.0*

*rgb_intrinsics: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 4.538317117309028 4e+02, 0., 3.4750341100277677e+02, 0.,*

*4.5342394265540486e+02, 2.8733922704001145e+02, 0., 0., 1. ]*

*rgb_distortion: !!opencv-matrix*

*rows: 1*

*cols: 5*

*dt: d*

*data: [ 1.5782581266995976e-01, -2.6705966682895449e-01,*

*3.8722500336058627e-04, 2.5853737795836502e-03,*

*1.4748972516351339e-01 ]*

*depth_intrinsics: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 5.9315541156816209e+02, 0., 3.2328862557433655e+02, 0.,*

*5.9300201865895576e+02, 2.3509603914688100e+02, 0., 0., 1. ]*

*depth_distortion: !!opencv-matrix*

*rows: 1*

*cols: 5*

*dt: d*

*data: [ -9.3005676375325530e-02, 3.7698605620393016e-01,*

*-1.1632681423432360e-03, -7.8314061322236198e-04,*

*-2.2958257169952864e-01 ]*

*R: !!opencv-matrix*

*rows: 3*

*cols: 3*

*dt: d*

*data: [ 9.9999409789737503e-01, 2.7450204296977669e-03,*

*2.0661638985476362e-03, -2.8751851853318308e-03,*

*9.9782328141468501e-01, 6.5881957901843285e-02,*

*-1.8808191208003091e-03, -6.5887509663598187e-02,*

*9.9782528460134945e-01 ]*

*T: !!opencv-matrix*

*rows: 3*

*cols: 1*

*dt: d*

*data: [ 2.4659216039291057e-01, -1.2954316437070665e-01,*

*6.1007098509885627e-01 ]*

*rgb_size: !!opencv-matrix*

*rows: 1*

*cols: 2*

*dt: i*

*data: [ 640, 480 ]*

*raw_rgb_size: !!opencv-matrix*

*rows: 1*

*cols: 2*

*dt: i*

*data: [ 640, 480 ]*

*depth_size: !!opencv-matrix*

*rows: 1*

*cols: 2*

*dt: i*

*data: [ 640, 480 ]*

*raw_depth_size: !!opencv-matrix*

*rows: 1*

*cols: 2*

*dt: i*

*data: [ 640, 480 ]*

*depth_base_and_offset: !!opencv-matrix*

*rows: 1*

*cols: 2*

*dt: f*

*data: [ 7.50000030e-02, 1090. ]*

## A.2 Code Snippets

This section contains a few of the various code snippets that are used in the libkinect API.

### A.2.1 3D Triangulation/Mesh Generation

The below code snippet shows the code that generates a 3D mesh representation of the user/scenebased on the point cloud obtained from each Kinect (see Algorithm 5).

```
// This function is from KinectClass class of libkinect API
// It takes processed depth stream (after applying
// depth/motion segmentations
// (see Section~\ref{Sec:Segmentation}) as input and produces an array of
// indices representing a triangle mesh
void updateIndices( unsigned short* depthParam )
{
// ScopedLock provides a lock on indices array
// mutex until the end of this function
Mutex::ScopedLock indexLock( indicesMutex );

// A new set of indices are generated every time
indicesSize = 0;
unsigned short max, min;

// For each pixel in the depth stream
for( unsigned int i = 0; i < FREENECT_MEDIUM_FRAME_H - 1; i++ )
{
for( unsigned int j = 0; j < FREENECT_MEDIUM_FRAME_W - 1; j++ )
{

// index of the current depth pixel (say P1)
unsigned int k = i * FREENECT_MEDIUM_FRAME_W + j;

// index of the depth pixel exactly below P1 (say P2)
unsigned int l = k + FREENECT_MEDIUM_FRAME_W + 1;
```

```
// make sure both P1, P2 represent valid depths in the depth stream
if( depthParam[ k ] < maxDepth && depthParam[ l ] < maxDepth )
{
// find the min/max of these two depths
min = depthParam[ k ];
max = depthParam[ l ];
if( min > max )
{
min = depthParam[ l ];
max = depthParam[ k ];
}

// also make sure that the depth pixel
// immediately to the right of P1 (say P3)
// is valid depth
if( depthParam[ k + 1 ] < maxDepth )
{
// Now use spatial coherence of depth pixels
// to make sure that we triangulate only those pixels
// that correspond to the same physical surface/mesh
// For that we check if the range of the 3 values is
// within a specific depth threshold (5 units here)
if( ( unsigned int )( std::max( max , depthParam[ k + 1 ] )
- std::min( min, depthParam[ k + 1 ] ) ) <= 5U )
{
indices[ indicesSize++ ] = k ;
indices[ indicesSize++ ] = l;
```

```
indices[ indicesSize++ ] = k + 1 ;
}
}
// Now repeat similar process to generate another triangle for
// P1, P2 and the pixel immediately to the left of P2 (say P4)
if( depthParam[ l - 1 ] < maxDepth )
{
if( ( unsigned int )( std::max( max , depthParam[ l - 1 ] )
- std::min( min, depthParam[ l - 1 ] ) ) <= 5U )
{
indices[ indicesSize++ ] = k ;
indices[ indicesSize++ ] = l - 1;
indices[ indicesSize++ ] = l ;
}
}
}
}
}
}
```

### A.2.2 Kinect Detection

The below code shows a code snippet that is used by the KinectManager class of the libkinect API to detect all the Kinects that are currently connected to the system.

```
// Context represents a USB context i.e,
// the current state information of all the USB devices.
libusb_context* context;
```

```
// Initialize context to the current USB context
if( libusb_init( &context ) != 0 )
{
// ERROR!
std::cout << "\nError Initializing USB context\n";
exit( 0 );
}
// deviceList represents an array of pointers.
// Each pointer holds points to one USB device.
libusb_device** deviceList;

// Fill the deviceList with the list of usb devices
// that are currently connected to the host computer
ssize_t enumerateResult = libusb_get_device_list( context, &deviceList);
numKinects = 0;

// Make sure we have at least one USB device connected
if(enumerateResult >= 0)
{
// desc holds information about each USB device
struct libusb_device_descriptor desc;

// display some application specific debugging information
displayFlagStatus( "Debug Mode: ", debugModeFlag );
displayFlagStatus( "Calibration Mode: ", calibrationModeFlag );

// Iterate through each USB device to identify Kinects
```

```
for( unsigned short i = 0; i < size_t( enumerateResult ); i++ )
{
// Load the device information for the current USB device
libusb_get_device_descriptor( deviceList[ i ], &desc );

// Extract Vendor ID and Product ID for the USB device
int temp1 = ( int )( desc.idVendor );
int temp2 = ( int )( desc.idProduct );

// Check if the current USB device has
// Microsoft Kinect specific vendor/product IDs
if( temp1 == 0x045e && temp2 == 0x02ae )
{
// We found a Kinect
// std::cout << std::hex << temp1 << " " << temp2 <<"\n";

// Declare and initialize a USB file handle to the current Kinect
libusb_device_handle* handle;
libusb_open( deviceList[ i ], &handle );

// Using the device handle, read device specific
// serial number into a string
unsigned char data[256];
libusb_get_string_descriptor_ascii( handle, desc.iSerialNumber,
 data, sizeof( data ) );

// The below line when uncommented alows you
```

```
// to select Kinects with specific Serial Numbers
// if( ( data[ 0 ] == 'A'
&& data[ strlen( ( const char *) data ) - 1 ] == 'A' )
|| ( data[ 0 ] == 'B'
&& data[ strlen( ( const char *) data ) - 1 ] == 'B' ) )
{
// The calibration for this Kinect will be found
// in the calibration with this serial number as suffix
std::string serialNumber = ( const char* ) data;
std::string calibrationFileName = calibrationFileNamePrefix;
calibrationFileName += serialNumber;
calibrationFileName += ".yml";

// The below line is necessary based on the
// applications to select a specific Kinect
if( !specificKinectIndexOnlyFlag
|| numKinects == specificKinectIndex )
{
// Add the current Kinect to the list of
// Kinects detected by the application along
// with its application specific state information
clientKinects.push_back( new OSGKinectClassWrapper(
numKinects, serialNumber, calibrationFileName,
serverName, serverPortNumber, debugModeFlag,
calibrationModeFlag, trackerModeFlag, numIRMarkers ) );
}
```

```
// Increment the number of detected Kinects
numKinects++;
}
}
}
}
```

## A.3 Miscellaneous

### A.3.1 List of Skeletal Joints

The following is a list of all the skeletal joints that can be tracked using Microsoft Kinect SDK [11]. HipCenter

Spine

ShoulderCenter

Head

ShoulderLeft

ElbowLeft

WristLeft

HandLeft

ShoulderRight

ElbowRight

WristRight

HandRight

HipLeft

KneeLeft

AnkleLeft

FootLeft

HipRight

KneeRight

AnkleRight

FootRight

Count