Iowa State University
Digital Repository

Graduate Theses and Dissertations Graduate College

2010

The AugmenTable: markerless hand manipulation of virtual objects in a tabletop augmented reality environment

Michael Van Waardhuizen
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd

Part of the Mechanical Engineering Commons

Recommended Citation

Van Waardhuizen, Michael, "The AugmenTable: markerless hand manipulation of virtual objects in a tabletop augmented reality environment" (2010). *Graduate Theses and Dissertations*. 11525.
http://lib.dr.iastate.edu/etd/11525

This Thesis is brought to you for free and open access by the Graduate College at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

The AugmenTable: markerless hand manipulation of virtual objects in a tabletop augmented reality environment

by

Michael Van Waardhuizen

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Human Computer Interaction

Program of Study Committee:
Jim Oliver, Major Professor
Eliot Winer
Stephen Gilbert

Iowa State University

Ames, Iowa

2010

Copyright © Michael Van Waardhuizen, 2010. All rights reserved.

Table of Contents

List of Figures

Abstract

The AugmenTable is an augmented reality workstation intended for conceptual design and prototyping. It combines a thin form factor display, inexpensive web cameras, and a PC into a unique apparatus that has advantages similar to a head mounted display. The system operates on well-established computer vision algorithms to detect unmarked fingertips within an augmented reality scene. An application was developed to allow a user to manipulate virtual objects within the scene. This manipulation is possible through the use of three-dimensional widgets and controls that allow the user to control objects with natural fingertip motion. This thesis also documents similar previous work, the methods used to create the AugmenTable, and a number of avenues for advancing the system and the interactions it can offer users.

Chapter One: Introduction

The computing world of 2010 is noticeably affected by many trends. These trends include the use of more natural, direct interfaces, the rise of consumer-grade mixed reality systems, and the application of virtual reality to design and manufacturing processes. These trends imply that users will soon need 3D interfaces to interact with technology. This paper describes a new, unique project that attempts to unify these trends and results in an augmented reality workstation that allows users to interact with 3D virtual objects directly with their bare hands.

Direct Manipulation Interfaces

Direct manipulation interfaces, also known as natural interfaces, are those that require few or no mediating controls for interaction [1]. For example, multitouch displays, like those found in Apple iPads, allow the user to touch application content and controls directly with his or her fingertip rather than using a mediating technology like a keyboard or mouse. Direct manipulation interaction has many benefits, most notably a decreased need for training or practice in order for a user to expertly operate the interface – humans have evolved to intuitively manipulate objects with their hands. These benefits often translate into easier, more attractive, and more successful designs. Direct hand manipulation of virtual objects was shown to be faster and more intuitive than using a keyboard/mouse interface by [2]. These benefits have led entire research groups, such as MIT's Tangible Media Group, to dedicate more than 10 years to integrating manipulatable objects with virtual objects and metadata.

A number of consumer technologies, both nascent and established, aim to increase the prevalence of direct manipulation interfaces. Multitouch displays allow finger presses directly to a screen and have become ubiquitous in smartphones and other displays. Other technologies that have not seen the same market penetration include haptic response devices, such as Phantom Falcon 3D pointers or haptic response gloves. These devices may become more common and less expensive as user's expectations for direct manipulation interfaces rise.

Augmented/Mixed Reality

Another trend is the expansion of augmented reality systems. Broadly, augmented reality is the (most frequently visual) superposition of real and virtual objects or information in one environment. As a research area, augmented reality has been pursued for many years with a number of wide-ranging applications. Many of these systems have never left the laboratory due to cost or other constraints rendering them impractical. However, due to the adoption of mobile devices with powerful processors, built-in cameras, and fast internet connections, augmented reality is beginning to infiltrate the average individual's life.

A number of augmented reality applications have appeared in the Apple and Google application stores (see [3] or [4] for examples.) These applications range from spur-of-the-moment information overlays, like location guides, reviews and ratings, to games that observe the user's motions to create virtual effects. One good example is Google's *Goggle* program [5], an application that accepts photos of landmarks, books, artwork, and many other object types and then returns a Google visual search on the object.

As the public uses of augmented reality are accelerating, so are the technologies that power them. Many examples of improved augmented reality applications are here or on their way. MIT's Sixth Sense demo combines an iPhone, video camera, and pico-projector to allow a user to record and display on any surface [6]. The Skinput system creates a similar effect using the user's skin as an input device [7]. Other consumer technologies such as Samsung's transparent OLED displays [8] will one day enable a generation of hands-off, information-everywhere augmented reality. This trend has just begun.

Increased use of VR in design/manufacturing

The trend of using virtual and augmented reality to support design and manufacturing processes is not one that receives significant attention from the general public, yet is a source of new thinking about what problems VR/AR can solve. Though many systems are proprietary, a number of design/manufacturing AR systems have been described in academic papers. Kim & Dey, for example, discuss the use of augmented reality for design prototyping activities [9]. Augmented and virtual reality provides the next extension to current computer-aided design systems, providing a means to more in-depth conceptual design, review, and prototyping.

Academic literature also provides several guidelines for industrial augmented reality systems. Kim & Dey claim that immersive displays such as head mounted devices (HMDs) are important to reach the full capability of an industrial AR system. Additionally, Bleser et al state that the use of markers for hand tracking systems is not acceptable for industrial applications [10]. These criteria create a necessity for a new industrial AR design.

User Needs

These trends have two co-dependent sources: technological innovation to create business opportunities and the creativity of developers to meet real user needs with technology. However, the ongoing growth of these trends is driven more by consumer and user adoption. Users seek direct manipulation because it is quicker, easier, and more pleasant to use. Users are using more augmented reality because it is becoming inexpensive and requires less expertise or preparation. VR is becoming more important to design and manufacturing because it is providing new means of studying and creating designs.

For a system to capitalize on the trends above it must have these same properties. It should be simple to use – it should not require learning a gestural language or require the user to wear or manipulate cumbersome equipment. It must provide a new way of approaching virtual (or real) objects to enable new perspectives. Finally, it must be an inexpensive system that can be assembled without great expertise. These are the requirements for a system to effectively provide value to end users.

The AugmenTable

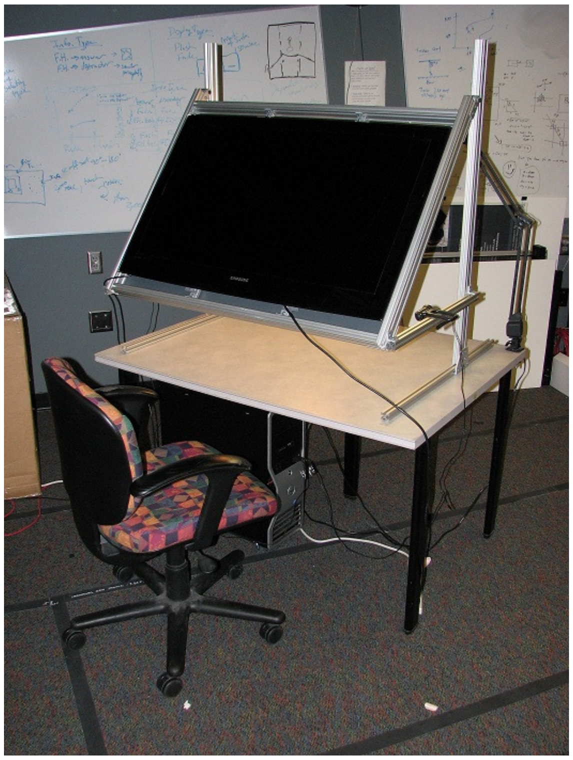
Figure 1: AugmenTable - Front

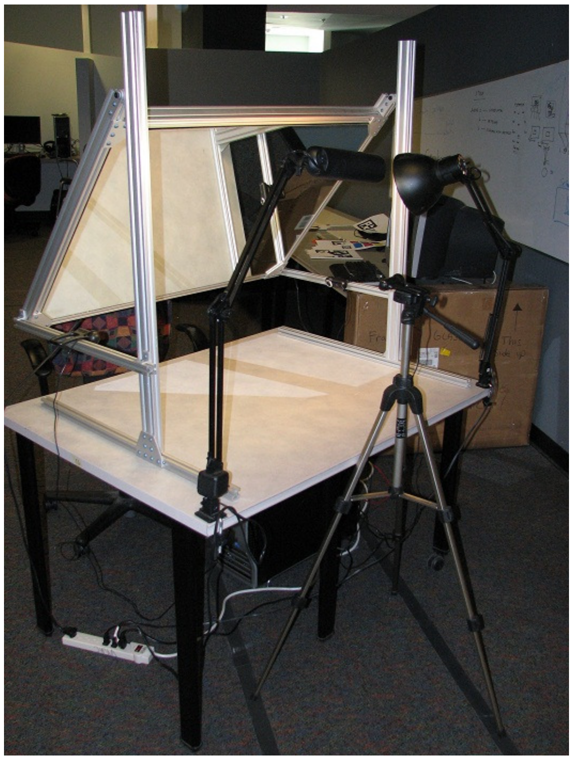
Figure 2: AugmenTable - Back

The system described in this paper provides an immersive augmented reality environment that enables a direct manipulation interface for a conceptual design process and enables new human-computer interaction. The system, called the AugmenTable, is a desk-based workstation that features inexpensive cameras, a thin display monitor to approximate a transparent display, established computer vision algorithms to identify and track a user's hands, and virtual affordances for a user to manipulate or interact with a virtual object using his or her bare hands. Furthermore, the system interaction is intended to provide direct manipulation with virtual objects that is inherently similar to the way that user's interact with real objects. This similarity enables a greater sense of immersion and suggests a number of interaction metaphors that can be directly copied from everyday life.   As a result, this system is intended to provide a test-bed for future research into three-dimensional hand-based interactions.

The apparatus provides this functionality without encumbering the user with wearable equipment. The system uses cameras and computer vision to track the hands without requiring gloves or ungainly makers. The display provides a view of the user's hands and virtual objects integrated together without necessitating a bulky HMD. The AugmenTable system has also been designed to allow both casual and collaborative use through its size.

To be effective, the system was designed under a set of constraints: the system had to account for the ergonomics of the human hand in its interaction design, it had to be real-time and avoid noticeable lag (defined by von Hardenberg & Bérard as a maximum update interval of 50ms, equating to a refresh rate of 20 Hz [11]), and had to be flexible to support a variety of application designs including multi-person collaboration. An additional requirement was for the system to be relatively inexpensive to encourage adoption.

This thesis describes the work previously done towards these goals, the methods used to realize it now, the strengths and weaknesses of the AugmenTable, and the future work implied.

Chapter Two: Related Work

The combination of augmented reality and gesture interaction is not a new goal. Many systems over the past 15 or more years have aimed to provide more natural interaction in virtual environments via gesture recognition. Each system that has been developed has its own set of strengths and limitations. Here, I will review previous work that could be considered as forming a significant branch of the larger tree that is hand tracking, gesture interaction, and augmented reality. Emphasis is given to systems published within the last ten years.

Augmented Reality

Augmented reality is the blending of sensory input from the “real world,” most typically visual information acquired from cameras or the user’s own eyes, and virtual sensory input. The virtual input can range from textual or visual information to 3D geometry such as guiding arrows or virtual objects. Most augmented reality systems today are based on computer vision techniques that identify preset markers (preregistered 2D images) in a camera image, calculate the marker’s position and attitude, and then superimpose the virtual inputs in the viewing stream.

This paper foregoes a thorough review of augmented/mixed reality literature in favor of examining integrated systems. A definitive bibliography can be found in the ACM SIGGRAPH Asia 2008 course documentation.

Gesture Interaction

Gesture interaction with computers also has a long history. Gesture interaction poses two problems: how to observe or track the hands, and how to translate the hand’s position, attitude, or motion into computer interaction.

Hand Tracking

Tracking of the hands is accomplished in one of two ways: applying some form of external accessory to the hands that is easily tracked (also called instrumented hands), or using computer vision algorithms and techniques to extract hand information from one or more cameras.

One form of accessory is fiducial markers. Fiducial markers are preset images typical of augmented reality systems such as those using HIT Lab's ARToolkit [12]. An example system using such markers is FingARTips [13]. This system requires users to wear a black glove adorned with several markers at important joints (see Figure 3.) The markers are then detected in an AR environment, allowing several direct manipulation interactions such as pressing, pointing, and grabbing. The use of fiducial markers for tracking reduces the complexity of the tracking system. However, it also limits the range of motion of the hands such that the markers must be visible at all times, restricting the angles and rotation of the hands in 3D. A similar marker-based gestures system was used by Kato et al for collaborative interaction as well [14].

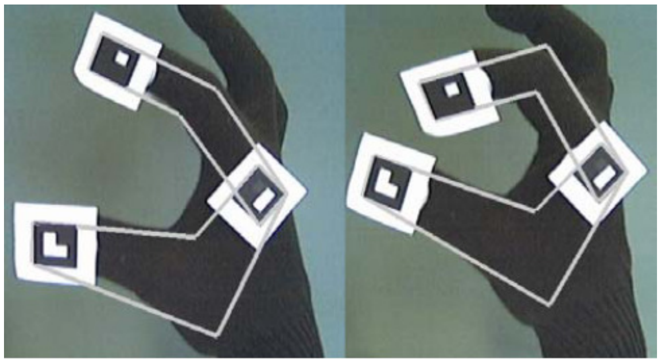
Figure 3: Simplified Hand Model of FingARTips [13]

Markers are not limited to fiducial markers for AR. Reifinger et al created a system using small markers on a glove that were tracked by infrared cameras, with a scene displayed via HMD [2]. This system is able to recognize both static and dynamic gestures (with help of a hidden Markov model.) The system supported grasping and scaling manipulations similar to the AugmenTable, but required unwieldy IR markers and specialized cameras to do so. These requirements imposed a high cost and reduced the immersiveness of the application.

Markers can provide useful information about the articulation of the hand, and so are often used in systems that create a computational model of hand geometry. Such information can also be gained from sensor equipped "datagloves." An older review of glove based inputs was performed by Sturman & Zeltzer [15]. More simply, colored gloves (as used by Keskin et al [16]) are much less encumbering and have been used to address difficulties with skin detection (discussed below.)

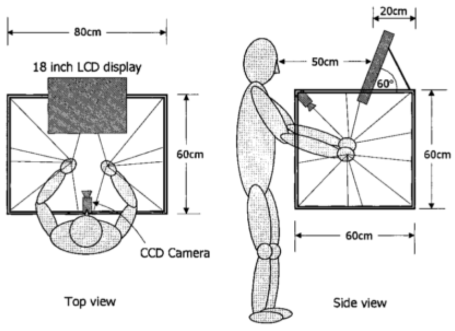

Figure 5: SPIDAR-8 Haptic AR [17]

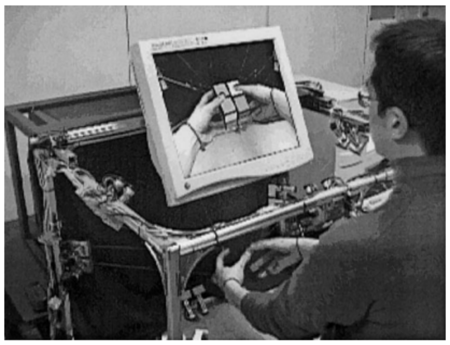

Figure 4: Ibid.

A unique tracking solution was established by Walairacht et al [17]. They describe a system where a user may manipulate a virtual object in a workspace with very real, natural hand movements. The system provides haptic feedback, enabling the user to touch and manipulate objects as if they were real. The system also tracks all of the user's fingers individually, allowing for geometric calculation of the user's perspective. This is accomplished through the use of a unique system of strings, attached to the user's hands during operation, as shown in Figures 4 & 5. This system, though enabling many capabilities, would not be practical for casual use and could likely result in a fair amount of user fatigue. Additionally, the system required numerous calculations which resulted in a lagged, slow system response.

When all encumbrances are removed, hand tracking is the province of computer vision techniques. The number of papers and techniques developed are many and myriad. Most systems utilize skin detection and object tracking algorithms (dicussed in depth below under Method.) Unfortunately, there is no “silver bullet” technique commonly accepted to detect hands. Each technique addresses some difficulties at the expense of others.

Markerless tracking of hands is not a new idea. DigitEyes was one of the earliest markerless hand tracking systems described in 1994 [18]. Four years later, Nölker and Ritter advocated markerless realtime hand tracking without using a geometric model to improve speed [19]. Though markerless tracking has been suggested and used for more than 15 years, marked tracking is still considered justifiable due to the difficulties of markerless hand recognition.

Erol et al describe the main difficulties hand tracking systems as follows [20]:

1. **High-dimensionality**. Counting all of the joints, the hand has more than 20 degrees of freedom. Reducing this through approximate kinematic modeling still leaves at least six dimensions of manipulation, not counting the hands’ positions and orientations.
2. **Self-occlusions**. From a single camera perspective, the hand has many configurations in which the nearest surfaces of the hand or fingers obscure how the rest of the hand is positioned.
3. **Processing Speed**. Real-time computer vision systems have to process a significant amount of data. Modern techniques of parallel programming and today’s hardware make this less relevant and improve formerly marginal techniques.
4. **Uncontrolled environments**. Object tracking of any kind in computer vision is made exponentially more difficult by unrestricted background and lighting conditions.
5. **Rapid hand motion**. The hand is capable of moving up to 5 m/s with 300 degree/s wrist rotation. Given typical camera frame rates of 60 fps, and tracking algorithms that typically run much slower, full tracking of the hand remains elusive.

The following systems all place restrictions on either the environment or the user's gestures in order to ameliorate the difficulties of hand tracking. The most common restrictions are uniform background and limited gesture speeds [11]. Other restrictions may be on the orientation of the hand to remove self-occlusions or to limit the tracking of the hand to two dimensions, such as on a desk surface [21], [22] , [18], [23], and [24].

These restrictions are often necessary for the systems to function, or may be implicit in the tasks the system supports. However, the more restrictions imposed on the user can render the experience less immersive and less realistic, reducing the value to the end user. As a result, most of these systems never leave the laboratory.

Hand Interaction

Erol et al categorize hand interactions into two types: gestures used for communication (in this context, to command and control interfaces) and object manipulation gestures (simulating life-like interactions, such as pointing or pinching) [20]. The former tends to utilize static hand poses or motion patterns which are then interpreted as commands. The latter may include poses and motion patterns, but also frequently include direct tracking of the hand or fingertips.

Pose and motion pattern recognition is developed either in creating three dimensional models of the hand through inverse kinematics, or in partial pose recognition based on 2D appearance [20]. Model based/inverse kinematic reconstruction is not discussed in this thesis. A useful review can be found in [20].

Pose recognition is separated into tasks of identifying the hand in one or more images, extracting relevant features, and passing them to a gesture classification system. Such a system uses statistical methods to determine the pose or gesture from a previously trained library. The variants of this technique are very popular for hand interaction, and were used by [11], [21], [25], [26], [27], [28], [29], and [30].

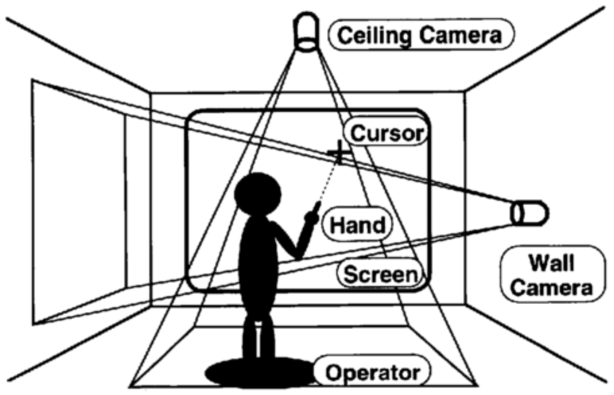

Figure 6: Example ray-casting system sytem [33]

Pose estimation is inherently limited. As pointed out by Petersen and Strieker [31], a system can only identify poses for which it has been trained. As a result, pose estimation based systems put a lot of focus and development on gesture classification. Classification is a step performed through neural networks, hidden Markov models, support vector machines, or other statistical/machine learning methods. A comparison of such systems is provided by Corso and Hager [32].

Statistical methods are less useful in systems that have object manipulation goals. In this case, systems must track the user with significant precision that typically isn't available from general recognition techniques. This is especially noticeable in one older technique of object manipulation known as "ray-casting". Ray-casting is the extrapolation of a finger or arm direction onto a surface, e.g. the system by shown in Figure 6 [33]. Ray-casting requires as precise measurement of finger orientation as possible through robust feature tracking.

Ray casting is one of several interaction methods broadly aimed at providing virtual pointers. Virtual pointers were found to be acceptable for selection tasks, but not for further manipulative tasks [34]. Addtionally, Mine et al found several advantages of working within arms' reach: more direct mapping between motion and response, better precision of movement, and better visual cues of parallax and stereopsis [35].

For our goal of direct virtual object manipulation, a more natural form of interaction is required. Poupyrev et al describe another broad category of 3D interactions in virtual environments: virtual hands [36]. Virtual hands are a metaphor for interaction where the user “touches” a virtual object for interaction. Similar taxonomies of hand interactions are described in [37] and [38] which include at-a-distance interaction (like ray-casting) and local selection (like touch), but also manipulation through grabbing, virtual manipulators, scaling/zoom interactions, and virtual menus.

In either taxonomy, a number of interactions exist. Some mimic actions descending from 2D interactions a la point and click. Early systems, such as one developed by Rehg & Kanade [18], tracked hands to develop a “3D mouse.” This led to general “picking,” or selection interactions, such as one described in [39]. Picking is also used for application control in place of gestures (through using real or virtual controls such as buttons) or viewpoint manipulation [38].

Other interactions arise from the development of native 3D interactions. Natural, realistic hand interactions such as grabbing, pinching, and bumping are new to 3D environments. When these are not possible, another class of interactions use virtual controls or widgets that are designed for 3D interaction. C. Hand found that well designed widgets can be less damaging to the feeling of directness than more abstract or invasive interfaces like gestures or physical controls [38].

Interactions in virtual or augmented spaces have to be designed to address the weaknesses of hand detection. Mine et al describe the issues with object manipulation in virtual environments as follows [35]:

1. Lack of haptic feedback: Humans depend on the sense of touch and weight for precise interaction with the real world.

2. Limited input information: The multimodal inputs of object manipulation in the real world (tools, spoken communication, measuring, etc.) are restricted or unavailable in virtual/augmented environments.
3. Limited precision: Most hand interactions in virtual environments have "boxing glove" precision; little or no fine motor control is available.

The interactions used for the AugmenTable attempt to address these shortcomings through use of arms-length interactions using superimposed 3D widgets, as described in the Applications section.

Comparable Systems

Several systems have previously been developed with the goal of virtual object manipulation in an augmented reality environment using markerless hand tracking. However, all do not entirely reach the goal of intuitive, unencumbered fingertip manipulation of virtual objects.

The apparatus for such a system is fairly well agreed upon. Erol et al point out that multiple cameras are necessary for object manipulation without using markers, or for allowing two handed interactions [20]. They also mention that combining multiple views to establish correspondences across cameras and 3D features has not been explored well. Abe et al used vertical and horizontally oriented cameras to develop a 3D position of a single finger, enabling 3D rotation and translation when combined with pose recognition based commands [21]. A similar multi-camera system was developed by [40] several years prior. These systems aren't augmented reality, though, since they do not integrate real objects with virtual objects.

An early tabletop AR system was developed by Oka et al called EnhanceDesk [41]. By using a color camera and an infrared camera, they were able to track fingertips through a combined approach of template matching and Kalman filtering. This system only tracked the fingers on the surface of a desktop and today would be more effectively implemented through multitouch surfaces. That said, this system's apparatus and methods have been applied to the 3D problem by subsequent systems, including the AugmenTable. A similar system with similar restrictions was more recently suggested in [42].

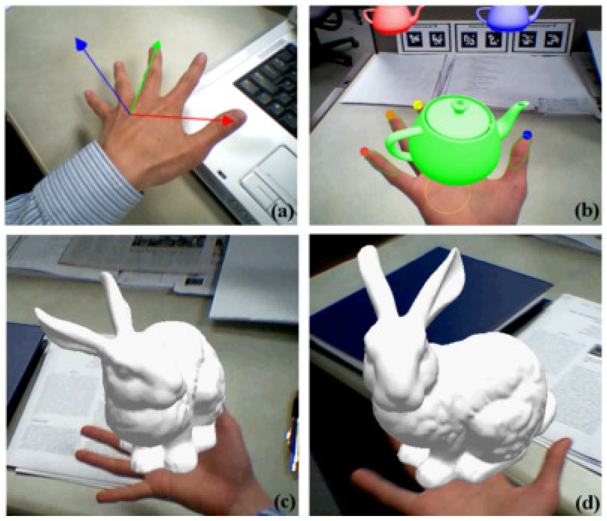

Figure 7: HandyAR Snapshots [43]

A more capable system described by Lee & Hollerer shares many of the same goals as this proposed system [43]. Based on previous "HandyAR" work [44] and markerless AR research [45], Lee & Hollerer use an optical flow algorithm to track an outstretched open hand. It determines the finger locations based on the thumb location, then uses pose estimation to determine the orientation of the hand (see Figure 7.) The finger positions are established through an initial calibration, then tracked using Kalman filtering. This enables a coordinate system or model to be matched to the user's hand as though the hand were a 2D fiducial AR marker. The recent extension to this work enabled the tracking of desktop surfaces for an additional AR surface, as well as a "grabbing" gesture through breaking the tracking of the outstretched hand in favor of a closed fist. This allows free manipulation of a virtual object with a user's hands, but at the cost of losing natural gestures such as pointing, grabbing, or pinching that deform the hand. This system is unable to track motion through self-occlusion as well. Rotating a virtual object with the hand can only be done within a limited range of motion. A model cannot be rotated to its side, for example.

A more effective system was described by Song et al [46]. This system tracked an individual finger in a 3D augmented reality environment. The authors created a set of interaction methods, combined with a physics engine, to provide a unique object manipulation system. The authors also ran a user study finding bare hand interactions to be more intuitive and pleasant for users than keyboard and mouse interfaces. Using a single finger, however, is pretty limiting and does not match natural human object manipulation.

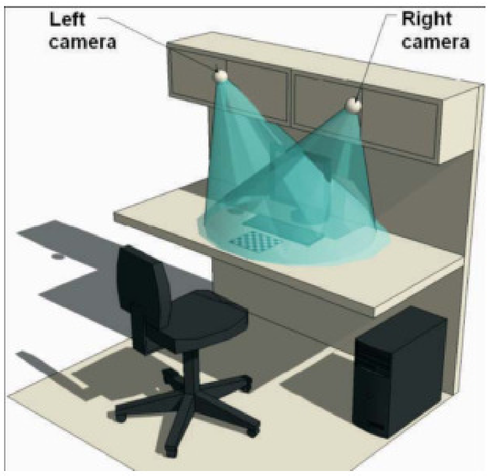

Figure 8: Similar Workspace Concept [47]

Of all the systems and prototypes reviewed, the one developed and described by Kolarić et al bears the most common ground with the proposed system [47]. Like the AugmenTable, theirs uses free, unmarked hand movements to manipulate virtual 3D objects. They use a computer vision system that tracks the hands in a stereo camera setup and uses the Viola-Jones tracking method paired with skin color histograms for detection. To manipulate objects, the authors define a set of hand poses for command communication: select, open, and closed, which are mapped to functions such as select, translate, and rotate.

This system (shown in Figure 8) bears the same functional purpose as the proposed AugmenTable system. However, the AugmenTable tracks fingertip points for higher controllability, does not use learned hand gestures in favor of developing intuitive manipulation widgets, and uses an apparatus that allows for the hands and virtual objects to inhabit the same perceptual space. Additionally, the proposed system supports multiple hands, multiple fingertips as well as rotation of the hand through arbitrary angles – a rare combination in the field.

All of these comparable systems use either head mounted devices or regular desktop displays. HMD systems limit the user's field of view, can become uncomfortable, and often feature a screen that is too dim [17]. Desktop displays are not immersive; in the case of Kolarić et al, the user can see his or her hands in the workspace in front of the monitor.

Commercial equivalents of comparable systems also exist. OrganicMotion [48] offers real-time, markerless tracking of human actors within a specific volume down to millimeter accuracy. Microsoft's impending Xbox Natal project offers similar tracking in anyone's living room. In the former case, however, it is unclear if the system provides tracking of finer finger motions or is appropriate for integrating real and virtual objects in real time. In the latter case, the technology has not been released and it is not known how capable the system will be.

Suprisingly, the use of virtual widgets for manipulation does not seem to be well explored in the tabletop AR literature. Song et al, as noted before, use a physics engine combined with a virtual "fishing line" widget [46]. A user selects a virtual object with a finger touch, and then a virtual line is extended from the object to the user's fingertip, allowing for physical control of the virtual object. No system was found through literature review, however, that provided a virtual manipulator with affordances for hand manipulation of virtual objects. In contrast, I believe this is a useful line of inquiry for tabletop AR applications and have developed a direct manipulation interface based on widgets for the AugmenTable. Furthermore, I hope that the AugmenTable will provide a means of prototyping and testing further 3D interactions, both widget based and otherwise.

Chapter Three: Method

The proposed system combines a number of well established computer vision techniques with a novel, inexpensive apparatus. This section details the apparatus and algorithms used and how they integrate together.

Apparatus

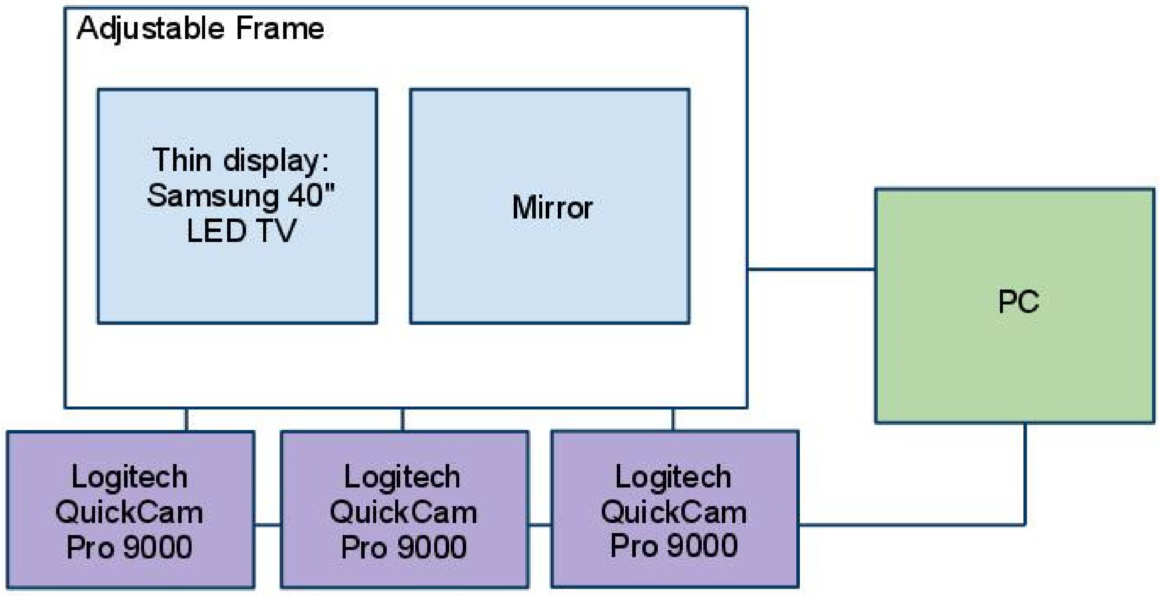
Adjustable Frame
Thin display: Samsung 40" LED TV
Mirror
PC
Logitech QuickCam Pro 9000
Logitech QuickCam Pro 9000
Logitech QuickCam Pro 9000

Figure 9: AugmenTable Apparatus

A novel element of the AugmenTable system is its ability to provide a near immersive augmented reality experience without requiring the user to wear or hold any devices. The apparatus places a thin-form factor display raised at an angle to face the user (see Figure 1 & 2 above.) The user may sit or stand in front of the display (depending on the height of the table on which it rests) and reach his or her hands underneath and behind the display. A mirror is mounted to the reverse side of the display, reflecting an image of the user's hands outwards towards a camera mounted on a tripod. The television, mirror, and additional tracking cameras are all mounted to an adjustable, light weight aluminum frame. The frame has been designed so as to allow adjustment to the height and angle of the television relative to the user. The display and each camera are connected to a PC equipped with multiple core processor(s). For this working prototype, a Samsung 40" LED TV (UN40B6000VF), three Logitech Webcam Pro 9000 cameras, and a Dell workstation featuring an Intel Xeon X5570 quad-core processor and a Nvidia Quadro FX 4500 graphics card were used. Depending on display size and computer power, a similar functional apparatus could be constructed for less than $4,000.

The AugmenTable provides an immersive experience by simultaneously hiding the user's hands and displaying them in a scene with virtual objects. To be fully immersive, augmented reality should provide visual-spatial, proprioceptive, and haptic cues. Haptic feedback cannot currently be simulated without requiring the user to wear a device, such as in those used in [17] or [13]. Proprioceptive feedback is the mind's self-awareness of the body and is generally a very weak, easily fooled sense - research has shown that human’s proprioception is dominated by the visual sense [49]. Visual-spatial cues are the visual phenomenon the brain uses to identify where it is in space relative to other objects. These cues include transparency, occlusion, size, shading gradients, and cross references such as shadows among others [50]. Overall, this system is limited to providing relative size and occlusion cues, masking proprioception, and very simplistic haptic cues when a virtual object is placed against the tabletop surface.

This apparatus proves more immersive than many other current augmented reality experiences. First, the experience of this system is more immersive than a traditional desktop monitor. By hiding the hands from the user and showing a representation of the hands within the virtual world, the user does not have to resolve seeing his or her hands in front of him or herself with also seeing his or her hands in a different location. This advantage may not be largely significant given human ability to map control of the body to manipulation of distant objects, as typified by using steering wheels, game controllers, and laser pointers for example.

More commonly, AR is provided through hand held devices such as mobile phones or tablet computers. These systems do not typically include any part of the user within the augmented reality "window", due to the user having to hold the device in place. In research, the HMD is the most frequently used device for experiencing augmented reality.

The AugmenTable offers both pros and cons compared to these two standards. A mobile phone/computer can provide augmented reality anywhere the user takes the device; the apparatus described here is stationary. An HMD provides a direct angle of view for the user to experience augmented reality; the apparatus described here will most likely display an angle of view slightly different than the user's direct gaze due to the stationary camera. HMDs also provide stereo viewing capability that is currently lacking in the AugmenTable. However, this apparatus does not require the user to carry a device, provides a large field of view that can eclipse the user's peripheral vision, and does not require the user to wear heavy equipment on his or her head. Finally, this system does not require any markers to be worn on the user’s hands. This improves the illusion of direct manipulation for the user and reduces the overhead of starting to use and learn the system.

This apparatus provides a good baseline immersive experience for an augmented reality workstation. Improvements are described in the Future Work section that could improve the experience even further.

Software Libraries

Most interesting software projects today would not be possible without having powerful libraries to stand upon. The AugmenTable relies on three libraries for a significant number of tasks; each was essential, and a significant effort was made to integrate them together.

First, as mentioned above, is the ARToolkit library [12]. ARToolkit is used here for a number of important initialization steps: determining camera distortion parameters, searching a 2D image for a stored 2D marker pattern, and calculating the inverse camera matrix based on the size and orientation of the detected marker. ARToolkit does not perform all of these steps perfectly, unfortunately. The 2D marker detection can be vulnerable to false positives. In this case, the system requires a recalibration before use. Currently, the system uses ARToolkit version 2.72.1. ARToolkit should be replaced with a more reliable augmented reality library in the future.

Second is the ubiquitous computer vision library OpenCV. OpenCV provides access to the raw camera image feeds, matrix calculation operators, and important 2D image processing algorithms such as color histogram matching, morphology operations, and contour detection. Each of these algorithms is discussed in depth below. The AugmenTable currently uses OpenCV version 2.1.0.

The third and final library used in the creation of this system is OpenSceneGraph. OpenSceneGraph is used to create and manage the 3D scene that comprises the augmented reality environment. It handles all three dimensional models, lights, and events including model intersections necessary for all interactions. OpenSceneGraph 2.7.2 is the current version used in this system.

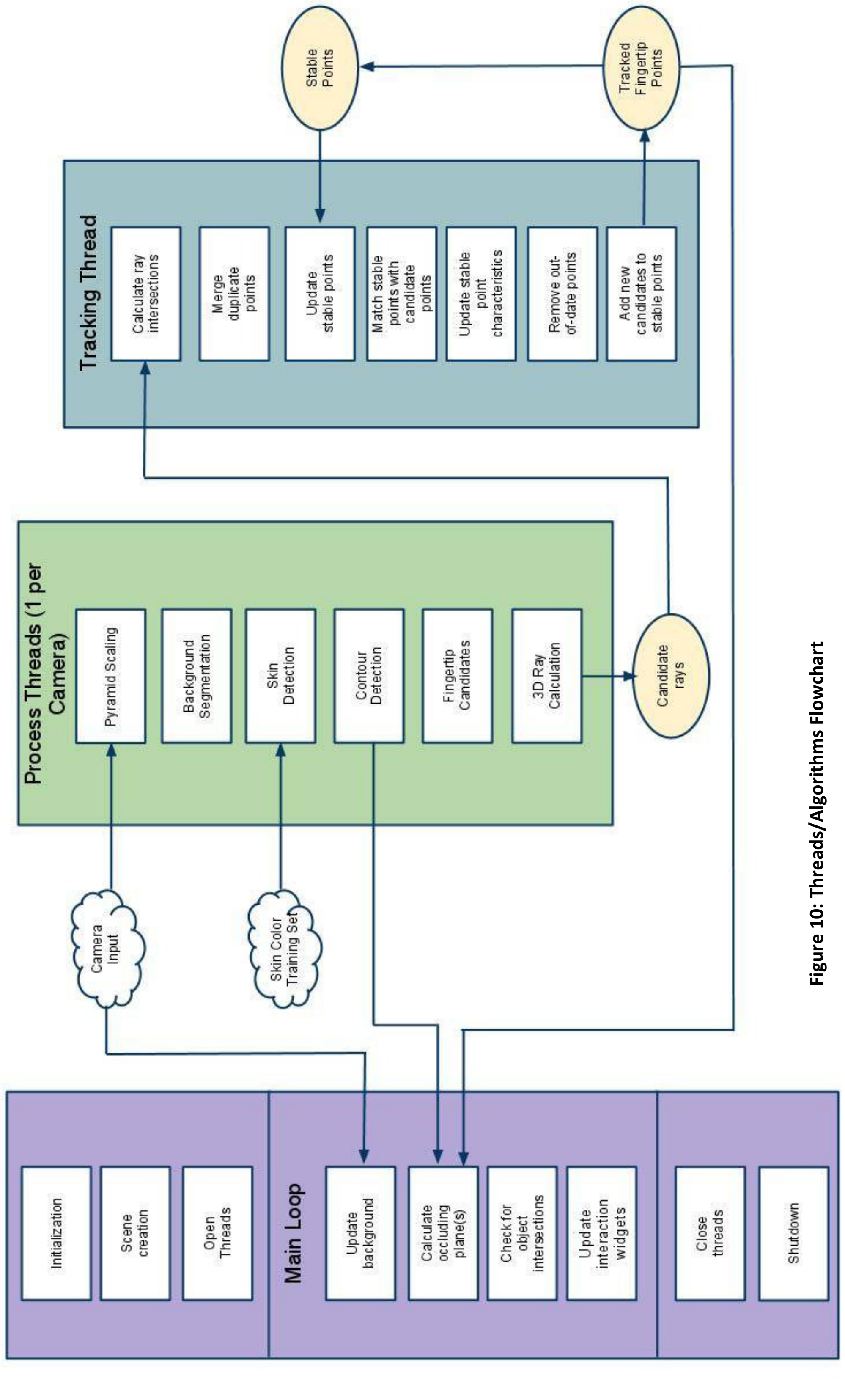

Figure 10: Threads/Algorithms Flowchart

Algorithm/Process

Figure 10 shows the algorithmic steps for identifying and interacting with hands within a virtual three-dimensional scene. The process broadly proceeds as follows:

1. The cameras and AR scene are initialized.
2. Input feeds from multiple cameras are reduced in size for processing.
3. The image backgrounds are segmented out of the frame.
4. Skin pixels in the foreground are detected and filtered.
5. Contours around the hand shapes are created, and then reduced to a polygonal approximation.
6. The outermost (convex angle) points are identified as candidate fingertips.
7. The candidate points and transformed into 3D rays that intersect the scene.
8. Each ray is tested against all other camera's rays for intersections.
9. Current frame intersections are tested against a set of stable, tracked 3D points to update the scene.
10. Tracked points are tested for intersections with scene objects to create interactions.
11. Tracked points are used to determine where in space the contours identified in step 5 are, so that they may be used for occlusion.

All code for the above steps is included in the Appendix.

Initialization

The process is broken into multiple, parallel threads. This enables the system to function in real time on modern multi-core processors. The process begins with initialization: in addition to typical variable and memory initialization, each camera calculates its position in space prior to starting image processing. Using ARToolkit, each camera searches for a predefined marker in its field of view. Upon locating the marker, its size and orientation are compared to the known marker parameters. This comparison enables ARToolkit to calculate the distance and orientation of the marker compared to the camera in matrix form. The inverse of this matrix results in the camera's position and orientation relative to the marker (see Figure 11.)   Each camera's viewport, projection, and model view matrices are calculated in this way and stored for future reference.

After calibration, the ARToolkit marker is extraneous; the system currently does not attempt to update the camera positions unless the user manually instigates a reset. This is due in part to an attempt to reduce the computational complexity of each frame update,

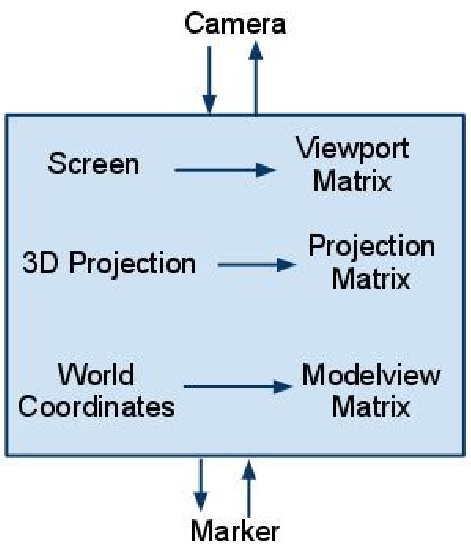

Figure 11: Relationship of Camera and Marker Transformations

but also because the basic ARToolkit is not capable of the parallel processing necessary to operate across multiple threads. ARToolkitPlus and other subsequent AR libraries have support for parallel processing, so if CPU bandwidth is available it may be possible to update the camera position on the fly, making the system more robust to movement and vibration.

ARToolkit is valuable in that it provides camera calibration without depending on epipolar geometry calculations. Epipolar geometry is comprises a series of calculations needed to correspond a 3D point in several 2D views. For example, other systems may calibrate through matching easily identified points like a torchlight [30]. Using ARToolkit for this purpose is not uncommon; other systems such as [51] have used it in a similar fashion.

Following the initialization of each camera, the scene (rendered through OpenSceneGraph) is created and the processing threads are executed. A thread is created for each camera to perform image processing in parallel with one additional thread to perform 3D calculations and point tracking (see Figure 10 above.) The incremental steps for determining fingertip positions in space are described in order.

Background Segmentation

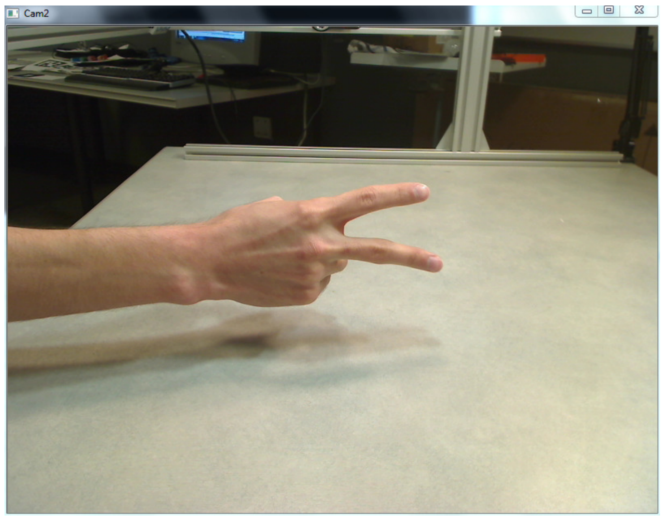

Figure 12: Example Input Frame

When a processing thread receives a new frame (example Figure 12) from its camera, the first step is to separate the background and foreground elements. Background subtraction is a very common operation in image processing; a number of techniques are described in [52]. The most basic background subtraction technique is frame differencing. Frame differencing is performed by storing an image of the static background and then comparing each incoming frame to the saved background. If a pixel's color, as measured by the RGB values, differs from the saved background's pixel color by a preset threshold, the pixel is labeled as a foreground pixel. If not, the pixel is part of the background.

The frame differencing method of background subtraction has several weaknesses. It is very susceptible to changes in lighting - even small changes to the ambient lighting conditions can alter a pixel's color sufficiently to be a false positive. Similarly, this method is vulnerable to camera noise, which can alter a pixel's color even with constant lighting.

Finally, frame differencing does not perform well when a moving object (which would normally be part of the foreground) has a color that very closely matches the background.

To address these shortcomings, a number of other methods have been developed. One such method is background averaging. In background averaging, the system accumulates a mean and a variance for each background pixel. When a foreground object moves into frame, it is classified as such if its color falls sufficiently out of range of the median, often between one or two standard deviations. This method is more resilient to regularly changing backgrounds due to lighting or gradual movement. Another method is to develop a "codebook" for background segmentation [53]. Similar to averaging, this method determines a range of values for the background over time, but a codebook stores an arbitrary number of ranges for a particular pixel. The codebook method is thus more capable of recognizing backgrounds that have a low number of discrete states.

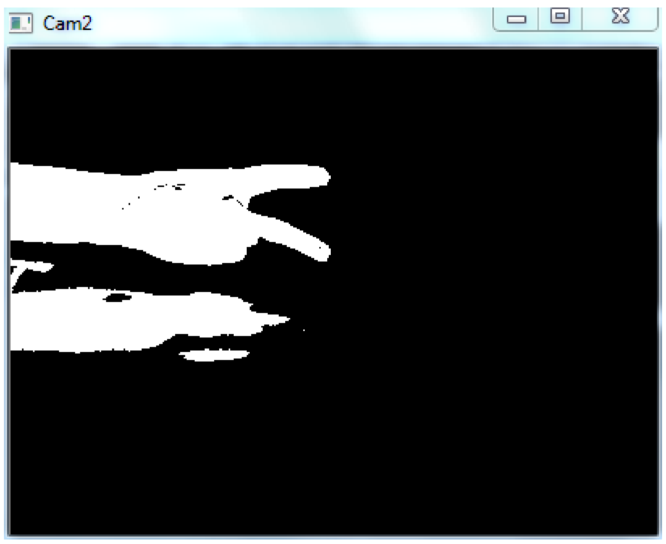

Figure 13: Example Foreground Mask

Despite the advantages offered by other methods, basic frame differencing was used for this system. The weaknesses of frame differencing are attenuated because the system is not intended for use "in the open" where backgrounds fluctuate, and also because the system relies on multiple image processing steps later to further refine the image. Frame differencing is used because of it is the fastest, least resource intensive background segmentation method. To be even quicker, the input image from the camera is reduced by 75% in size through Gaussian pyramid reduction before segmentation. Each camera has to perform a number of computationally intensive processing steps; background segmentation is intended primarily to reduce the search space for these subsequent steps. As a result, a lightweight implementation of frame differencing was chosen over the more intensive codebook method. Background averaging was tested, but discarded due to issues with ghosting as foreground objects were rolled into the background averages. Codebooks were not pursued due to higher processing requirements.

One possible improvement to the background segmentation would be to convert to HSV color space (further discussed under Skin Detection) and segment based on difference only within hue or saturation values. This approach was fundamental to finger tracking used in [54]. This was attempted for the AugmenTable, but yielded worse results than RGB color space.

Skin Detection

After the foreground has been identified (which may include some false positives), the system must identify the user's hands through skin detection. Skin detection is the comparison of a particular pixel's color to some preset value that matches the color of human skin. As pointed out by [55] and [56], this has several problems: the colors seen by a camera are affected by ambient light, movement, and other contextual factors; different cameras produce different colors for the same object under identical conditions; and finally, the color of skin can vary widely from person to person. As a result, a number of different approaches have been developed for skin detection.

The most basic method of skin detection is color thresholding. In most images, individual pixels are values from 0 to 255 in red, green, and blue channels (RGB.) Thresholding is accomplished by finding the RGB levels common to shades of skin. For instance, the red hue may be 30 to 50% of the color in skin with corresponding values for green and blue. If a pixel matches this profile, it is included in the process output. Thresholding is very fast, having O(n) calculations to identify skin pixels, but it is vulnerable to every issue identified in the previous paragraph.

A common improvement to skin detection systems is to use a statistical distribution of skin color. In a study of more than 1,000 images of people, skin color was found to have a normal distribution in the RGB color space. Yang et al [55] were then able to track a variety of skin hues through linearly adapting a Gaussian model based on the changing environment. These Gaussian mixture models were also utilized by Kurata et al [57]. However, Jones & Rehg [58] found that simpler histograms performed better (in training and operation) than Gaussian mixture models of skin color.

Histograms have been shown to be a reliable and powerful means of identifying skin pixels within an image. Jones & Rehg conducted a thorough review of skin detection methods built on a dataset of thousands of images from the internet and found histograms to be optimal. They further determined that a bin count of 32 yielded optimal results. Based on this data, a pretrained histogram of 32 bins was used for skin detection in the AugmenTable.

Histograms are better at handling variations in color, but still have room for improvement. Jones & Rehg also describe a modification to improve histogram detection. Instead of a single skin histogram, they utilized two histograms: one for finding the probability that a pixel was skin colored, and one for finding the probability that it was not. They then determined whether a pixel was skin colored through the following equation:

$$\frac{P(RGB\ |Skin)}{P(RGB\ |Not\ Skin)} \geq \Theta$$

where P(RGB | Skin) is the probability a given pixel is skin colored based on the skin histogram, P(RGB | Not Skin) is the probability the given pixel is not skin colored based on the histogram of all not-skin colors, and theta is a cutoff threshold. This method was attempted for this project but was found to yield less robust results than using the skin color histogram alone. This may be due to a desktop environment that has many white and gray hues that match Caucasian skin under fluorescent light.

Another possible reason for my difficulty with Jones & Rehg's method is their use of the RGB color space. A fair amount of literature is dedicated to the exploration of color spaces used for skin detection. The default color space of most images is RGB, but images can be converted to several other different color spaces though, most notably Hue, Saturation, and Value (HSV).

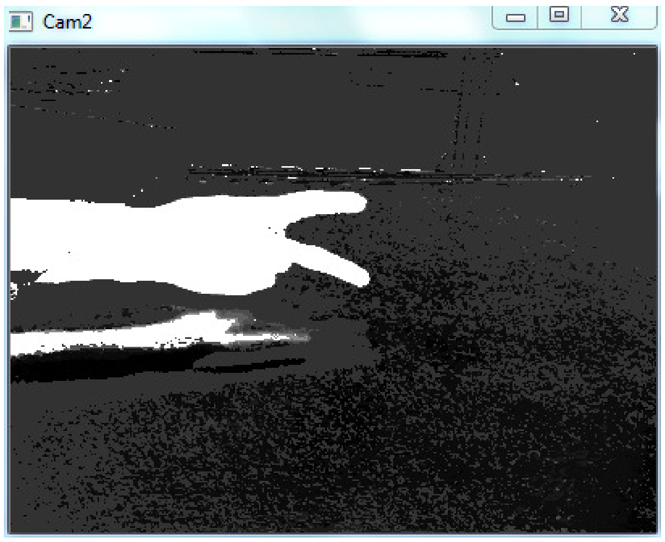

Figure 14: Raw Skin Detection (HSV histogram)

HSV has been found in several studies, [59] for example, as being preferable to RGB color space for tracking skin in hands and faces. Even the hue alone has been suggested as being sufficient and preferable for tracking skin [60]. However, using a static pre-trained histogram in this project found better skin detection results through employing both hue and saturation channels. Further analysis of color spaces for skin detection can be found in [61] .

HSV has not always been found to be better than RGB in tracking systems. Kolsch & Turk found worse performance from HSV with respect to RGB, partly because due to its CAMshift algorithm's reliance on RGB color space [62]. Dadgostar & Sarrafzadeh even developed a hand tracking system that disregarded hue entirely and focused on gray levels [26].

Despite these reservations, hue-saturation histograms have been used successfully by a number of systems, though, such as those described by [47] and [22]. A more thorough description and analysis of color spaces for skin detection, as well as the methods that use them, can be found in [56].

Static histograms like the AugmenTable's are still vulnerable to lighting/background changes. *Adaptive* histograms have been found to be more robust to lighting changes or similarly colored backgrounds. One method of adaptive histogram use was described by Dadgostar & Sarrafzadeh as follows:

1. Train an initial histogram with a priori color data.
2. Detect skin pixels in a current frame through motion detection and use of the initial skin color histogram.
3. Calculate a histogram of the pixels detected in #2 and combine with the initial histogram with some weighting assigned.
4. Calculate minimum and maximum thresholds for matching such that the thresholds cover 90% of the new histogram.

An effort was made to provide this system with an adaptive histogram to reinforce detection against illumination changes. This effort proved unsuccessful as the adaptation included some colors outside of the original histogram range, resulting in runaway detection of wood and desk surfaces until virtually all pixels were detected as skin. This result is not an indictment of the method described above. With more time, this method could prove more successful than a static histogram and ought to be pursued.

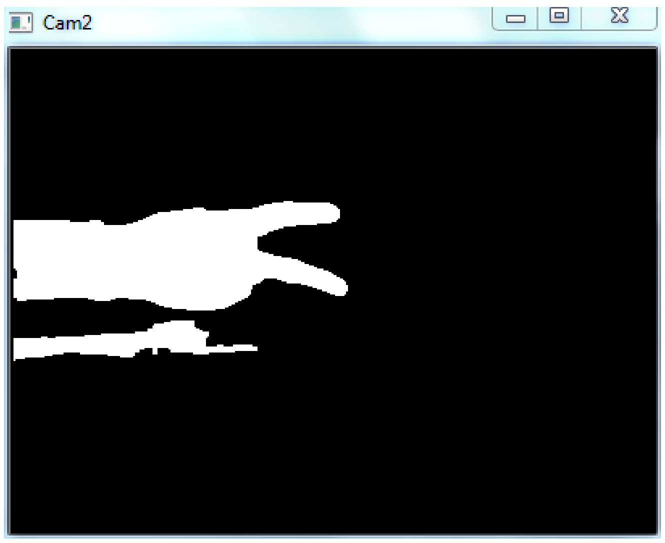

Figure 15: Filtered Skin Detection

Following segmentation of skin color, the mask is processed with thresholding and morphological filtering to reduce noise. This results in the connected components pertaining to the users hands. The results of these processes are shown in Figure 15. The results of this process are not perfect. In Figure 15, some of the hand's shadow reflected on the tabletop surface is detected as skin. This false positive is acceptable, however, because later correspondence of cameras is unlikely to propagate the false positive into 3D.

Fingertip Detection

Detecting fingertips is a sub-problem of tracking hands overall, and so has a number of approaches in literature. One popular method described by von Hardenberg & Bérard [11] refined an early fingertip detection method by Fukumoto et al [33]. Von Hardenberg's algorithm begins in a familiar way using background segmentation to identify a region of interest containing the user's hands. The algorithm then processes the following steps:

1. Determine a set of candidate points (somewhat spaced out to avoid duplicate matches for a finger).
2. Define a circle of radius r around the candidate point; if the circle is filled with skin pixels, continue.
3. Define a square around the candidate point, slightly larger than the circle from Step Two. Count the number of skin pixels that lay on the perimeter of the square.

4. If the pixel count from step 3 is greater than a defined minimum threshold and less than a defined maximum threshold, the candidate point identifies a fingertip (Figure 16.) If the count is below the minimum threshold, it may be noise or poorly positioned; if the count is above the maximum, it may be the middle of a finger or the palm.

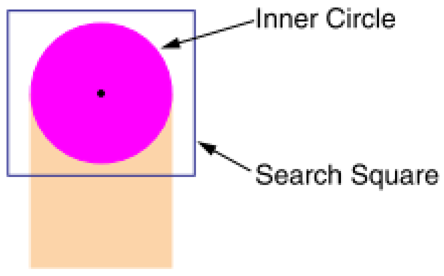

Figure 16: Fingertip detection [11]

This method has several advantages in that it precludes an additional hand search step and it can provide finger orientation information (depending on which part of the square perimeter matched skin pixels). The authors also claimed it to be one of very few systems able to track several fingertips simultaneously. As a result, this and similar methods were used in a number of successful projects such as [54], [24], [42], and [46].

However, this method is not sufficient for the AugmenTable project. Von Hardenberg's system required the user to keep his or her hands at a relatively constant distance from the camera. This criterion is necessary for the various finger size thresholds to be accurate. However, the system described in this paper must support hands at a range of distances from the camera, anywhere from a few inches to several feet. Burns & Mazzarino suggested that the finger size could be calculated as proportionate to the overall size of the hand [63]. However, this would require a relatively constant orientation of the hand to provide accurate relative measurements. This requirement also renders this method unusable for this particular system.

Other methods used to track fingers and fingertips include parallel edge detection with smooth gradient shading [31], reference templates similar to von Hardenberg's [64], Hough circles [63], as well as subtracting the palm away from the hand through a series of morphological operations [28].Many of these methods have similar restrictions as von Hardenberg's.

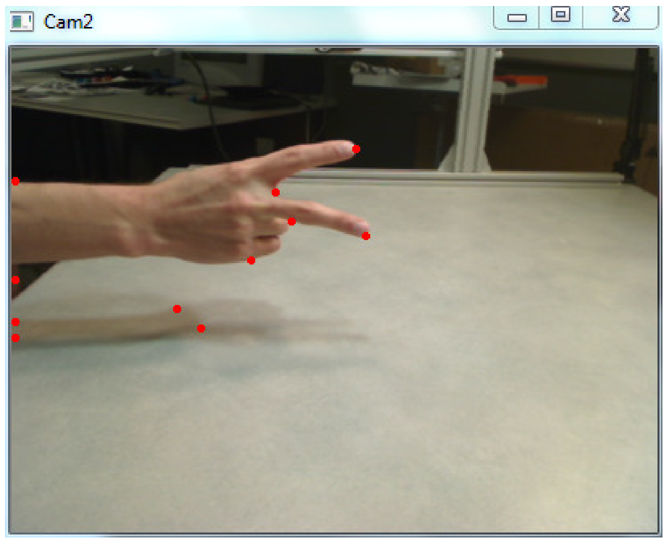

Figure 17: Polygon Vertex Approximation

Instead, this system relies on an older method of contour detection. OpenCV has a contour retrieval method based on the work of Suzuki and Abe [65]. Using the binary image resulting from skin detection above, OpenCV determines the edges of all the binary shapes as a sequence of pixels. In this particular case, the shape is assumed to be solid with no contour holes, so only the most extreme outer contours of the shapes are calculated. After the outermost contours are detected, the pixel set is reduced to a set of polygon vertexes using the Douglas-Peucker (DP) approximation [66]. The result of this approximation is shown in the Figure 17. This method is similar to that used by Chen et al [22] for a model-based hand system.

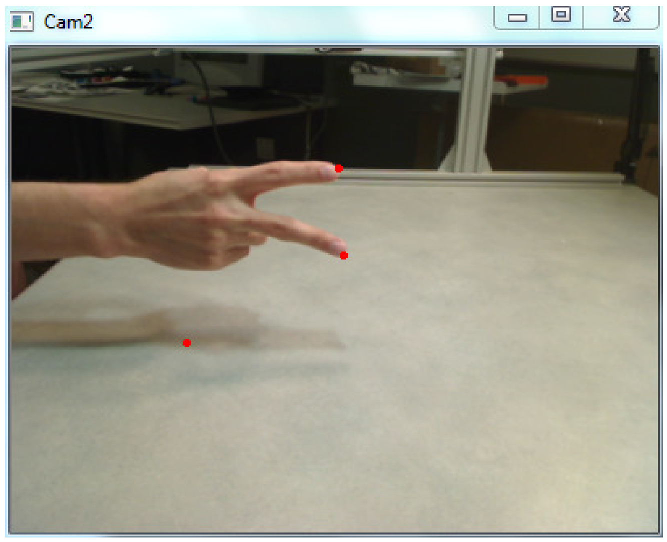

Figure 18: 2D Candidate Points

As shown in Figure 18, the polygon approximation of a hand contour enables simple geometric identification of the finger tips. To handle multiple hands, every contour above a size threshold is approximated with a polygon. Each point in the polygon is tested to determine if the point is farther away from its polygon's centroid than the two points immediately next to it. The points that pass this test are identified as candidate points.

In other words, the process selects any convex points on the contour as a candidate fingertip. Unfortunately, this can create false positives around the wrists and other knuckle bones. These false positives are not unmanageable, though, due to the physical constraints of the apparatus used and the interaction design of the applications. The user is not likely to attempt interacting with a virtual object with anything other than his or her fingertips. False positives are not expected to cause undesired interactions.

2D to 3D

Once a set of two dimensional fingertip points have been identified, the next step is to calculate the *three* dimensional position of the fingertip. The AugmenTable accomplishes this task through ray intersection. During the initialization step described above, each camera identifies its position in space relative to an augmented reality marker. With this information, and information about the intrinsic camera characteristics, a modelview,

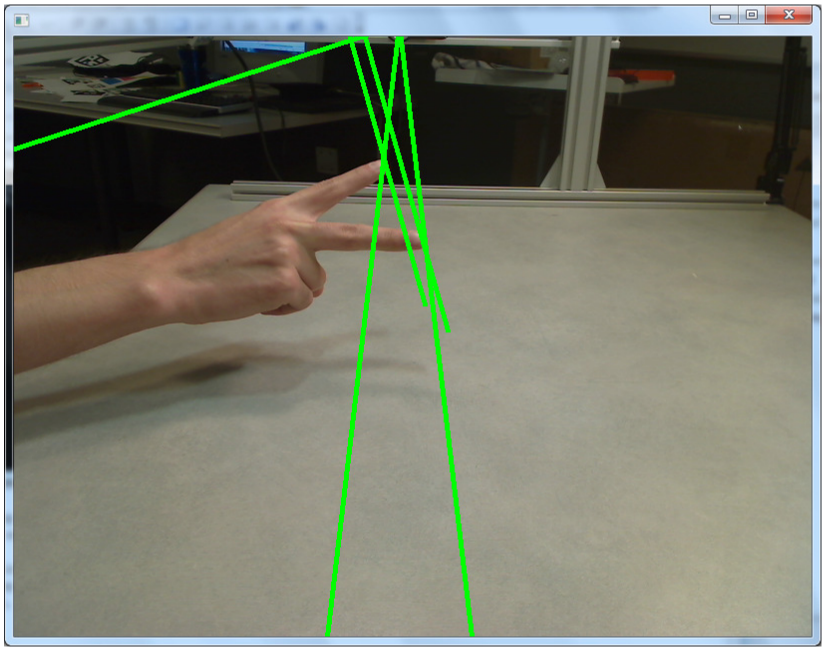

Figure 19: Ray-casting for 3D point calculation

projection, and viewport matrix are constructed for each camera.

Using the camera's matrices, the 2D candidate points are calculated at the near and far plane of the camera's view volume. These two points are the endpoints of a ray segment that projects through the scene. A ray is developed for every candidate point for every camera (see Figure 19.) The apparatus is designed in such a way that the camera view volumes intersect to create the working volume behind the display.

Epipolar geometry is often required to find the correspondence among cameras with different viewpoints (see [67] for a review). However, as described in Software Libraries, the calibration of camera's through ARToolkit provided an alternative means of acquiring correspondence without additional calculations.

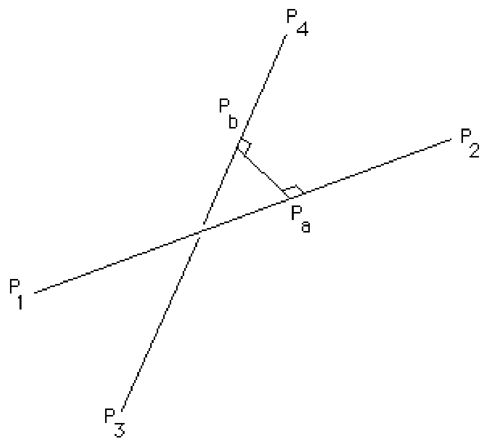

Figure 20: Closest distance between 3D lines [68]

To calculate the 3D fingertip points, a separate intersection testing thread calculates how close each ray from each camera passes to all other cameras' rays. A mathematical explanation of the algorithm can be found in [68]. If two rays pass within a set threshold distance of each other, a 3D fingertip point is determined to exist at the midpoint of the shortest line segment between the rays. Cross checking all rays has an algorithm complexity of $O(n^y)$ where y is the number of cameras, which is fairly poor performance. However, the number of candidate points, n, is strictly limited to the number of convex skin points in the camera's view. This number could be as much as forty, but in practice the number is typically less than 10. The number of cameras is also expected to be low. As a result, these calculations do not consume significant processing resources.

Currently, this system uses a threshold of 4 mm to determine if a pair of rays intersects. Ideally, the system would only consider rays that intersect with a ray separation distance of 0. However, two factors limit the accuracy of the intersections and thus require a bit of an error envelope. First, depending on the angle of the camera relative to the hand, cameras may see the tip of a finger in a slightly different position than the others. At some angles, the edge of the finger appears to be on the side of the finger; at others, it appears to be directly on the finger tip. Second, camera calibration involves a series of matrix calculations that can be prone to propagating minor errors (which unfortunately result from ARToolkit's tracking.) These errors can affect the precision of the camera measurements and result in slightly inaccurate ray calculations. A threshold of 4 mm prevents an unacceptable number of false negatives.

This threshold is minimized, though, to reduce the number of false positives. As the number of rays crossing a limited volume increases, so does the likelihood of random intersections. The system does not limit the number of points a single ray can identify to account for the possibility of one finger occluding the other. However, this allowance means that if a camera identifies a candidate point at or near the position of another camera, it can create false positive 3D points. When made visible, these points can obscure the view of the scene, but they do not have much of a significant effect on interaction. False positive points far from the actual fingertips are typically out of the way and fleeting enough to not noticeably affect interaction with objects.

False positive points can and do occur, however, very close to the real fingertips. These points are dealt with through a tracking system that coordinates how to remember which fingertip is which frame to frame.

Tracking and Filtering

Tracking and filtering are topics as old as computer vision itself. Naturally, a number of algorithms and techniques have been explored, each with strengths and weaknesses. Tracking is a difficult problem comprised of sub-problems including identification, prediction, non-linear behavior, and interruption (through occlusion or other reasons). The solutions to these problems often have to balance complexity and resource requirements with robustness and accuracy.

Most AR+gesture systems track only the hand without attention to fingers. These systems use a variety of established tracking algorithms such as optical flow in [60] and [43], MeanShift and continuously adaptive MeanShift in [57] and [69], the Viola-Jones tracking algorithm in [47] and [29],the KLT tracking algorithm in [62] and [51], and the more recent SIFT/SURF techniques [43] or condensation algorithms [70]. These algorithms, though powerful, are unable to track an arbitrary, changing number of objects – like the number of fingertips visible to a camera.

Fingertip tracking as a topic does not attract the same interest as the broader problem of hand tracking, though many of the same issues apply. Since this system is intended for natural object manipulation, the only features necessary to track were individual fingertips. Hand orientation information was not necessary. Two methods are popular in literature for tracking individual points: Kalman filtering and particle filtering.

A Kalman filter creates a (typically linear) model of a point and its movements [71]. The filter creates a prediction of the point's movement based on the model and is iteratively updated based on the measurement of the point's actual movement. Kalman filters are appropriate when the error in the measurements are Gaussian, but otherwise tend to make erroneous predictions. This method has been proven to be useful for tracking a marked finger in stereoscopic environment [72].

Kalman filters are considered "single hypothesis" filters, meaning the filter has only one guess about where the tracked point could be. Multiple hypothesis filters exist, most notably particle filters. Particle filters create recursive, Bayesian estimates of particles based on measurements and are suitable for tracking points that may have multiple likely positions at a given time. Particle filters have been used for hand tracking by [26], [27], [73], [74], and [69].

At first glance, it would seem that particle filters are more appropriate for this system because it has to track multiple fingertips through motion that is not linear and unlikely to have Gaussian measurement errors. However, particle filters require significantly more computing power to run. To ensure real-time or near real-time processing speeds and to reduce complexity, this system employed a form of Kalman filtering.

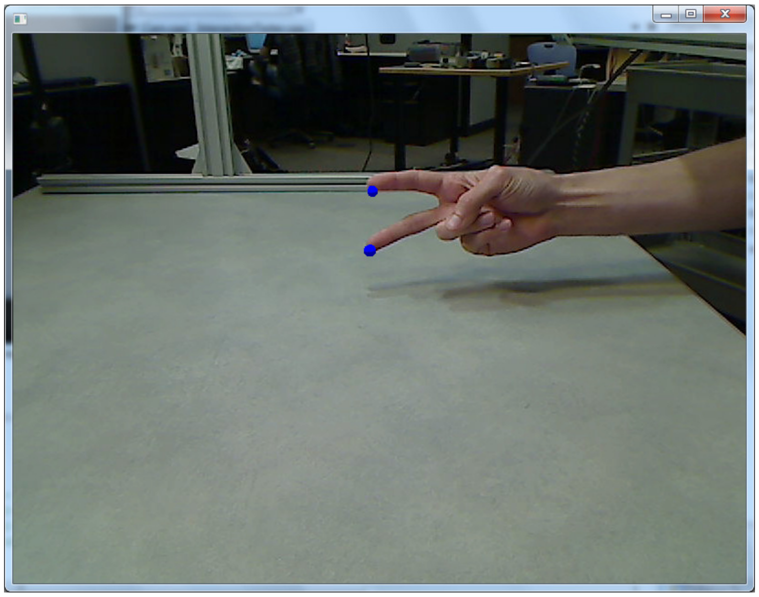
Figure 21: Stable, Tracked Points 1

Tracking is currently paired with intersection calculation in an independent thread. As previously described, the tracking system receives 3D points representing fingertips each update. These points may include clusters of false positives around the fingertips. To eliminate as many false positives as possible, all points within three centimeters are merged together into one average point. Three centimeters is acceptable because the system does not currently support any interactions of fingers pressed together and three centimeters is a distance of slightly spaced apart fingers on an average hand.

The tracking algorithm maintains a vector of tracked points and velocities. Each iteration, every point is updated with its linear velocity vector. The updated points are then paired with candidate points based on shortest geometric distance. That is, the system determines the closest pair of tracked and current points. The tracked point and velocity are updated based on the new point using a moving average calculation. Thanks to a relatively small number of points and the thread's processing speed, a 20 frame moving average is calculated without noticeable lag. The updated points are then removed from the lists. This process continues until all tracked points or all detected points have been updated.

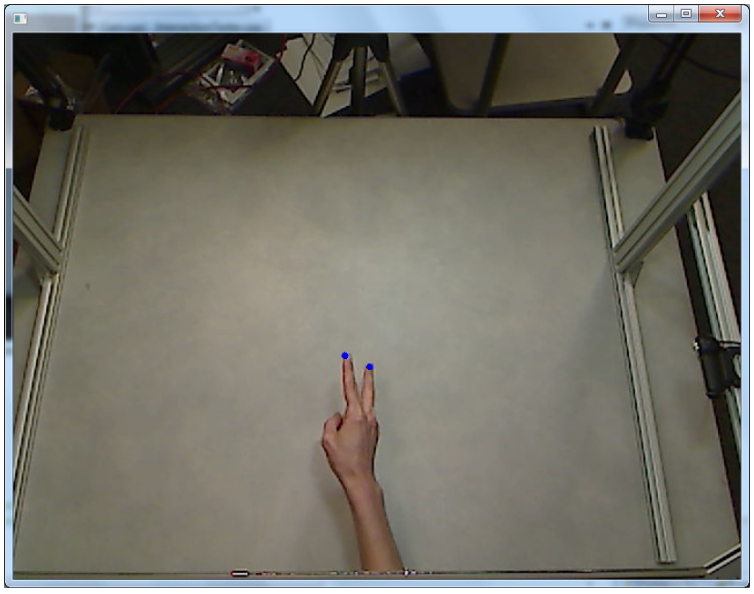

Figure 22: Stable Tracked Points 2

Tracked points that do not find a candidate point for updating are left as-is and allowed to persist for up to 15 frames without an update. If no update is found at that point, the tracked point is removed from the system. Candidate points that are left over without a corresponding tracked point are added to the vector of tracked points for future iterations.

This system provides acceptable tracking of fingertip points. In parallel, a list of indices to tracked points is maintained such that the main application thread can track individual tracked points, enabling interactions like translation and rotation using the fingertips.

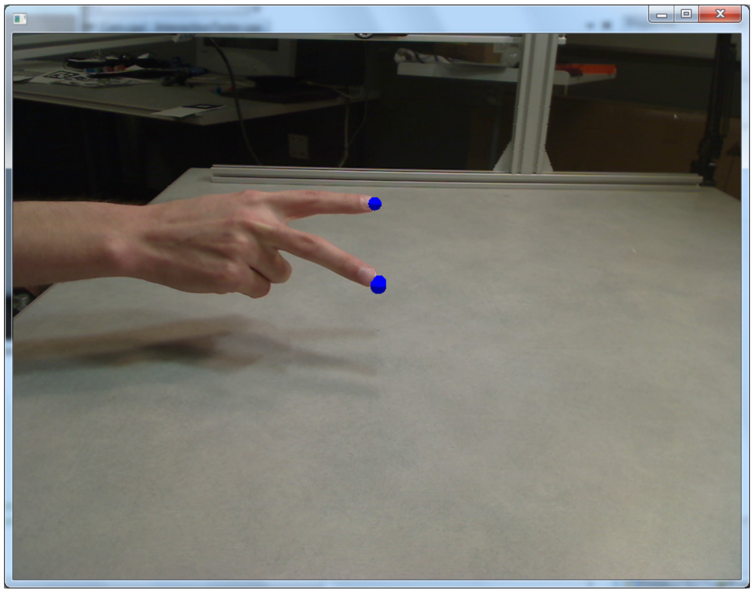

Figure 23: Overlapping 3D False Positives

The tracking system is not perfect, however. False positives still remain and can persist for 15 frames at a time. Rapid movement can create false positives which then persist. Fortunately, the tracked velocity of these points tends to be high, removing them from the area of interaction quickly. Perhaps surprisingly, the Gaussian noise sensitivity of Kalman filters does not seem to be a problem. OpenCV has an implementation of Kalman filters and was tested and found to be very jittery and noisy. In contrast, the system's current tracking system does not have significant difficulty with changes in momentum or direction.

This Kalman-like tracking system is fundamentally similar to a method described by Argyros and Lourakis [75]. Argyros & Lourakis developed a system using adaptive skin histograms and blob tracking which used iteratively updated hand position hypotheses to follow the hands. Their hypotheses in turn were robust against changes in momentum and even occlusion. Further improvements to this system's tracking could be to more fully implement Argyros & Lourakis' statistical tracking methods. Another avenue of improvement would be to test the precision and computing requirements of particle filters.

Occlusions and the Illusion of Depth

As discussed under Previous Work, the occlusion of objects in space is an important sensory cue for determining the depth of a scene. Given the AugmenTable's current lack of haptic feedback or shadow cues, it was important to develop a means for the user to identify his or her hand position relative to the virtual objects in the workspace. The method developed works as follows.

In the contour detection step of identifying hands, the system creates a list of all contour bounding rectangles for each camera. When a camera is selected as the active view, this list of rectangles is imported into the main scene creation thread. Each frame, the 3D detected points within the scene are transformed back into the 2D plane and tested to see if they fall within any of the camera's contour rectangles. If they do, their depth value is averaged and applied to the contour rectangle image.

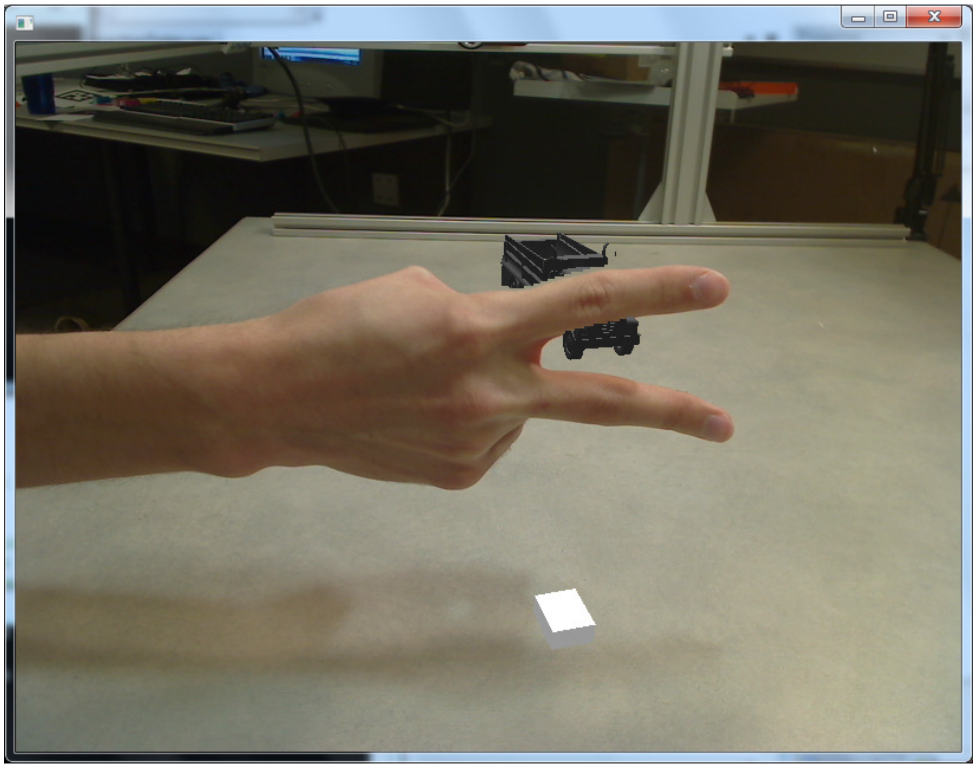

Figure 24: Occlusion of Virtual Objects

With the average depth value of the rectangle, the rectangle’s corner points undergo the reverse transformation to render the 2D points in 3D. An OpenSceneGraph rectangle object is then created in the scene. Finally, the skin color mask created in the camera’s image processing is used to create a transparent texture that shows the user’s hand within the rectangle but allows the scene behind the rectangle to be visible around and through the user’s fingers (Figure 24.) The AugmenTable’s PC is fast enough to allow this to run at close to real time speeds.

Close, though, is not fast enough. Additionally, there is some artifacting that results from applying the 2D texture of a hand to a 3D dimensional rectangle. As a result, the occlusion is functional but not optimal. Future refinement may yield a better occlusion rendering system.

Since speed is critical, an alternative is to not render the hands as occluding planes, but to instead display small spheres where fingerpoints are detected. These spheres are then culled and positioned accurately within the scene and can be used to infer the depth of the fingertips. This is not as intuitive as visual hand occlusion, but the simplicity of this method results in a much faster update rate for the scene geometry. Further testing is necessary to determine which method may be more preferable to users.

Chapter Four: Application & Discussion

Tracking unmarked fingertips in an augmented reality environment offers a number of opportunities as well as challenges in developing interactions and applications. A number of interactions have been explored in the cases of marked or similarly tracked hands and when using an individual unmarked finger, as previously described. Many of these interactions, such as 3D drawing using a fingertip, are interesting demonstrations of technology, but this system has been designed with a more industrial usage in mind. I've attempted to design the application described here to reflect this.

Object Manipulation Prototype

This prototype provides a means of manipulating a virtual object through selection, translation, rotation, and scaling widgets. As shown at left, a virtual object exists within the augmented reality workspace. This object can be any lightweight model supported by OpenSceneGraph. A user may reach into the space and select the object by "touching" it with his or her finger. Since the object is not real, there is no haptic response. Instead, when an intersection of the object and fingertip is detected, a manipulation widget appears (Figure 26.) This manipulation widget expires after a set period of time if the user does not interact with it.

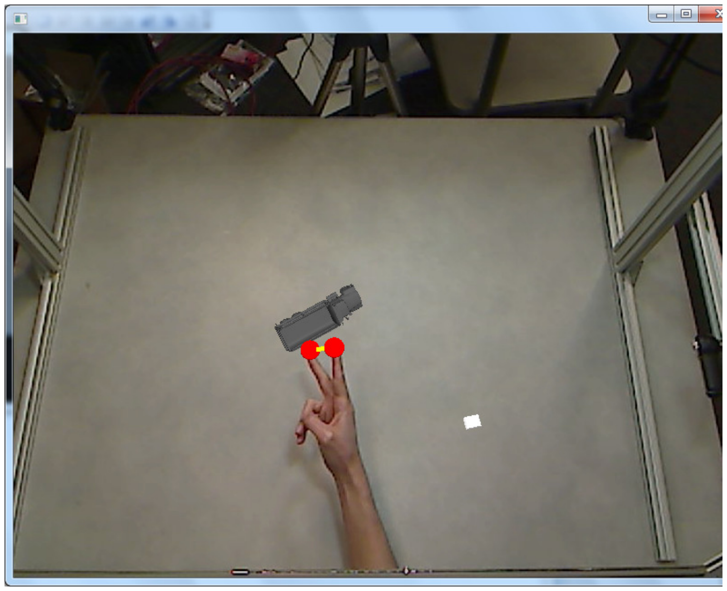

Figure 25: Object Translation and Rotation

After selection, the user modifies his or her hand posture to extend any two fingers into the spherical ends of the manipulator widget. When a fingertip is within both spheres, the spheres turn red and "lock" onto the user's fingertips. The widget supports translation and rotation. The user can move his or her hand anywhere within the volume and the object will track with it. This interaction can handle fairly rapid hand motions. If the user rotates his or her fingers, the locked object is also rotated in kind. With only two points, the user cannot rotate the object about the axis of the two points. However, this can be worked around by an intermediate rotation to change the axis of the fingertips. To stop the translation/rotation mode, the user can break the lock by moving his or her fingers in or out along the two sphere axis.

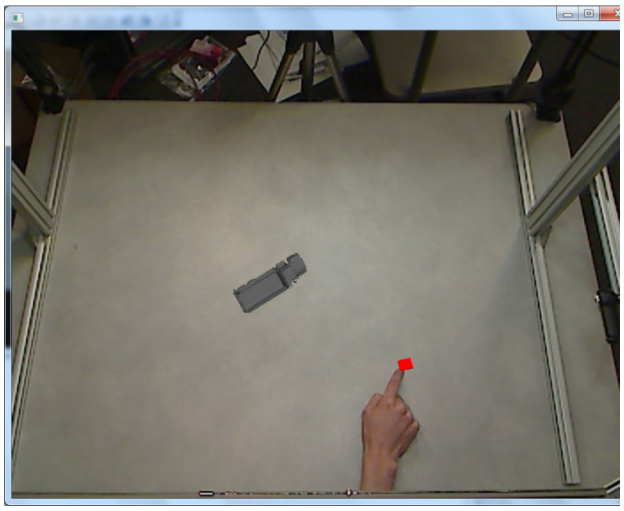

Figure 26: Virtual Button Press

The application features an additional cube in the scene that functions as a button. When the user “presses” the button, it changes color to indicate the engagement of scaling mode. Now, when a user intersects a scene object, the manipulator widget appears to provide a scaling interaction. In this case, the user locks his or her fingers into the manipulator spheres as before. The user can then move his or her fingers along the axis of the two spheres to enlarge or shrink the object model. This interaction mode is broken by making a translation or rotation style gesture, removing the fingers from the spheres.

As noted in the previous work section above, the use of 3D control widgets has not been well explored in literature. 3D widgets were described fairly early by Mine [37], but the only authors that used them in any way were Dachselt and Hinz [76]. As far as I am aware, the uses of 3D widgets in this application to translate, rotate, and scale an object through direct superposition and control is unique.

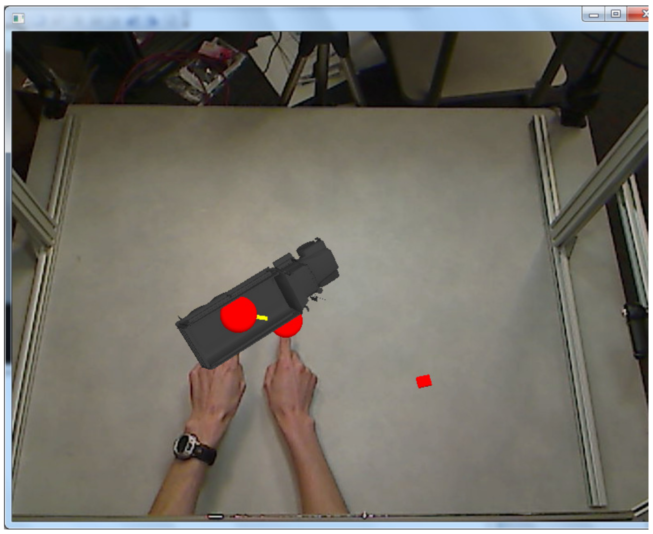

Figure 27: Object Scaling

The advantage of this application is direct object manipulation. Though there is an intermediate widget between the user and the object, the user rotates both the widget and object by rotating his or her hand, moves with his or her hand, and scales by spacing his or her hands apart. This interaction is one step closer to holding an object and playing with it than previously seen. A user study to evaluate the learnability of this theoretically more "natural" gesture compared to learning the manipulation techniques required in most CAD design packages would be illuminating.

This application also shows the limitations of the interaction design as well. Rotation with a hand is very simple only within a certain range of motion. This is due to the kinematic limitations of the wrist joint and the rotation of the forearm. In normal tool use, humans work around these limitations by positioning their arm through the shoulder and even body position. The apparatus for this system, however, expects a specific orientation of the user (sitting or standing facing the screen) which limits the freedom of movement to the elbow and above.

Other restrictions can be observed from the limitations of the system. For one, the space a user can move objects around in is limited to the volume that is intersected by more than one camera's field of view. Unfortunately, there currently are no cues to inform the user where the boundaries of this volume are. Another is that if the hand occludes itself in certain angles of rotation, fingertip tracking can be lost. These problems are common to nearly all hand tracking systems, and one credible solution is to increase the number of cameras at various angles of view. As this incurs a resource cost on the operating PC, a more common solution is to limit or otherwise design the interaction gestures around the constraint.

One possible solution for dealing with this limitation is to provide a widget control for the entire scene. This would enable the user to adjust the scale of the scene to enable him or her to utilize the entire working volume as well as shrink too-large scenes. It would also provide a means of changing the user's perspective of the scene. This solution has not yet been implemented.

Despite the limitations, this prototype application implies a number of interesting interactions. Object manipulation can also be applied to virtual controls such as sliders, knobs, or levers. The AugmenTable is designed to support exploring and researching interaction through virtual widgets and without.

Discussion

Computer vision is a vast and complicated field. It's a field that is characterized by strong imagination to envision the ways computers ought to see, myriad arcane techniques to accomplish it, and results that are seldom as compelling as the original vision. In that respect, this project was no different. This section will discuss some of the realized benefits, some shortcomings, and the future work that could dramatically extend the system's capabilities.

Realized Benefits

Originally, the vision for this project was for a person to reach out a hand and move and inspect a virtual object in a work environment. The large majority of this vision has been realized. A user can walk up to the apparatus, reach under and behind the display and manipulate virtual objects and real objects side by side with only his or her bare hands. As discussed above, this is a rare accomplishment yet today. Most interactions require some mediating technology like colored gloves, accelerometers, fiducial markers, or other even more conspicuous equipment. What's more, the system can recognize an arbitrary number of hands or fingers so long as occlusions are addressed with improved camera coverage. This is also uncommon with the comparable systems.

The AugmenTable also proves successful in realizing a believable mixed reality environment. Through the use of visual display, hand obfuscation, occlusion, and some quasi-haptic feedback (as provided by the tabletop surface,) the system provides a suspension of disbelief about the nature of the virtual objects within the workspace scene. This suspension is not complete. A user still has to use a constructed interaction technique to manipulate virtual objects, but it can be effective.

One benefit is the low price tag. All of the hardware used is commonly accessible and inexpensive. Custom or expensive components (and the algorithms that rely on them) were purposefully avoided. The largest expense in the project is the multi-core workstation PC. Similarly, all of the software libraries used are open source and freely available. Total hardware costs are less than $4,000 today and the software only had personal time as an expense.

Research Contributions

This system exemplifies a few novel ideas within the augmented reality research field. One is a novel apparatus that expands the common mobile, hand-held window metaphor into a large-scale desktop system. This scaling of the AR window begins to take on characteristics of the more immersive HMD setup by expanding the view to encompass more of the user's field of vision and allowing the placement of the user's body (in this case hands and arms) within the AR environment. Like an HMD, this setup acts as an intermediary between reality and the user's vision, enabling more rich mixed reality experiences, but without the added steps of donning an uncomfortable head-mounted piece of hardware. This has proven to be an advantage in demonstrating the system’s capabilities. The apparatus was set up at a conference alongside a typical HMD system and received noticeably more attention and use.

This AugmenTable also features minor innovations in creating interaction styles based on the tracking of an arbitrary number of hands/fingertips in a specified volume. Though bare-handed finger interactions have been developed and described above, only one other system manipulates the virtual objects through use of intuitive 3D widgets. The barbell-shaped widget used in the prototype application allows for comfortably controlling many differently sized or shaped objects with whichever hand(s) and fingers the user prefers. The scope of this project, however, was insufficient to provide a study of the interaction techniques and compare their intuitiveness or learnability to other methods.

Challenges

The process of developing the system described here featured several surprises. First and foremost was the plethora of computer vision techniques to accomplish a given task. Tracking of hands has been done in numerous ways (as shown in the Method section) with algorithms of varying levels of complexity. In most cases, I selected methods for their computational efficiency when possible. An important goal was to have real time interaction that could follow human movements at speed. This required light weight algorithms that would not slow processing down too much. Even with this intent the application requires relatively powerful off-the-shelf hardware to run well in real time. In the end, almost 20 variations of the system's code were developed to test assorted algorithms and techniques.

By far, the most challenging aspect of developing this system was the selection and fine-tuning of a tracking algorithm. Originally, Kalman filtering was selected due to its relative simplicity and for the built-in functionality for such tracking in OpenCV. Kalman filtering was applied to the three-dimensional fingertip points, but several rounds of tweaking were unable to result in tracked points stable enough to use for interaction. A second attempt was made to simplify the information being tracked by creating a hand data structure that stored all points relative to each identified hand's center of mass. This encoding did not enable sufficient tracking accuracy as well. A third revision was attempted to move the tracking of points upstream into the 2D processing. This had better results for accuracy but dramatically reduced the operating speed of the processing threads.

As noted before, part of the difficulty lay with the managing OpenCV's implementation of Kalman filters. This implementation expects measurement updates to occur at set intervals. However, as the number of points being processed varies from zero to many, the amount of time the intersection and merging process takes varies. This variability resulted in fluctuating velocity of tracked points that rendered the output too jittery for interaction.

This problem was addressed by parsing out the steps of a Kalman filter into a more step-by-step moving average process as described in Tracking and Filtering. In this way, the Kalman filter is integrated directly into the calculation thread and is more adaptable to fluctuations in update rates due to processing burdens. The tracking system was also made much less susceptible to erroneous measurements by dramatically increasing the weighting of the average measurement compared to the most recent update by a ratio of 20:1. Unfortunately, the tracking system is not perfect and can lose the fingertips in motion in favor of false positives. This problem merits further work and examination.

Overall, the project was successful at realizing its goals, but time constraints prevented the level of testing and exploration of alternative algorithms and techniques that would have ensured the best possible product. The lessons learned throughout the development of this system, though, have yielded a number of strong contending ideas to improve the system significantly.

System Limitations

Many limitations implicit in the system have already been noted: a restricted actionable volume, limited movement due to kinematic restrictions of the human hand and arm, and others. The system's occlusion is only an approximation of hand position and does not calculate individual finger occlusions. Additionally, the angle of view provided to the user can result in difficulty evaluating the position of the interface widgets when user rotation causes it to be occluded by other virtual objects or itself. Finally, the tracking method is not perfect and still results in false positives, some jitter, or missed fingertips.

In Comparable Systems, I discussed the limitations of several AR environments that had similar goals to the AugmenTable's. One limitation was the adaptability of pose recognition systems: pose recognition systems require statistical training and do not allow arbitrary gestures. Unfortunately, the current incarnation of the AugmenTable suffers the same limitation through use of predefined 3D widgets. This limitation and others, however, can be addressed through suggestions presented under Future Work.

Future Work

This section describes some of the improvements and future work that could extend the capabilities of the AugmenTable. These improvements are divided into three general categories of Applications, Apparatus, and Concept.

Application

First and foremost future work would be the refinement and improvement in the tracking accuracy. More advanced detection and tracking methods such as adaptive histograms and particle filters require greater computational resources, but offer statistical inference as a tool to more accurately predict and update measurements of fingertips in stasis as well as in motion. These methods merit further research to enable an interaction that reacts consistently to user input across gestures, angles, and lighting conditions.

The intuitive aspects of the prototype applications can be improved upon by the addition and extension of realistic (or at least intuitive) physics. Elements of gravity would add to the immersiveness of the applications and enable both fun and practical interactions. Similarly, giving objects a level of mass or inertia (like that seen in the multitouch swipe gestures of interfaces such as the iPad) can increase the power of gestures without reducing the benefits of direct manipulation. Physics and mass would enable another set of interactions such as pinch, bump, and momentum transfers. More broadly, physics could possibly be extrapolated to creating shadows of objects that would provide an additional depth cue and increase the melding of virtual and real within the workspace. Finally, physics provides an overall expectation of interaction. Users are accustomed to the physical world where objects behave in a reliable manner due to the laws of physics. With a software physics engine, a similar expectation is created in a virtual environment. This expectation allows users to more easily extrapolate real actions to virtual actions. Physics is therefore a significant step to opening an augmented reality to arbitrary object manipulation without intermediary widgets.

That said, it is important that the value of the current and future interactions is proven against that of existing methods. To this end, a user based study of interactions of different augmented reality systems should be conducted. This study could evaluate the practicality, usability, learnability, and overall satisfaction of manipulating virtual and/or real objects with a variety of tools and techniques.

The capabilities of the application could also be extended through using a gesture classification system. The tracking of the finger points currently performed by the system could be used as gestural inputs to a pose recognition system similar to those described under Previous Work. This would provide both object manipulation and command/control functionality to the user.

Apparatus

In the introduction, I mentioned several technological trends which are rapidly being shaped by the cutting edge of technology. This system apparatus also could benefit from a number of additional advanced technologies. For instance, 3D displays are coming onto the market in 2010. The thin display here could be replaced with a 3D capable display and provide stereo perception to improve the immersiveness. Another possible display change would be swapping the simple display with a multitouch display. This could enable a mixture of 2D and 3D control of virtual objects and interfaces.

One possibility would be to add a forward facing camera that provides face tracking of the user. Face tracking can enable changing the perspective of the display in order to provide correct occlusion of objects relative to the user's perspective. This creates a much greater three dimensionality effect than stereo display alone, and is much cheaper than the nascent 3D monitor technology.

Concept

All of the above ideas would add to the immersiveness or practical capabilities of the system. A more compelling line of inquiry, though, is rethinking the entire method of tracking fingertips in favor of creating a 3D representation of the entire hand. Such a representation would render fingertip tracking unnecessary and enable a host of natural interactions like grabbing, pinching, flicking, and more. A model of the hand in space would also simplify and dramatically improve the rendering of hand/object occlusion in the workspace. A 3D representation would thus solve the two thorniest problems of this system.

A number of methods exist that could enable 3D structure of hands. The simplest (and therefore least accurate) would be to extrapolate the contour detection method described above to "carve" the entire contour out of space. As each camera carves out the negative spaces between contours, the remainder is a blocky approximation of the hand. It would be sensitive to hand occlusions, but with enough camera coverage it could provide better sensing of the hand than the current method. This method was used with success by Schlattman and Klein [39].

A number of other more powerful (and complicated) methods exist in literature. Structured light has been used to identify 3D surfaces at a high resolution (for example, in [77]), though at high computational cost. Inverse kinematics, the process of matching an approximation of the human hand skeleton to a tracked hand image, is another well-researched method for creating and positioning a hand model in space. Erol et al review a number of papers that utilize inverse kinematics in a pose estimation context [20]. Finally, the present method could be replicated in part with depth sensing webcams such as the 3DV ZCam (or possibly the upcoming Xbox Natal camera). If the time-of-flight infrared cameras can coexist within a viewing volume, the depth maps could be used to recreate the hand with decent resolution. The resulting meshes could be combined into an articulated model using methods described in [78].

Each of the alternatives described here may require significant computing power to enable interactions in close to real time, but the simplification of hand positioning and interaction in the scene would create a truly new augmented reality experience. Moreover, this list is not exhaustive; new methods of inferring three dimensions from two are frequently developed. I believe that an augmented reality environment such as the AugmenTable will become much more valuable to end users when the hand is recognized and recreated in the virtual space and can fully interact with the scene.

Chapter Six: Conclusion

This project successfully realized recognition of unmarked, unencumbered hands towards integration with virtual objects in a novel augmented reality workspace. It combined a number of well established computer vision algorithms with a new interaction metaphor of superimposed, hand-sized widgets and unique apparatus. These interactions enable manipulating virtual objects and controls and can provide an advanced experience for conceptual design or play. Despite this success, the AugmenTable has not been perfected. It has many avenues for advancement including 3D immersion, multitouch, and structured hand models. These improvements can increase the accuracy, immersiveness, and potential interactions of the system.

This research was funded through a grant from Rockwell-Collins Inc.

Bibliography

[1] B. Shneiderman, "Direct manipulation," *Proceedings of the joint conference on Easier and more productive use of computer systems. (Part - II) Human interface and the user interface*, 1981, p. 143.

[2] S. Reifinger, F. Wallhoff, M. Ablassmeier, T. Poitschke, and G. Rigoll, "Static and dynamic hand-gesture recognition for augmented reality applications," *Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments*, 2007, p. 728–737.

[3] A. Eliott, "10 Amazing Augmented Reality iPhone Apps," *http://mashable.com/2009/12/05/augmented-reality-iphone/*, 2010.

[4] O. Inbar, R. Nir, and T.K. Carpenter, "Games Alfresco," *http://gamesalfresco.com/*, 2010.

[5] Google, "Google Goggles," *http://www.google.com/mobile/goggles/#landmark*, 2010.

[6] P. Mistry and P. Maes, "Sixth sense: integrating information with the real world," *http://www.pranavmistry.com/projects/sixthsense/*, 2010.

[7] C. Harrison, D. Tan, and D. Morris, "Skinput: Appropriating the Body as an Input Surface," *Proceedings of the 28th Annual SIGCHI Conference on Human Factors in Computing Systems*, Atlanta, Georgia: 2010.

[8] D. Dumas and Wired.com, "CES 2010: Hands-On With Transparent Display of the Future," *http://www.wired.com/video/ces-2010-hands-on-with-transparent-display-of-the-future/60826805001*, 2010.

[9] S. Kim and A.K. Dey, "AR interfacing with prototype 3D applications based on user-centered interactivity," *Computer-Aided Design*, vol. 42, 2010, pp. 373-386.

[10] G. Bleser, Y. Pastarmov, and D. Stricker, "Real-time 3d camera tracking for industrial augmented reality applications," *Journal of WSCG*, 2005, p. 47–54.

[11] C. von Hardenberg and F. Bérard, "Bare-hand human-computer interaction," *Proceedings of the 2001 workshop on Perceptive user interfaces*, New York, New York, USA: ACM New York, NY, USA, 2001, p. 1–8.

[12] P. Lamb, "ARToolkit," *http://www.hitl.washington.edu/artoolkit/*, 2007.

[13] V. Buchmann, "FingARtips – Gesture Based Direct Manipulation in Augmented Reality," *Virtual Reality*, vol. 1, 2004, pp. 212-221.

[14] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana, "Virtual object manipulation on a table-top AR environment," *IEEE and ACM International Symposium on Augmented Reality, 2000.(ISAR 2000). Proceedings*, 2000, p. 111–119.

[15] D. Sturman and D. Zeltzer, "A survey of glove-based input," *IEEE Computer Graphics and Applications*, 1994.

[16] C. Keskin, A. Erkan, and L. Akarun, "Real time hand tracking and 3D gesture recognition for interactive interfaces using HMM," *ICANN/ICONIPP*, 2003, p. 26–29.

[17] S. Walairacht, K. Yamada, S. Hasegawa, Y. Koike, and M. Sato, "4+ 4 fingers manipulating virtual objects in mixed-reality environment," *Presence: Teleoperators \& Virtual Environments*, vol. 11, 2002, p. 134–143.

[18] J. Rehg and T. Kanade, "DigitEyes: vision-based hand tracking for human-computer interaction," *Proceedings of 1994 IEEE Workshop on Motion of Non-rigid and Articulated Objects*, 1994, pp. 16-22.

[19] C. Nölker and H. Ritter, "Detection of fingertips in human hand movement sequences," *Gesture and Sign Language in Human-Computer Interaction*, Springer, 1998, p. 209–218.

[20] A. Erol, G. Bebis, M. Nicolescu, R.D. Boyle, and X. Twombly, "Vision-based hand pose estimation: A review," *Computer Vision and Image Understanding*, vol. 108, 2007, pp. 52-73.

[21] K. Abe, H. Saito, and S. Ozawa, "Virtual 3-D interface system via hand motion recognition from two cameras," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 32, 2002, pp. 536-540.

[22] W. Chen, R. Fujiki, D. Arita, and R. Taniguchi, "Real-time 3D Hand Shape Estimation based on Image Feature Analysis and Inverse Kinematics," *14th International Conference on Image Analysis and Processing (ICIAP 2007)*, 2007, pp. 247-252.

[23] K. Oka, Y. Sato, and H. Koike, "Real-time tracking of multiple fingertips and gesture recognition for augmented desk interface systems," *Proceedings of the fifth IEEE international conference on automatic face and gesture recognition*, IEEE Computer Society Washington, DC, USA, 2002, p. 429.

[24] Y. Sato, Y. Kobayashi, and H. Koike, "Fast tracking of hands and fingertips in infrared images for augmented desk interface," *International conference on automatic face and gesture recognition*, Grenoble, France: 2000, pp. 462-467.

[25] L. Bonansea, "3D Hand gesture recognition using a ZCam and an SVM-SMO classifier," *Journal of empirical research on human research ethics : JERHRE*, vol. 5, 2010.

[26] L. Bretzner, I. Laptev, and T. Lindeberg, "Hand gesture recognition using multi-scale colour features, hierarchical models and particle filtering," *Proceedings of Fifth IEEE International Conference on Automatic Face Gesture Recognition*, 2002, pp. 423-428.

[27] T. Gumpp, P. Azad, K. Welke, E. Oztop, R. Dillmann, and G. Cheng, "Unconstrained Real-time Markerless Hand Tracking for Humanoid Interaction," *2006 6th IEEE-RAS International Conference on Humanoid Robots*, 2006, pp. 88-93.

[28] S. Kang, M. Nam, and P. Rhee, "Color Based Hand and Finger Detection Technology for User Interaction," *Convergence and Hybrid Information Technology, 2008. ICHIT'08. International Conference on*, 2008, p. 229–236.

[29] M. Kolsch and M. Turk, "Robust hand detection," *Proc. of the Sixth IEEE Int. Conf. on Automatic Face*, vol. 17, 2004, pp. 614-619.

[30] C. Malerczyk and G. Darmstadt, "Dynamic Gestural Interaction with Immersive Environments," *Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 2008.

[31] N. Petersen and D. Strieker, "Fast Hand Detection Using Posture Invariant Constraints," *KI 2009: Advances in Artificial Intelligence: 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009, Proceedings*, Springer, 2009, p. 106.

[32] J. Corso and G. Hager, "Gesture Recognition Using 3D Appearance and Motion Features," *2004 Conference on Computer Vision and Pattern Recognition Workshop*, 2004, pp. 160-160.

[33] M. Fukumoto, Y. Suenaga, and K. Mase, "“Finger-Pointer”: Pointing interface by image processing," *Computers & Graphics*, vol. 18, 1994, pp. 633-642.

[34] D.A. Bowman and L.F. Hodges, "An Evaluation of Techniques for Grabbing and Manipulating Objects in Immersive Virtual Environments," *Proceedings of the 1997 symposium on Interactive 3D graphics*, 1997, pp. 35-38.

[35] M. Mine, F. Brooks Jr, and C. Sequin, "Moving objects in space: exploiting proprioception in virtual-environment interaction," *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 1997, p. 19–26.

[36] I. Poupyrev, T. Ichikawa, S. Weghorst, and M. Billinghurst, "Egocentric Object Manipulation in Virtual Environments: Empirical Evaluation of Interaction Techniques," *Computer Graphics Forum*, vol. 17, 1998, pp. 41-52.

[37] M. Mine, "Virtual environment interaction techniques," *UNC Chapel Hill Computer Science Technical Report TR95-018*, 1995, p. 507248–2.

[38] C. Hand, "A survey of 3D interaction techniques," *Computer graphics forum*, vol. 16, 1997, pp. 269-281.

[39] M. Schlattman and R. Klein, "Simultaneous 4 gestures 6 DOF real-time two-hand tracking without any markers," *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, ACM, 2007, p. 42.

[40] J. Segen and S. Kumar, "Gesture vr: vision-based 3d hand interace for spatial interaction," *Proceedings of the sixth ACM international conference on Multimedia*, ACM New York, NY, USA, 1998, p. 455–464.

[41] K. Oka, Y. Sato, and H. Koike, "Real-time fingertip tracking and gesture recognition," *IEEE Computer Graphics and Applications*, vol. 22, 2002, pp. 64-71.

[42] P. Song, S. Winkler, S. Gilani, and Z. Zhou, "Vision-based projected tabletop interface for finger interactions," *Lecture Notes in Computer Science*, vol. 4796, 2007, p. 49.

[43] T. Lee and T. Hollerer, "Hybrid Feature Tracking and User Interaction for Markerless Augmented Reality," *2008 IEEE Virtual Reality Conference*, 2008, pp. 145-152.

[44] T. Lee and T. Höllerer, "Handy AR: Markerless inspection of augmented reality objects using fingertip tracking," *International Symposium on Wearable Computers*, Citeseer, 2007, pp. 83-90.

[45] A.I. Comport, E. Marchand, M. Pressigout, and F. Chaumette, "Real-time markerless tracking for augmented reality: the virtual visual servoing framework.," *IEEE transactions on visualization and computer graphics*, vol. 12, 2006, pp. 615-28.

[46] P. Song, H. Yu, and S. Winkler, "Vision-based 3D finger interactions for mixed reality games with physics simulation," *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, ACM, 2008, p. 7.

[47] S. Kolarić, A. Raposo, and M. Gattass, "Direct 3D Manipulation Using Vision-Based Recognition of Uninstrumented Hands," *Symposium of Virtual and Augmented Reality*, 2008, pp. 212-220.

[48] A. Tschesnok, "Organic Motion," *http://organicmotion.com/*, 2010.

[49] M.S. Graziano, "Where is my arm? The relative role of vision and proprioception in the neuronal representation of limb position," *Proceedings of the National Academy of Sciences of teh United States of America*, vol. 96, 1999, pp. 10418-10421.

[50] C. Furmanski, R. Azuma, M. Daily, and H.R. Laboratories, "Augmented-reality visualizations guided by cognition : Perceptual heuristics for combining visible and obscured information," *Symposium A Quarterly Journal In Modern Foreign Literatures*, 2002.

[51] Y. Pang, M.L. Yuan, A.Y. Nee, S.K. Ong, and K. Youcef-toumi, "A Markerless Registration Method for Augmented Reality based on Affine Properties," *Proceedings of the 7th Australian User Interface Conference*, Hobart, Australia: 2006, pp. 24-32.

[52] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers, "Wallflower : Principles and Practice of Background Maintenance," *Seventh International Conference on Computer Vision, Vol. 1*, Corfu, Greece: 1999, p. 255.

[53] K. Kim, T. Chalidabhongse, D. Harwood, and L. Davis, "Real-time foreground–background segmentation using codebook model," *Real-Time Imaging*, vol. 11, 2005, pp. 172-185.

[54] J. Letessier and F. Bérard, "Visual tracking of bare fingers for interactive surfaces," *Symposium on User Interface Software and Technology*, 2004.

[55] J. Yang, W. Lu, and A. Waibel, "Skin-color modeling and adaptation," *Lecture Notes in Computer Science*, 1997, p. 687–694.

[56] P. Kakumanu, S. Makrogiannis, and N. Bourbakis, "A survey of skin-color modeling and detection methods," *Pattern Recognition*, vol. 40, 2007, pp. 1106-1122.

[57] T. Kurata, T. Okuma, M. Kourogi, and K. Sakaue, "The Hand Mouse: GMM hand-color classification and mean shift tracking," *Proceedings IEEE ICCV Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems*, 2009, pp. 119-124.

[58] M. Jones and J. Rehg, "Statistical color models with application to skin detection," *International Journal of Computer Vision*, vol. 46, 2002, p. 81–96.

[59] O. Ikeda, "Segmentation of faces in video footage using HSV color for face detection and image retrieval," *International Conference on Image Processing*, 2003, p. 913–6.

[60] F. Dadgostar and a. Sarrafzadeh, "An adaptive real-time skin detector based on Hue thresholding: A comparison on two motion tracking methods," *Pattern Recognition Letters*, vol. 27, 2006, pp. 1342-1352.

[61] J. Terrillon and S. Akamatsu, "Comparative performance of different chrominance spaces for color segmentation and detection of human faces in complex scene images," *Proc. of the 12th Conf. on Vision Interface*, 1999, pp. 19-21.

[62] M. Kolsch and M. Turk, "Fast 2d hand tracking with flocks of features and multi-cue integration," *CVPRW'04: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04*, Citeseer, 2004, p. 158.

[63] A. Burns and B. Mazzarino, "Finger tracking methods using eyesweb," *Lecture Notes in Computer Science*, vol. 3881, 2006, p. 156.

[64] J. Crowley, F. Berard, and J. Coutaz, "Finger tracking as an input device for augmented reality," *International Workshop on Gesture and Face Recognition, Zurich*, Citeseer, 1995, pp. 1-8.

[65] S. Suzuki and K. Abe, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, 1985, pp. 32-46.

[66] D. Douglas and T. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Communications of the Association for Computing Machinery*, vol. 15, 1972, pp. 11-15.

[67] Y. Piao and J. Sato, "Computing Epipolar Geometry from Unsynchronized Cameras," *14th International Conference on Image Analysis and Processing (ICIAP 2007)*, 2007, pp. 475-480.

[68] P. Bourke, "The Shortest Line Between Two Lines in 3D," *http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline3d/*, 1998.

[69] C. Shan, T. Tan, and Y. Wei, "Real-time hand tracking using a mean shift embedded particle filter," *Pattern Recognition*, vol. 40, 2007, pp. 1958-1970.

[70] E. Koller-Meier and F. Ade, "Tracking multiple objects using the condensation algorithm," *Robotics and Autonomous Systems*, 2001, pp. 1-18.

[71] R. Kalman, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, 1960, pp. 35-45.

[72] K. Dorfmuller-Ulhaas and D. Schmalstieg, "Finger tracking for interaction in augmented environments," *Proceedings IEEE and ACM International Symposium on Augmented Reality*, IEEE Comput. Soc, 2001, pp. 55-64.

[73] I. Laptev and T. Lindeberg, "Tracking of Multi-state Hand Models Using Particle Filtering and a Hierarchy of Multi-scale Image Features," *Scale-Space and Morphology in Computer Vision*, Berlin: Springer Berlin/ Heidelberg, 2001, pp. 63-74.

[74] J. MacCormick and M. Isard, "Partitioned sampling, articulated objects, and interface-quality hand tracking," *Lecture Notes in Computer Science*, vol. 1843, 2000, p. 3–19.

[75] A. Argyros and M. Lourakis, "Real-time tracking of multiple skin-colored objects with a possibly moving camera," *Lecture Notes in Computer Science*, 2004, p. 368–379.

[76] R. Dachselt and M. Hinz, "Three-dimensional widgets revisited-towards future standardization," *New directions in 3D user interfaces, Shaker Verlag*, 2005, p. 89–92.

[77] S. Zhang and S. Yau, "Three-dimensional shape measurement using a structured light system with dual cameras," *Optical Engineering*, vol. 47, 2008, p. 013604.

[78] D. Anguelov, D. Koller, H. Pang, P. Srinivasan, and S. Thrun, "Recovering articulated object models from 3D range data," *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, AUAI Press Arlington, Virginia, United States, 2004, p. 18–26.

Appendix: Project Source Code

Background.h

```
#ifndef __OPENCV200

      #include <opencv/cv.h>
      #include <opencv/cxcore.h>
      #include <opencv/highgui.h>

      #define __OPENCV200

#endif

#include <math.h>

#ifndef __BACKGROUNDSEGMENT

#define __BACKGROUNDSEGMENT

#define BACKGROUND_THRESHOLD 16

void getForegroundMask(IplImage*, IplImage*, IplImage*);
void getForegroundMask1(IplImage*, IplImage*, IplImage*);
void getForegroundMask2(IplImage*, IplImage*, IplImage*);

#endif
```

Background.cpp

```
#include "Background.h"

void getForegroundMask(IplImage *imgNew, IplImage *imgBackground,
IplImage* imgOut) {
      getForegroundMask1(imgNew, imgBackground, imgOut);
}

void getForegroundMask1(IplImage *imgNew, IplImage *imgBackground,
IplImage* imgOut) {

      // Manual background separation
      // Assume that imgNew and imgBackground are similarly sized
      // Tried HSV for segmentation, but BGR seems to work much
better...hmm.


      int width = imgNew->width;
      int height = imgNew->height;
      int widthStep = imgNew->widthStep;
      int channels = imgNew->nChannels;

      for(int i = 0; i < height ; i++) {
            for(int j = 0; j < width; j++) {

                  // This is crap, see if you can improve it.
                  int b = (int)((uchar *)(imgNew->imageData +
i*widthStep))[j*channels] - (int)((uchar *)(imgBackground->imageData +
i*widthStep))[j*channels];
                  int g = (int)((uchar *)(imgNew->imageData +
i*widthStep))[j*channels+1] - (int)((uchar *)(imgBackground->imageData
+ i*widthStep))[j*channels+1];
                  int r = (int)((uchar *)(imgNew->imageData +
i*widthStep))[j*channels+2] - (int)((uchar *)(imgBackground->imageData
+ i*widthStep))[j*channels+2];

                  if(  (abs(r) > BACKGROUND_THRESHOLD) || (abs(g) >
BACKGROUND_THRESHOLD) || (abs(b) > BACKGROUND_THRESHOLD) ) {
                        cvSetReal2D(imgOut, i, j, 255);
                  } else {
                        cvSetReal2D(imgOut, i, j, 0);
                  }
            }
      }
}

void getForegroundMask2(IplImage *imgNew, IplImage *imgBackground,
IplImage* imgOut) {

      // OpenCV differencing and thresholding

      IplImage* tmp = cvCreateImage(cvSize(imgNew->width, imgNew-
>height), imgNew->depth, imgNew->nChannels);

      // Find the differences between frames
      cvAbsDiff(imgNew, imgBackground, tmp);
      cvCvtColor(tmp,imgOut, CV_BGR2GRAY);
      cvThreshold(imgOut, imgOut, BACKGROUND_THRESHOLD, 255,
CV_THRESH_BINARY);
      //cvAdaptiveThreshold(imgOut, imgOut, 255,
CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY, 3, 5);

      // Use morphology to reduce noise
      cvErode(imgOut, imgOut, NULL, 1);
      cvDilate(imgOut, imgOut, NULL, 1);

      cvReleaseImage(&tmp);
}
```

Cam.h

```
#include <stdio.h>
#include <process.h>
#include <windows.h>
#include <stdlib.h>
#include <vector>

#ifndef __OPENCV200

      #include <opencv/cv.h>
      #include <opencv/cxcore.h>
      #include <opencv/highgui.h>

      #define __OPENCV200

#endif

#ifndef __AR
      #include <GL/gl.h>
      #include <GL/glut.h>

      #include <AR/gsub.h>
      #include <AR/gsub_lite.h>
      #include <AR/video.h>             // Needed?
      #include <AR/param.h>
      #include <AR/ar.h>
      #include <AR/arMulti.h>

      #define __AR
#endif

#include <osg/Camera>
#include <osg/Matrix>


#include "Background.h"
#include "Fingerpoint.h"

#define M_PI 3.14159265358979323846

#ifndef __CAM

#define __CAM

class Cam : public osg::Camera {

      public:
            Cam();
            Cam(int);
            ~Cam();
```

```
	static unsigned __stdcall threadedEntry( void *pThis) {
		Cam * pCam = (Cam*)pThis;
		pCam->threadProcess();
		return 1;
	}

	void threadProcess();

	void resetBackground();

	void setKey(int key) {
		//EnterCriticalSection( &m_CriticalSection );
		intKey = key;
		//LeaveCriticalSection( &m_CriticalSection );
	}

	void resetCamMatrix() {
		initCamMatrix();
	}

	void getCrit() {
		EnterCriticalSection( &critCandidates );
	}

	void releaseCrit() {
		LeaveCriticalSection( &critCandidates );
	}

	IplImage *imgOutput;
	cv::Mat imgForeMask;

	std::vector<cv::Rect> vOutRects;
	std::vector<cv::Mat> vOutROI;

	std::vector<osg::Vec3> vNear;
	std::vector<osg::Vec3> vFar;

	osg::Matrix matView;
	osg::Matrix matInvView;
	osg::Matrix matProjection;
	osg::Matrix matViewport;


private:
	CvCapture* vid;
	int intCamera;
	int intKey;

	void initialize();
	void initCamMatrix();
	void updateCamMatrix();
```

```
            // Working images
            IplImage *imgInput;
            IplImage *imgSmallInput;
            IplImage *imgBackground;
            IplImage *imgForeground;
            IplImage *imgMask;

            cv::Mat imgA, imgB, imgC;

            int intImageWidth;
            int intImageHeight;
            int intImageDepth;
            int intNear;
            int intFar;

            // Multithread via critical section
            CRITICAL_SECTION critCandidates;
            CRITICAL_SECTION critOutput;
            CRITICAL_SECTION critForeground;

            // AR Toolkit parameters - should rename to match my
convention
            int intTargetID;
            double dblTgtCenter[2];
            double dblTargetWidth;
            int intThreshold;

            ARMultiMarkerInfoT  *config;

            double dblTransform[3][4];
            double dblProjection[16];

            std::vector<cv::Mat> vROI;
            std::vector<cv::Rect> rects;


            // Histogram variables
            int hbins, sbins;                 // Number of levels to
quantize to.
            int histSize[2];
            float hranges[2];          // hue varies from 0 to 179, see
cvtColor
            float sranges[2];            // saturation varies from 0
(black-gray-white) to 255 (pure spectrum color)
            const float* ranges[2];
            int channels[2];                  // we compute the
histogram from the 0-th and 1-st channels (hue and saturation)

            cv::MatND histSkin, histNotSkin;
            double dMaxSkin, dMaxNotSkin;

            void initHist();
```

```
		void updateHist(cv::Mat *, cv::Mat *);

};



#endif
```

Cam.cpp

```
#include "Cam.h"
#include <time.h>

Cam::Cam() {
	intCamera = 0;
	initialize();
}

Cam::Cam(int value) {
	intCamera = value;
	initialize();
}

void Cam::initialize() {

	// Initialize crit section
	InitializeCriticalSection(&critCandidates);
	InitializeCriticalSection(&critOutput);
	InitializeCriticalSection(&critForeground);

	// Open camera feed
	vid = cvCaptureFromCAM(intCamera);
	if(!vid)
    {
		printf("Could not access camera %d.\n", intCamera);
        exit(0);
    }

	// Set resolution
	cvSetCaptureProperty( vid, CV_CAP_PROP_FRAME_WIDTH, 800);
	cvSetCaptureProperty( vid, CV_CAP_PROP_FRAME_HEIGHT, 600);
	cvSetCaptureProperty( vid, CV_CAP_PROP_FPS, 30);

	// Get frames from camera to allow for focus/aperture adjustment?
	imgInput = cvRetrieveFrame(vid);
	intImageWidth = imgInput->width;
	intImageHeight = imgInput->height;
	intImageDepth = imgInput->depth;

	// Set working images
	imgSmallInput = cvCreateImage(cvSize(imgInput->width/2, imgInput-
>height/2), imgInput->depth, imgInput->nChannels);
```

```
      cvPyrDown(imgInput, imgSmallInput, CV_GAUSSIAN_5x5);

      imgBackground = cvCreateImage(cvSize(intImageWidth/2,
intImageHeight/2), imgInput->depth, imgInput->nChannels);
      imgForeground = cvCreateImage(cvSize(intImageWidth/2,
intImageHeight/2), imgInput->depth, imgInput->nChannels);
      imgMask = cvCreateImage(cvSize(intImageWidth/2,
intImageHeight/2), imgInput->depth, 1);
      imgOutput = cvCreateImage(cvSize(intImageWidth, intImageHeight),
imgInput->depth, imgInput->nChannels);
      //imgForeOut = cvCreateImage(cvSize(intImageWidth,
intImageHeight), imgInput->depth, imgInput->nChannels+1);

      imgA = cv::Mat(intImageHeight/2,intImageWidth/2, CV_8U);
      imgB = cv::Mat(intImageHeight/2,intImageWidth/2, CV_8U);
      imgC = cv::Mat(intImageHeight/2,intImageWidth/2, CV_8U);


      initCamMatrix();

      //expose imgoutput for background initialization
      cvCopy(imgInput, imgOutput);

      // Set background image
      cvPyrDown(imgInput, imgBackground,CV_GAUSSIAN_5x5);

      //std::stringstream s;
      //s << "Cam" << intCamera;
      //cv::namedWindow(s.str(), 1);

      // Histogram setup
      hbins = 32, sbins = 32;                                  //
Number of levels to quantize to.
      histSize[0] = hbins, histSize[1] = sbins;
      hranges[0] = 0, hranges[1] = 256;
      sranges[0] = 0, sranges[1] = 256;
      ranges[0] = hranges, ranges[1] = sranges;
      channels[0] = 0, channels[1] = 1;                  // we compute
the histogram from the 0-th and 1-st channels (hue and saturation)

      initHist();


}

Cam::~Cam() {

      // Release AR resources?
    argCleanup();            //close out gsub resources - not sure if
necessary????
```

```
      // Release images
      cvReleaseImage(&imgSmallInput);
      cvReleaseImage(&imgBackground);
      cvReleaseImage(&imgForeground);
      cvReleaseImage(&imgMask);
      cvReleaseImage(&imgOutput);

      // Release capture
      cvReleaseCapture(&vid);

      cvDestroyAllWindows();

      // Release crit section
      DeleteCriticalSection(&critCandidates);
      DeleteCriticalSection(&critOutput);
      DeleteCriticalSection(&critForeground);

}

void Cam::resetBackground() {
      cvPyrDown(imgInput, imgBackground,CV_GAUSSIAN_5x5);
}

void Cam::initCamMatrix() {
      // Uses AR toolkit to establish camera position matrix

      ARParam arpInitialParam;
      ARParam arpCameraParam;

      // initialize global ar vars
      intThreshold = 100;
      dblTgtCenter[0] = 0.0;
      dblTgtCenter[1] = 0.0;
      dblTargetWidth = 80.0;

    // Set the initial camera parameters
    if( arParamLoad("trial2-3.dat", 1, &arpInitialParam) < 0 ) {
       printf("Camera parameter load error !!\n");
        exit(0);
    }

      // Adjust parameters
    arParamChangeSize( &arpInitialParam, intImageWidth, intImageHeight,
&arpCameraParam );
    arInitCparam( &arpCameraParam );

      // Load pattern
    if( (intTargetID = arLoadPatt("Patterns\\patt.kanji")) < 0 ) {
        printf("Target pattern load error!!\n");
        exit(0);
    }
```

```
      // Run the camera for 60 frames to gain an average position of
the marker
      for( int intFrame = 0; intFrame < 60; intFrame++) {

            // Get next input frame
            imgInput = cvRetrieveFrame(vid);              // Should
check if there is a better method of retrieval

            ARUint8         *arImageData;
            ARMarkerInfo    *armMarkerInfo;
            int             intMarker;

            // This conversion works b/c I configured ARToolkit to
expect BGR images instead of BGRA
            arImageData = (ARUint8 *)imgInput->imageData;

            // detect the markers in the video frame
            if( arDetectMarkerLite(arImageData, 128, &armMarkerInfo,
&intMarker) < 0 ) {
                  printf("Marker not detected!\n");
            }



            // Check for object visibility - sorts through all markers
and finds the best.
            // I need to find out if this accepts bad marker
matches...if so, determine a cf floor
            //int k = -1;
            //for( int j = 0; j < intMarker; j++ ) {
            //    if( armMarkerInfo[j].id == intTargetID ) {
            //          if( k == -1 ) k = j;
            //          else {
            //                if( armMarkerInfo[k].cf <
armMarkerInfo[j].cf ) k = j;
            //          }
            //    }
            //}

            // Get camera matrices
            if( intFrame == 0 ) {
                  if( arGetTransMat(armMarkerInfo, dblTgtCenter,
dblTargetWidth, dblTransform) < 0 ) continue;
            } else {
                  if( arGetTransMatCont( armMarkerInfo, dblTransform,
dblTgtCenter, dblTargetWidth, dblTransform) < 0) continue;
            }     // or something like that.

      }
      //Set up Camera matrices

      // Projection Matrix...not sure about near an far planes
      intNear = 20;           // in mm
      intFar = 1000;          // in mm
      arglCameraFrustumRH(&arpCameraParam, (float)intNear,
(float)intFar, dblProjection );

      matProjection = osg::Matrix(dblProjection[0], dblProjection[1],
dblProjection[2], dblProjection[3],
                                               dblProjection[4],
dblProjection[5], dblProjection[6], dblProjection[7],
                                                dblProjection[8],
dblProjection[9], dblProjection[10], dblProjection[11],
                                                dblProjection[12],
dblProjection[13], dblProjection[14], dblProjection[15] );
      this->setProjectionMatrix( matProjection );


      // View Matrix
      double arr[16];
      argConvGlpara(dblTransform, arr);
      matView = osg::Matrix(arr[0], arr[1], arr[2], arr[3],
                               arr[4], arr[5], arr[6], arr[7],
                               arr[8], arr[9], arr[10], arr[11],
                               arr[12], arr[13], arr[14], arr[15]
);

      // make an adjustment for OSG vs. ART camera orientation
expectations
      osg::Matrixd matCamRotation;
      matCamRotation.makeRotate( M_PI, osg::Vec3(1,0,0),
                                          0, osg::Vec3(0,1,0),
                                          0, osg::Vec3(0,0,1) );

      matCamRotation = osg::Matrixd::inverse( matCamRotation );
      matView.postMult( matCamRotation );

      // Create inverse in case you need to position items???
      matInvView = osg::Matrixd::inverse( matView );
      this->setViewMatrix( matView );

      double dblInv[3][4];
      arUtilMatInv(dblTransform, dblInv);

      printf("Camera %d: X: %2.2f, Y: %2.2f, Z:%2.2f\n", intCamera,
dblInv[0][3], dblInv[1][3], dblInv[2][3]);

      // Viewport
      osg::ref_ptr<osg::Viewport> v = new osg::Viewport();
      v->setViewport(0,0,intImageWidth/2,intImageHeight/2);
      matViewport = v->computeWindowMatrix();
      argCleanup();

}

void Cam::threadProcess() {


      // Constantly process frames until escape is pressed in parent
loop
      while(intKey != 27) {

            // Get next input frame
            imgInput = cvRetrieveFrame(vid);              // Should
check if there is a better method of retrieval

            // Shrink down to minimize processing
            cvPyrDown(imgInput, imgSmallInput, CV_GAUSSIAN_5x5);


            // Get background mask
            getForegroundMask(imgSmallInput, imgBackground, imgMask);

            // Get foreground
            cvZero( imgForeground );
            cvCopy(imgSmallInput, imgForeground, imgMask);


            // Get color masks - need mats instead of images here
            cv::Mat mtx(imgForeground);
            cv::Mat hsv;

            cv::cvtColor(mtx, hsv, CV_BGR2HSV);
            cv::calcBackProject( &hsv, 1, channels, histSkin, imgA,
ranges, 1.0, true);
            //cv::calcBackProject( &hsv, 1, channels, histNotSkin,
imgB, ranges, 1.0, true);

            // Find skin probability - P(pixel|skin) / P(pixel|notSkin)
>= theta
            //for(int i = 0; i < hsv.rows-1; i++) {
            //    for(int j = 0; j < hsv.cols-1; j++) {
            //          // Get current h pixel, s pixel
            //          cv::Vec3b value = hsv.at<cv::Vec3b>(i,j);

            //          // determine the bin for hPix & sPix via divide
by hbins and sbins
            //          int hPix = value[0] / hbins;
            //          int sPix = value[1] / sbins;

            //          // look up skin hist value and sat hist value -
scale by cvRound(binVal*255/maxvalue)
```

```
            //          double dSkin = histSkin.at<float>(hPix, sPix) *
255.0 / dMaxSkin;
            //          double dNotSkin = histNotSkin.at<float>(hPix,
sPix) * 255.0 / dMaxNotSkin;
            //          //double dNotSkin = 1.0;

            //          // calc ratio of skin/nonskin value for this
pixel & compare to theta.
            //          if( (dSkin / dNotSkin) >= 1.0 ) {
            //                imgC.at<uchar>(i,j) = 255;
            //          } else {
            //                imgC.at<uchar>(i,j) = 0;
            //          }
            //    }
            //}

            //for(int i = 0; i < imgA.rows-1; i++) {
            //    for(int j = 0; j < imgB.cols-1; j++) {
            //          double dRatio = ((double)(imgA.at<uchar>(i,j)) /
(double)(imgB.at<uchar>(i,j))) ;
            //          if( dRatio > 1.0 ) {
            //                imgC.at<uchar>(i,j) = 255;
            //          } else {
            //                imgC.at<uchar>(i,j) = 0;
            //          }
            //    }
            //}

            //std::stringstream s;
            //s << "Cam" << intCamera;
            //cv::imshow( s.str(), imgA);


            //Threshold & morph to filter
            cv::threshold(imgA, imgA, 250, 255, cv::THRESH_BINARY);
            cv::morphologyEx(imgA, imgA, cv::MORPH_OPEN,
cv::Mat(3,3,1,1.0), cvPoint(1,1), 1,0,0);
            cv::morphologyEx(imgA, imgA, cv::MORPH_CLOSE,
cv::Mat(3,3,1,1.0), cvPoint(1,1), 1,0,0);


            //std::stringstream s;
            //s << "Cam" << intCamera;
            //cv::imshow( s.str(), imgA);

            // Update histograms
            //updateHist( &(cv::Mat(imgSmallInput)), &imgA); - this
gets out of hand really quickly...false positives explode.

            // Get foreground
            //cv::Mat fore;
            //cv::Mat(imgInput).copyTo(fore, cv::Mat());
            // Copy input into output in case it is used for AR
            EnterCriticalSection( &critOutput );
            cvCopy(imgInput, imgOutput);
            imgA.copyTo(imgForeMask, cv::Mat() );
            LeaveCriticalSection( &critOutput );




            // Get fingertip points
            std::vector<cv::Point2f> pt;
            rects.clear();
            findCandidates3(imgA, &pt, &rects);

            // Get rectangle ROIs of foreground
            //vROI.clear();
            //for(int i = 0; i < rects.size(); i++) {
            //    cv::Mat tmp;
            //    cv::getRectSubPix(fore, cv::Size(rects[i].width*2,
rects[i].height*2), cv::Point2f(rects[i].x*2 + rects[i].width,
rects[i].y*2 + rects[i].height),tmp,-1);

            //    cv::Mat roi = cv::Mat(tmp.rows, tmp.cols, CV_8UC4,
cv::Scalar(0,0,0,0));
            //    // changing channels,so can't use copyto op
            //    for(int j = 0; j < tmp.rows; j++) {
            //          for(int k = 0; k < tmp.cols; k++) {
            //                if( imgA.at<uchar>(rects[i].y+(j/2),
rects[i].x+(k/2)) > 0 ) {
            //                      cv::Vec3b tmpPt =
tmp.at<cv::Vec3b>(j,k);
            //                      cv::Vec4b tmpPt2 =
cv::Vec4b(tmpPt[0], tmpPt[1], tmpPt[2], 255);
            //                      roi.at<cv::Vec4b>(j,k) = tmpPt2;
            //                }
            //          }
            //    }

            //    vROI.push_back( roi );
            //}



            //std::stringstream s;
            //s << "Cam" << intCamera;
            //cv::Mat ptOut = cv::Mat(imgSmallInput);
            //for(int j = 0; j < pt.size(); j++)
            //    cv::rectangle( ptOut, cv::Point(pt[j].x-1, pt[j].y-1),
cv::Point(pt[j].x+1, pt[j].y+1),cv::Scalar(0,0,255,0),2,8,0);

            //cv::imshow( s.str(),  ptOut);
            // Calculate points in 3D space - checked math.
            // very large z values gets you to the cam.  z of 1 gets
you to the far plane; I don't know why it works...but it works.
            osg::Matrix mat;
            mat.invert(matView * matProjection * matViewport);

            // Get cam coords into world coords
            EnterCriticalSection( &critCandidates );
            vNear.clear();
            vFar.clear();
            for(int i = 0; i < pt.size(); i++) {
                  // Get cam coords into world coords, then multiply by
matrices to create ray
                  vNear.push_back( osg::Vec3(pt[i].x, (intImageHeight/2)
- pt[i].y, 1000) * mat );
                  vFar.push_back( osg::Vec3(pt[i].x, (intImageHeight/2)
- pt[i].y, 1) * mat );
            }
            vOutRects.clear();
            //vOutROI.clear();
            for(int i = 0; i < rects.size(); i++) {
                  vOutRects.push_back( rects[i]);
                  //vOutROI.push_back( vROI[i] );
            }
            LeaveCriticalSection( &critCandidates );

      }
}

void Cam::initHist() {
      // Create histograms for further use

      cv::Mat src, hsv;

      // Create skin histogram
      if( !(src=cv::imread("skin2.jpg", 1)).data )  //this will not
work if you mix release/debug libraries
        exit(-1);
      cv::cvtColor(src, hsv, CV_BGR2HSV);


      cv::calcHist( &hsv, 1, channels, cv::Mat(), // do not use mask
        histSkin, 2, histSize, ranges,
        true, // the histogram is uniform
        false );

      // Get max value for later use
      cv::minMaxLoc(histSkin, 0, &dMaxSkin, 0, 0);


      // Create non-skin histogram
      //if( !(src=cv::imread("notSkin.jpg", 1)).data )    //this will
not work if you mix release/debug libraries
 //       exit(-1);
 //   cvtColor(src, hsv, CV_BGR2HSV);

      //cv::calcHist( &hsv, 1, channels, cv::Mat(), // do not use mask
 //       histNotSkin, 2, histSize, ranges,
 //       true, // the histogram is uniform
 //       false );

      //// Get max value for later use
      //cv::minMaxLoc(histNotSkin, 0, &dMaxNotSkin, 0, 0);

}

void Cam::updateHist(cv::Mat *frame, cv::Mat *mask) {

      //Update histograms with found pixels
      cv::Mat notMask;
      cv::subtract( 255, *mask, notMask, cv::Mat() );          //Not
sure this is what I want exactly...I want cvNot. :p

      cv::calcHist( frame, 1, channels, *mask, histSkin, 2, histSize,
ranges, true, true );
      cv::minMaxLoc(histSkin, 0, &dMaxSkin, 0, 0);

      cv::calcHist( frame, 1, channels, notMask, histNotSkin, 2,
histSize, ranges, true, true );
      cv::minMaxLoc(histNotSkin, 0, &dMaxNotSkin, 0, 0);

}
```

Fingerpoint.h

```
#ifndef __OPENCV200

      #include <opencv/cv.h>
      #include <opencv/cxcore.h>
      #include <opencv/highgui.h>

      #define __OPENCV200

#endif

#include <math.h>

#ifndef __FINGERPOINTS

#define __FINGERPOINTS

void getDominantPoints(CvSeq*, IplImage*);
void getDominantPoints1(CvSeq*, IplImage*);
void getDominantPoints2(CvSeq*, IplImage*);
void findCandidates(IplImage* imgIn, CvPoint2D32f
arrCandidates[4][100], int intSize[4], int intMaxSize);

void findCandidates2(cv::Mat, std::vector<cv::Point2f> *);
void findCandidates3(cv::Mat, std::vector<cv::Point2f> *,
std::vector<cv::Rect> *);

#endif
```

Fingerpoint.cpp

```
#include "Fingerpoint.h"

void getDominantPoints(CvSeq* contour, IplImage *img) {
	getDominantPoints1(contour, img);
}

void getDominantPoints1(CvSeq* contour, IplImage* img) {

	// This method searches out points on the contour that are
farther from the center of mass than
	// the adjacent two points on the curve

	//cvDrawContours(img, contour, CV_RGB(0,0,255),
CV_RGB(0,0,0),2,1,8,cvPoint(0,0));

	while(contour != NULL) {
		// Only process large contours
		double dblContourArea =
fabs(cvContourArea(contour,CV_WHOLE_SEQ));

		if( dblContourArea > 200 ) {

			// Get a set of points around the contour perimeter
			CvMemStorage *storagePoints = cvCreateMemStorage(0);
			CvSeq *contourPoints = 0;
			contourPoints = cvApproxPoly(contour,
sizeof(CvContour), storagePoints, CV_POLY_APPROX_DP,
cvContourPerimeter(contour) * 0.01, 1);

			// Convert to an array for easy access - necessary or
fast?
			CvPoint* arrPoints = (CvPoint *)malloc(contourPoints-
>total * sizeof(CvPoint));
			cvCvtSeqToArray(contourPoints, arrPoints,
CV_WHOLE_SEQ);

			// Get center of contour
			CvPoint2D32f center;
			CvMoments* moments = new CvMoments();
```

```
                cvContourMoments(contour, moments);
                center.x = (float)(moments->m10/moments->m00);
                center.y = (float)(moments->m01/moments->m00);

                cvRectangle(img, cvPoint((int)center.x - 1,
(int)center.y - 1), cvPoint((int)center.x + 1, (int)center.y + 1),
CV_RGB(0,255,0),2,8,0);

                // select concave points
                for(int i = 0; i < contourPoints->total; i++) {

                      double x1, x2, x3;
                      double y1, y2, y3;
                      double dist1, dist2, dist3;

                      if( i == 0 ) {
                            x1 = arrPoints[contourPoints->total-1].x -
center.x;
                            x2 = arrPoints[i].x - center.x;
                            x3 = arrPoints[i+1].x - center.x;

                            y1 = arrPoints[contourPoints->total-1].y -
center.y;
                            y2 = arrPoints[i].y - center.y;
                            y3 = arrPoints[i+1].y - center.y;
                      } else if (i < contourPoints->total - 1 ) {
                            x1 = arrPoints[i-1].x - center.x;
                            x2 = arrPoints[i].x - center.x;
                            x3 = arrPoints[i+1].x - center.x;

                            y1 = arrPoints[i-1].y - center.y;
                            y2 = arrPoints[i].y - center.y;
                            y3 = arrPoints[i+1].y - center.y;
                      } else if (i == contourPoints->total - 1) {
                            x1 = arrPoints[i-1].x - center.x;
                            x2 = arrPoints[i].x - center.x;
                            x3 = arrPoints[0].x - center.x;

                            y1 = arrPoints[i-1].y - center.y;
                            y2 = arrPoints[i].y - center.y;
                            y3 = arrPoints[0].y - center.y;
                      }

                      dist1 = x1*x1 + y1*y1;
                      dist2 = x2*x2 + y2*y2;
                      dist3 = x3*x3 + y3*y3;

                      if( (dist2 >= dist1) && (dist2 >= dist3) ) {
                            cvRectangle(img, cvPoint(arrPoints[i].x-
1,arrPoints[i].y-1),
cvPoint(arrPoints[i].x+1,arrPoints[i].y+1),CV_RGB(255,0,0), 2, 8, 0);
                      }
```

```
                    }

                    // Clean up
                    cvReleaseMemStorage(&storagePoints);
                    free(arrPoints);
              }

              // Increment to next contour
              if(contour->h_next)    {
                    contour = contour->h_next;
              }
              else {
                    contour = NULL;
              }
       }
}
void getDominantPoints2(CvSeq* contour, IplImage* img) {

       // This method finds the convex hull; however, it has more points
than just the individual fingertips,
       // so I don't think I'll use this method.

       while(contour != NULL) {
              // Only process large contours
              double dblContourArea =
fabs(cvContourArea(contour,CV_WHOLE_SEQ));

              if( dblContourArea > 100 ) {
                    // Get a set of points around the contour perimeter
                    CvMemStorage *storagePoints = cvCreateMemStorage(0);
                    CvSeq *contourPoints = 0;
                    contourPoints = cvConvexHull2(contour, storagePoints,
CV_CLOCKWISE,1);

                    // Convert to an array for easy access - necessary or
fast?
                    CvPoint* arrPoints = (CvPoint *)malloc(contourPoints-
>total * sizeof(CvPoint));
                    cvCvtSeqToArray(contourPoints, arrPoints,
CV_WHOLE_SEQ);

                    // Get center of contour
                    CvPoint2D32f center;
                    CvMoments* moments = new CvMoments();
                    cvContourMoments(contour, moments);
                    center.x = (float)(moments->m10/moments->m00);
                    center.y = (float)(moments->m01/moments->m00);

                    // draw points
                    cvRectangle(img, cvPoint((int)center.x - 1,
(int)center.y - 1), cvPoint((int)center.x + 1, (int)center.y + 1),
CV_RGB(0,255,0),2,8,0);
```

```
                for(int i = 0; i < contourPoints->total; i++) {
                    cvRectangle(img, cvPoint(arrPoints[i].x-
1,arrPoints[i].y-1),
cvPoint(arrPoints[i].x+1,arrPoints[i].y+1),CV_RGB(255,0,0), 2, 8, 0);
                }

                // Clean up
                cvReleaseMemStorage(&storagePoints);
                free(arrPoints);
            }

            // Increment to next contour
            if(contour->h_next)    {
                contour = contour->h_next;
            }
            else {
                contour = NULL;
            }
        }
}


void findCandidates(IplImage* imgIn, CvPoint2D32f
arrCandidates[4][100], int intSize[4], int intMaxSize=100) {

        // This method searches out points on the contour that are
farther from the center of mass than
        // the adjacent two points on the curve

        CvMemStorage *storage = cvCreateMemStorage(0);
        CvSeq * contour = 0;
        cvFindContours(imgIn, storage, &contour, sizeof(CvContour),
CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0));

        CvSeq * largestContour[4] = {0,0,0,0};;
        double fArea[4] = {0,0,0,0};

        while(contour != NULL) {
            // Only process largest contours
            double dblContourArea =
fabs(cvContourArea(contour,CV_WHOLE_SEQ));

            for(int i = 0; i < 4; i++) {
                if( dblContourArea > fArea[i] ) {

                    for(int j = i+1; j < 4; j++) {
                        largestContour[j] = largestContour[j-1];
                        fArea[j] = fArea[j-1];
                    }

                    largestContour[i] = contour;
                    fArea[i] = dblContourArea;
```

```
                        break;
                  }
            }

            // Increment to next contour
            if(contour->h_next)    {
                  contour = contour->h_next;
            }
            else {
                  contour = NULL;
            }

      }

      for(int i = 0; i < 4; i++ ) {
            intSize[i] = 0;

            if( fArea[i] > 200 ) {

                  // Get a set of points around the contour perimeter
                  CvMemStorage *storagePoints = cvCreateMemStorage(0);
                  CvSeq *contourPoints = 0;
                  contourPoints = cvApproxPoly(largestContour[i],
sizeof(CvContour), storagePoints, CV_POLY_APPROX_DP,
cvContourPerimeter(largestContour[i]) * 0.02, 1);

                  // Convert to an array for easy access - necessary or
fast?
                  CvPoint* arrPoints = (CvPoint *)malloc(contourPoints-
>total * sizeof(CvPoint));
                  cvCvtSeqToArray(contourPoints, arrPoints,
CV_WHOLE_SEQ);

                  // Get center of contour
                  CvPoint2D32f center;
                  CvMoments* moments = new CvMoments();
                  cvContourMoments(largestContour[i], moments);
                  center.x = (float)(moments->m10/moments->m00);
                  center.y = (float)(moments->m01/moments->m00);
                  arrCandidates[i][0] = center;
                  intSize[i]++;

                  // select concave points
                  for(int k = 0; k < contourPoints->total; k++) {

                        double x1, x2, x3;
                        double y1, y2, y3;
                        double dist1, dist2, dist3;

                        if( k == 0 ) {
                              x1 = arrPoints[contourPoints->total-1].x -
center.x;
```

```
                              x2 = arrPoints[k].x - center.x;
                              x3 = arrPoints[k+1].x - center.x;

                              y1 = arrPoints[contourPoints->total-1].y -
center.y;
                              y2 = arrPoints[k].y - center.y;
                              y3 = arrPoints[k+1].y - center.y;
                        } else if (k < contourPoints->total - 1 ) {
                              x1 = arrPoints[k-1].x - center.x;
                              x2 = arrPoints[k].x - center.x;
                              x3 = arrPoints[k+1].x - center.x;

                              y1 = arrPoints[k-1].y - center.y;
                              y2 = arrPoints[k].y - center.y;
                              y3 = arrPoints[k+1].y - center.y;
                        } else if (k == contourPoints->total - 1) {
                              x1 = arrPoints[k-1].x - center.x;
                              x2 = arrPoints[k].x - center.x;
                              x3 = arrPoints[0].x - center.x;

                              y1 = arrPoints[k-1].y - center.y;
                              y2 = arrPoints[k].y - center.y;
                              y3 = arrPoints[0].y - center.y;
                        }

                        dist1 = x1*x1 + y1*y1;
                        dist2 = x2*x2 + y2*y2;
                        dist3 = x3*x3 + y3*y3;

                        if( (dist2 >= dist1) && (dist2 >= dist3) &&
(intSize[i] < intMaxSize)) {
                              arrCandidates[i][intSize[i]].x =
arrPoints[k].x;
                              arrCandidates[i][intSize[i]].y =
arrPoints[k].y;
                              intSize[i]++;
                        }

                  }

                  // Clean up
                  delete moments;
                  cvReleaseMemStorage(&storagePoints);
                  contourPoints = NULL;
                  free(arrPoints);
                  arrPoints = NULL;

            }
      }

      cvReleaseMemStorage( &storage );
}
```

```
void findCandidates2(cv::Mat imgIn, std::vector<cv::Point2f> *pts) {

      // This method searches out points on the contour that are
farther from the center of mass than
      // the adjacent two points on the curve

      std::vector<std::vector<cv::Point>> contours;
      std::vector<cv::Point2f> tmp;
      cv::Moments mom;

      pts->clear();
      cv::findContours( imgIn, contours, cv::RETR_EXTERNAL ,
cv::CHAIN_APPROX_SIMPLE, cv::Point(0,0));


      double dAreaA = 100;
      for(int i = 0; i < contours.size(); i++) {

            std::vector< cv::Point2f > fContours;
            for(int j = 0; j < contours[i].size(); j++) {
                  fContours.push_back( contours[i][j] );
            }
            cv::Mat mat = cv::Mat( fContours );


            double d = fabs(cv::contourArea( mat ));
            if( d > dAreaA) {

                  bool t = mat.isContinuous();
                  bool u = (mat.depth() == CV_32F);
            bool v = ((mat.rows == 1 && mat.channels() == 2) ||
(mat.cols * mat.channels() == 2));

                  cv::approxPolyDP(mat, tmp, cv::arcLength(mat,1)*0.02,
1);

                  mom = cv::moments( cv::Mat(contours[i]), false );
                  cv::Point2f center =
cv::Point2f((float)(mom.m10/mom.m00),(float)(mom.m01/mom.m00));

                  for(int j = 0; j < tmp.size(); j++) {
                        cv::Point2f t1, t2, t3;

                        double dist1, dist2, dist3;

                        if( j == 0 ) {
                              t1 = tmp[tmp.size()-1] - center;
                              t2 = tmp[j] - center;
                              t3 = tmp[j+1] - center;
                        } else if (j < tmp.size() - 1 ) {
                              t1 = tmp[j-1] - center;
                        t2 = tmp[j] - center;
                        t3 = tmp[j+1] - center;
                    } else if (j == tmp.size() - 1) {
                        t1 = tmp[j-1] - center;
                        t2 = tmp[j] - center;
                        t3 = tmp[0] - center;
                    }

                    dist1 = t1.x*t1.x + t1.y*t1.y;
                    dist2 = t2.x*t2.x + t2.y*t2.y;
                    dist3 = t3.x*t3.x + t3.y*t3.y;

                    if( (dist2 >= dist1) && (dist2 >= dist3) ) {
                        pts->push_back( tmp[j] );
                    }
                }
            }
        }
        contours.clear();

}

void findCandidates3(cv::Mat imgIn, std::vector<cv::Point2f> *pts,
std::vector<cv::Rect> *rects) {

        // This method searches out points on the contour that are
farther from the center of mass than
        // the adjacent two points on the curve

        std::vector<std::vector<cv::Point>> contours;
        std::vector<cv::Point2f> tmp;
        cv::Moments mom;

        pts->clear();
        cv::findContours( imgIn, contours, cv::RETR_EXTERNAL ,
cv::CHAIN_APPROX_SIMPLE, cv::Point(0,0));


        double dAreaA = 100;
        for(int i = 0; i < contours.size(); i++) {

            std::vector< cv::Point2f > fContours;
            for(int j = 0; j < contours[i].size(); j++) {
                fContours.push_back( contours[i][j] );
            }
            cv::Mat mat = cv::Mat( fContours );


            double d = fabs(cv::contourArea( mat ));
            if( d > dAreaA) {

                cv::Rect tmpRect = cv::boundingRect( mat );
```

```
                  rects->push_back( tmpRect );

                  cv::approxPolyDP(mat, tmp, cv::arcLength(mat,1)*0.02,
1);

                  mom = cv::moments( cv::Mat(contours[i]), false );
                  cv::Point2f center =
cv::Point2f((float)(mom.m10/mom.m00),(float)(mom.m01/mom.m00));

                  for(int j = 0; j < tmp.size(); j++) {
                        cv::Point2f t1, t2, t3;

                        double dist1, dist2, dist3;

                        if( j == 0 ) {
                              t1 = tmp[tmp.size()-1] - center;
                              t2 = tmp[j] - center;
                              t3 = tmp[j+1] - center;
                        } else if (j < tmp.size() - 1 ) {
                              t1 = tmp[j-1] - center;
                              t2 = tmp[j] - center;
                              t3 = tmp[j+1] - center;
                        } else if (j == tmp.size() - 1) {
                              t1 = tmp[j-1] - center;
                              t2 = tmp[j] - center;
                              t3 = tmp[0] - center;
                        }

                        dist1 = t1.x*t1.x + t1.y*t1.y;
                        dist2 = t2.x*t2.x + t2.y*t2.y;
                        dist3 = t3.x*t3.x + t3.y*t3.y;

                        if( (dist2 >= dist1) && (dist2 >= dist3) ) {
                              pts->push_back( tmp[j] );
                        }
                  }
            }
      }
      contours.clear();

}
```

IntersectionTester.h

```
#include <process.h>
#include <windows.h>
#include <stdlib.h>
#include <vector>

#include <osg/Matrix>
#include <osg/Vec3>
#include "PtFilter.h"
```

```
#define NUM_CAMS 3


#ifndef __INTERSECTIONTESTER

#define __INTERSECTIONTESTER
#define PERSIST_FRAMES 15         // 0.5 second


class IntersectionTester {

      public:
            IntersectionTester();
            ~IntersectionTester();

            static unsigned __stdcall threadedEntry( void *pThis) {
                  IntersectionTester * pInt =
(IntersectionTester*)pThis;
                  pInt->threadProcess();
                  return 1;
            }

            void threadProcess();

            void setInputPts(std::vector< std::vector<osg::Vec3> >
vStart, std::vector< std::vector<osg::Vec3> > vEnd) {
                  EnterCriticalSection( &critInput );
                  vStartPts = vStart;
                  vEndPts = vEnd;
                  bNewInputs = true;
                  LeaveCriticalSection( &critInput );
            }

            void setKey( int s ) {
                  intKey = s;
            }

            void setThreshold( int s ) {
                  intThreshold = s*s;
            }

            void getCrit() {
                  EnterCriticalSection( &critOutput );
            }

            void leaveCrit() {
                  LeaveCriticalSection( &critOutput );
            }

            std::vector<osg::Vec3> getOutput() {
                // Create a copy for output use; this may not work
with the widget manipulators,
                // may need to change to a pointer.
                std::vector<osg::Vec3> tmp;
                EnterCriticalSection( &critOutput );
                tmp = vOutputPts;
                LeaveCriticalSection( &critOutput );

                return tmp;
           }

           void getOutputs(std::vector<osg::Vec3> * pts,
std::vector<int> *indices) {
                EnterCriticalSection( &critOutput );
                *pts = vOutputPts;
                *indices = vDeletedIndices;
                vDeletedIndices.clear();
                LeaveCriticalSection( &critOutput );

           }

private:
           // Multithread via critical section
           CRITICAL_SECTION critInput;
           CRITICAL_SECTION critOutput;

           // input vectors
           std::vector< std::vector<osg::Vec3> > vStartPts;
           std::vector< std::vector<osg::Vec3> > vEndPts;
           bool bNewInputs;

           bool intersect(osg::Vec3, osg::Vec3, osg::Vec3, osg::Vec3,
osg::Vec3*, osg::Vec3*);
           int intKey;
           int intThreshold;
           double EPS;

           // output objects
           std::vector<osg::Vec3> vOutputPts;
           std::vector<int> vDeletedIndices;

};

#endif
```

IntersectionTester.cpp

```
#include "IntersectionTester.h"

IntersectionTester::IntersectionTester() {

     InitializeCriticalSection( &critInput );
```

```
	InitializeCriticalSection( &critOutput );

	for(int i = 0; i < NUM_CAMS; i++) {
		std::vector< osg::Vec3 > *tmp = new std::vector< osg::Vec3 
>;
		vStartPts.push_back( *tmp );

		std::vector< osg::Vec3 > *tmp2 = new std::vector< osg::Vec3 
>;
		vEndPts.push_back( *tmp2 );
	}

	intKey = 0;
	intThreshold = 25;	// initialized to a distance of 5 mm
	EPS = 1.0E-3;
}

IntersectionTester::~IntersectionTester() {
	// Release crit section
	DeleteCriticalSection( &critInput );
	DeleteCriticalSection( &critOutput );
}

void IntersectionTester::threadProcess() {

	// Test for intersections or near intersections of rays described 
by vec3 arrays
	osg::Vec3 startA, startB, endA, endB;
	osg::Vec3 * startC = new osg::Vec3();
	osg::Vec3 * endC = new osg::Vec3();
	osg::Vec3 delta;
	int distance;

	// working vectors
	std::vector< std::vector< osg::Vec3 > > vTmpStPts, vTmpEndPts;
	std::vector< osg::Vec3 > vTmpOutPts, vStablePts;

	std::vector< osg::Vec3 > vPosPts, vVelPts;
	std::vector< int > vUpdateAttempts;
	std::vector<int> vTmpDelIndices;

	while(intKey != 27 ) {

		if( bNewInputs ) {

			// clear old vector
			vTmpOutPts.clear();

			// Copy input vectors
			EnterCriticalSection( &critInput );
			vTmpStPts = vStartPts;
			vTmpEndPts = vEndPts;
```

```
                LeaveCriticalSection( &critInput );

                bNewInputs = false;

                // Find intersections between lines
                for( int i = 0; i < NUM_CAMS; i++ ) {
                      for( int j = 0; j < vTmpStPts[i].size(); j++ ) {

                            // Get first line
                            startA = vTmpStPts[i][j];
                            endA = vTmpEndPts[i][j];

                            for( int x = i + 1; x < NUM_CAMS; x++ ) {
      // only test other camera's lines, no self-intersections please
                                  for( int y = 0; y <
vTmpStPts[x].size(); y++ ) {

                                        // Get second line
                                        startB = vTmpStPts[x][y];
                                        endB = vTmpEndPts[x][y];

                                        // Get shortest distance
between the rays
                                        intersect(startA, endA, startB,
endB, startC, endC);
                                        delta = osg::Vec3( startC->x()
- endC->x(), startC->y() - endC->y(), startC->z() - endC->z() );
                                        distance = delta.x() *
delta.x() + delta.y() * delta.y() + delta.z() * delta.z();

                                        // If the distance is less than
the threshold we add a point
                                        if( distance <= intThreshold )
{
                                              vTmpOutPts.push_back(
osg::Vec3( startC->x() + 0.5*delta.x(),

                         startC->y() + 0.5*delta.y(),

                         startC->z() + 0.5*delta.z() ) );
                                        }
                                  }
                            }
                      }
                }

                // Test points for duplicates
                bool bMatchesExist = true;

                while( bMatchesExist ) {

                      int intMinDist = 500;  // 2 cm squared
```

```
                        int index1 = -1;
                        int index2 = -1;

                        // Find closest pair of points within the
minimum distance
                        for(unsigned int i = 0; i < vTmpOutPts.size();
i++ ) {
                              for(unsigned int j = 0; j <
vTmpOutPts.size(); j++ ) {
                                    if( i != j ) {
                                          osg::Vec3 pt1 = vTmpOutPts[i];
                                          osg::Vec3 pt2 = vTmpOutPts[j];

                                          delta = osg::Vec3d( pt1.x() -
pt2.x(),

      pt1.y() - pt2.y(),

      pt1.z() - pt2.z() );
                                          distance = delta.x() *
delta.x() + delta.y() * delta.y() + delta.z() * delta.z();

                                          // If closer than any other
match, update the indices and distance to test against
                                          if( distance < intMinDist ) {
                                                intMinDist = distance;
                                                index1 = i;
                                                index2 = j;
                                          }
                                    }
                              }
                        }

                        // If points found within the minimum distance,
merge them 50/50.
                        if( index1 >= 0 && index2 >= 0 ) {
                              osg::Vec3 pt1 = vTmpOutPts[index1];
                              osg::Vec3 pt2 = vTmpOutPts[index2];

                              vTmpOutPts[index1] = osg::Vec3( (pt1.x() +
pt2.x()) / 2.0,

      (pt1.y() + pt2.y()) / 2.0,

      (pt1.z() + pt2.z()) / 2.0 );

                              vTmpOutPts.erase(vTmpOutPts.begin() +
index2);
                        } else {
                              bMatchesExist = false;
                        }
```

```
                }

                // Tracking Section

                // Update positions
                for(int i = 0; i < vPosPts.size(); i++)
                      vPosPts[i] = vPosPts[i] + vVelPts[i];

                // Match new points to old points
                bMatchesExist = true;
                std::vector<int> vOldIndices, vNewIndices;

                for(int i = 0; i < vPosPts.size(); i++)
                      vOldIndices.push_back( i );

                for(int i = 0; i < vTmpOutPts.size(); i++)
                      vNewIndices.push_back( i );

                while( bMatchesExist ) {

                      int intMinDist = 625;  // 3 cm squared
                      int index1 = -1;
                      int index2 = -1;

                      // Find closest pair of points within the
minimum distance
                      for(unsigned int i = 0; i < vOldIndices.size();
i++ ) {
                            for(unsigned int j = 0; j <
vNewIndices.size(); j++ ) {
                                  osg::Vec3 pt1 =
vPosPts[vOldIndices[i]];
                                  osg::Vec3 pt2 =
vTmpOutPts[vNewIndices[j]];

                                  delta = osg::Vec3d( pt1.x() -
pt2.x(),
                                                                  pt1.y()
- pt2.y(),
                                                                  pt1.z()
- pt2.z() );
                                  distance = delta.x() * delta.x() +
delta.y() * delta.y() + delta.z() * delta.z();

                                  // If closer than any other match,
update the indices and distance to test against
                                  if( distance < intMinDist ) {
                                        intMinDist = distance;
                                        index1 = i;
                                        index2 = j;
                                  }
                            }
```

```
                        }

                        double coeff1 = 29.0;
                        double coeff2 = 1.0;

                        // If points found within the minimum distance,
merge them 3:2
                        if( index1 >= 0 && index2 >= 0 ) {
                              osg::Vec3 pt1 =
vPosPts[vOldIndices[index1]];
                              osg::Vec3 pt2 =
vTmpOutPts[vNewIndices[index2]];

                              // Update position
                              vPosPts[vOldIndices[index1]] = osg::Vec3(
(coeff1*pt1.x() + coeff2*pt2.x()) / (coeff1 + coeff2),

                    (coeff1*pt1.y() + coeff2*pt2.y()) / (coeff1 +
coeff2),

                    (coeff1*pt1.z() + coeff2*pt2.z()) / (coeff1 +
coeff2) );
                              // Update velocity
                              osg::Vec3 v1 =
vVelPts[vOldIndices[index1]];
                              osg::Vec3 v2 = pt2 - pt1;

                              vVelPts[vOldIndices[index1]] = osg::Vec3(
(coeff1*v1.x() + coeff2*v2.x()) / (coeff1 + coeff2),

                    (coeff1*v1.y() + coeff2*v2.y()) / (coeff1 + coeff2),

                    (coeff1*v1.z() + coeff2*v2.z()) / (coeff1 + coeff2)
);
                              // Update attempts
                              vUpdateAttempts[vOldIndices[index1]] = 0;

                              // Update indices list
                              vOldIndices.erase( vOldIndices.begin() +
index1 );
                              vNewIndices.erase( vNewIndices.begin() +
index2 );
                        } else {
                              bMatchesExist = false;
                        }

                  }

                  // Increment attempts for non-updated points
                  for(int i = vOldIndices.size()-1; i >= 0 ; i--) {
                        if( vUpdateAttempts[vOldIndices[i]] >
PERSIST_FRAMES ) {
```

```
                              vUpdateAttempts.erase(
vUpdateAttempts.begin() + vOldIndices[i] );
                              vPosPts.erase( vPosPts.begin() +
vOldIndices[i] );
                              vVelPts.erase( vVelPts.begin() +
vOldIndices[i] );

                              vTmpDelIndices.push_back( vOldIndices[i]
);
                        } else {
                              vUpdateAttempts[vOldIndices[i]]++;
                        }
                  }

                  // Add new points
                  for(int i = 0; i < vNewIndices.size(); i++) {
                        vPosPts.push_back( vTmpOutPts[vNewIndices[i]] );
                        vVelPts.push_back( osg::Vec3(0,0,0) );
                        vUpdateAttempts.push_back( 0 );
                  }

                  EnterCriticalSection( &critOutput );

                  vOutputPts = vPosPts;        //probably need to change
this to keep persistance
                  for(int i = 0; i < vTmpDelIndices.size(); i++)
                        vDeletedIndices.push_back( vTmpDelIndices[i] );

                  LeaveCriticalSection( &critOutput );

                  vTmpDelIndices.clear();
            }

      }
      delete startC;
      delete endC;

}

bool IntersectionTester::intersect(osg::Vec3 p1, osg::Vec3 p2,
osg::Vec3 p3, osg::Vec3 p4, osg::Vec3* pa, osg::Vec3* pb) {
   /*
   Calculate the line segment PaPb that is the shortest route between
   two lines P1P2 and P3P4. Calculate also the values of mua and mub
where
      Pa = P1 + mua (P2 - P1)
      Pb = P3 + mub (P4 - P3)
   Return FALSE if no solution exists.
*/

      // Found at
http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline3d/
```

```
   osg::Vec3 p13, p43, p21;
   double d1343, d4321, d1321, d4343, d2121;
   double mua, mub;
   double numer, denom;

   p13.x() = p1.x() - p3.x();
   p13.y() = p1.y() - p3.y();
   p13.z() = p1.z() - p3.z();
   p43.x() = p4.x() - p3.x();
   p43.y() = p4.y() - p3.y();
   p43.z() = p4.z() - p3.z();


   if (abs(p43.x())  < EPS && abs(p43.y())  < EPS && abs(p43.z())  <
EPS)
      return(false);

   p21.x() = p2.x() - p1.x();
   p21.y() = p2.y() - p1.y();
   p21.z() = p2.z() - p1.z();

   if (abs(p21.x())  < EPS && abs(p21.y())  < EPS && abs(p21.z())  <
EPS)
      return(false);

   d1343 = p13.x() * p43.x() + p13.y() * p43.y() + p13.z() * p43.z();
   d4321 = p43.x() * p21.x() + p43.y() * p21.y() + p43.z() * p21.z();
   d1321 = p13.x() * p21.x() + p13.y() * p21.y() + p13.z() * p21.z();
   d4343 = p43.x() * p43.x() + p43.y() * p43.y() + p43.z() * p43.z();
   d2121 = p21.x() * p21.x() + p21.y() * p21.y() + p21.z() * p21.z();

   denom = d2121 * d4343 - d4321 * d4321;

   if (abs(denom) < EPS)
      return(false);

   numer = d1343 * d4321 - d1321 * d4343;

   mua = numer / denom;
   mub = (d1343 + d4321 * mua) / d4343;

   // Need to test if pa and pb are allocated memory??

   pa->x() = p1.x() + mua * p21.x();
   pa->y() = p1.y() + mua * p21.y();
   pa->z() = p1.z() + mua * p21.z();

   pb->x() = p3.x() + mub * p43.x();
   pb->y() = p3.y() + mub * p43.y();
   pb->z() = p3.z() + mub * p43.z();
   return(true);
}
```

ManipulatorWidget.h

```
#include <windows.h>
#include <time.h>

#include <osg/Geometry>
#include <osg/Geode>
#include <osg/Shape>
#include <osg/ShapeDrawable>
#include <osg/MatrixTransform>
#include <osg/LineWidth>
#include <osg/BoundingBox>

#ifndef __MANIPULATORWIDGET


#define __MANIPULATORWIDGET

class ManipulatorWidget {

public:
      // Eponymous func's
      ManipulatorWidget(bool);
      ~ManipulatorWidget();

      // Update func's
      void updatePosition(osg::Vec3, osg::Vec3);
      void updateScale();
      void updateTR();
      void update();


      // Type
      bool bType;      //true for scale, false for trans/rot

      // Geometry access
      osg::ref_ptr<osg::Geode> geoSphereA;
      osg::ref_ptr<osg::ShapeDrawable> shapeA;
      osg::ref_ptr<osg::Geode> geoSphereB;
      osg::ref_ptr<osg::ShapeDrawable> shapeB;
      osg::ref_ptr<osg::Geode> geoLine;
      osg::ref_ptr<osg::MatrixTransform> matTransform;

      // Parent object
      osg::MatrixTransform * matParent;
```

```
        // Timer item
        time_t tUpdate;

        // Lock items
        bool bLock;
        osg::Vec3 vLockedA;
        osg::Vec3 vLockedB;

        // History items
        osg::Vec3 vOldA;
        osg::Vec3 vOldB;

};

#endif
```

ManipulatorWidget.cpp

```
#include "ManipulatorWidget.h"


ManipulatorWidget::ManipulatorWidget(bool bScale) {

        // Create widget endpoints - all units in mm
        osg::Vec3 ptA;
        osg::Vec3 ptB;
        osg::ref_ptr<osg::Sphere> unitSphereA, unitSphereB;
        //if( bScale ) {
                ptA = osg::Vec3(30,0,0);
                ptB = osg::Vec3(-30,0,0);
                unitSphereA = new osg::Sphere(ptA, 15);
                unitSphereB = new osg::Sphere(ptB, 15);
        //} else {
        //      ptA = osg::Vec3(0,0,30);
        //      ptB = osg::Vec3(0,0,-30);
        //      unitSphereA = new osg::Sphere(ptA, 10);
        //      unitSphereB = new osg::Sphere(ptB, 10);

        //}

        // Create sphere A
        shapeA = new osg::ShapeDrawable( unitSphereA.get() );
        geoSphereA = new osg::Geode();
        geoSphereA->addDrawable( shapeA.get() );

        // Create sphere B
        shapeB = new osg::ShapeDrawable( unitSphereB.get() );
        geoSphereB = new osg::Geode();
        geoSphereB->addDrawable( shapeB.get() );
```

```
      // Create interstitial line
      osg::ref_ptr<osg::Geometry> geomLine = new osg::Geometry();
      geoLine = new osg::Geode();

      osg::ref_ptr<osg::Vec3Array> vecCoords = new osg::Vec3Array( 2 );

      (*(vecCoords.get()))[0] = ptA;
      (*(vecCoords.get()))[1] = ptB;

      osg::ref_ptr<osg::Vec4Array> color = new osg::Vec4Array( 1 );
      (*(color.get()))[0] = osg::Vec4(1.f, 1.f, 0.f, 1.f);

      osg::ref_ptr<osg::StateSet> stateset = new osg::StateSet;
      osg::ref_ptr<osg::LineWidth> linewidth = new osg::LineWidth();
      linewidth->setWidth(5.0f);
      stateset->setAttributeAndModes( linewidth.get(),
osg::StateAttribute::ON );
      stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);

      geomLine->setColorArray( color.get() );
      geomLine->setColorBinding(osg::Geometry::BIND_OVERALL);
      geomLine->setVertexArray( vecCoords.get() );
      geomLine->addPrimitiveSet(new
osg::DrawArrays(osg::PrimitiveSet::LINES, 0, 2));
      geomLine->setStateSet(stateset.get());
      geomLine->setUseDisplayList(false);

      geoLine->addDrawable( geomLine.get() );

      // Create parent matrix
      matTransform = new osg::MatrixTransform();
      matTransform->addChild( geoSphereA.get() );
      matTransform->addChild( geoSphereB.get() );
      matTransform->addChild( geoLine.get() );
    matTransform->getOrCreateStateSet()->setRenderBinDetails(60,
"RenderBin");

      // Set update time
      time( &tUpdate );

      // Initially not locked
      bLock = false;

      // Set type
      bType = bScale;

}

ManipulatorWidget::~ManipulatorWidget() {
      matParent = NULL;
}
```

```
void ManipulatorWidget::updatePosition(osg::Vec3 pos, osg::Vec3 orient)
{
      osg::Matrixd tmp;
      tmp.setTrans( pos );
      //tmp.preMult( tmp.rotate( osg::Vec3(0,0,1), orient ) );
      matTransform->setMatrix( tmp );
}

void ManipulatorWidget::update() {
      if( bType ) {
            updateScale();
      } else {
            updateTR();
      }
}

void ManipulatorWidget::updateScale() {


      // Update the widget & parent object scale upon lock
      if( bLock ) {
            // Update locking
            osg::BoundingSphere ba = geoSphereA->getBound();
            ba.set(ba.center() * matTransform->getMatrix(),
ba.radius());
            osg::BoundingSphere bb = geoSphereA->getBound();
            bb.set(bb.center() * matTransform->getMatrix(),
bb.radius());

            if( ba.contains( vLockedA ) && bb.contains( vLockedB ) ) {
                  bLock = true;

            } else {
                  bLock = false;

            }

            // Get old scaling based on finger distance
            osg::Vec3 v = vOldA - vOldB;
            double dOldDist = v.length();

            // Get new scaling based on finger distance
            v = vLockedA - vLockedB;
            double dNewDist = v.length();

            // Update matrices with new scaling
            double dScale = dNewDist / dOldDist;
            osg::Matrixd tmp = matTransform->getMatrix();
            tmp.preMultScale( osg::Vec3( dScale, dScale, dScale ) );
            matTransform->setMatrix( tmp );

            osg::Matrix s;
            s.postMultScale(osg::Vec3( dScale, dScale, dScale ));
            matParent->preMult( s );

            // Update history
            vOldA = vLockedA;
            vOldB = vLockedB;

            // Update clock
            time( &tUpdate );

      }
}

void ManipulatorWidget::updateTR() {

      // Update the widget & parent object translation & rotation upon
lock
      if( bLock ) {

            // Get rotation
            osg::Vec3 a = vOldA - vOldB;
            a.normalize();

            osg::Vec3 b = vLockedA - vLockedB;
            b.normalize();


            osg::Matrixd matRotate = osg::Matrix::rotate( a, b );

            // Apply rotation - unwind translation then apply rotation?
            osg::Matrixd tmp = matTransform->getMatrix();
            osg::Vec3 transWidget = tmp.getTrans();

            tmp.postMultTranslate( -transWidget );
            tmp.postMult( matRotate );
            tmp.postMultTranslate( transWidget );
            matTransform->setMatrix( tmp );

            tmp = matParent->getMatrix();
            osg::Vec3 transModel = tmp.getTrans();

            tmp.postMultTranslate( -transWidget );  //used transWidget
to keep relative rotation between model and widget
            tmp.postMult( matRotate );
            tmp.postMultTranslate( transWidget );
            matParent->setMatrix( tmp );

            // Get translation
            a = vOldA - vOldB;
            a = a * 0.5;
            osg::Vec3 vOldPosition = vOldB + a;
            a = vLockedA - vLockedB;
            a = a * 0.5;
            osg::Vec3 vLockedPosition = vLockedB + a;
            osg::Vec3 vTrans = vLockedPosition - vOldPosition;

            osg::Matrixd matTranslate = osg::Matrixd::translate( vTrans
);

            // Apply translation
            tmp = matTransform->getMatrix();
            tmp.postMult( matTranslate );     // using postmult to
apply translation after rotations/scales/etc.
            matTransform->setMatrix( tmp );

            matParent->postMult( matTranslate );

            // Update history
            vOldA = vLockedA;
            vOldB = vLockedB;

            // Update locking
            osg::BoundingSphere ba = geoSphereA->getBound();
            ba.set(ba.center() * matTransform->getMatrix(),
ba.radius());
            osg::BoundingSphere bb = geoSphereB->getBound();
            bb.set(bb.center() * matTransform->getMatrix(),
bb.radius());

            bool testA = ba.contains( vLockedA );
            bool testB = bb.contains( vLockedB );

            if( testA && testB ) {
                  bLock = true;

            } else {
                  bLock = false;
            }

            // Update clock
            time( &tUpdate );

      }
}


//int main() {
//
//    osg::Vec3 a(1,0,0);
//    osg::Vec3 b(0,1,0);
//
//    osg::Vec3 c = a^b;
//
```

```
//    osg::Matrix t;
//    t.setTrans(5, 1, 0);
//    t.postMult( osg::Matrixd::rotate(osg::Vec3(1,0,0),
osg::Vec3(0,1,0)) );
//
//    int q = 4;
//
//}
```

PtFilter.h

```
#ifndef __OPENCV200

      #include <opencv/cv.h>
      #include <opencv/cxcore.h>
      #include <opencv/highgui.h>

      #define __OPENCV200

#endif

class PtFilter {

public:
      PtFilter(double fps) {
            // initialize kalman variables
            kalman = cvCreateKalman(6,3,0);         //dynamic: x, y,z,
dx, dy, dz; measure: x,y,z; no control;
            state = cvCreateMat(6, 3, CV_32FC1);
            measurement = cvCreateMat(3, 1, CV_32FC1);

            const float A[] = { 1,0,0,1000.0/fps,0,0,
                                    0,1,0,0,1000.0/fps,0,
                                    0,0,1,0,0,1000.0/fps,
                                    0,0,0,1,0,0,
                                    0,0,0,0,1,0,
                                    0,0,0,0,0,1 };              //
transition matrix

            memcpy( kalman->transition_matrix->data.fl, A, sizeof(A) );
      // set kalman's transition matrix to A

            // Set process variables...not sure what's optimal here. :|
            cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1)
);
            cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-
5) );
            cvSetIdentity( kalman->measurement_noise_cov,
cvRealScalar(0.5) );
            cvSetIdentity( kalman->error_cov_post, cvRealScalar(1) );
            // initialize state to random
          CvRNG rand = cvRNG(-1);
            cvRandArr( &rand, kalman->state_post, CV_RAND_NORMAL,
cvRealScalar(0), cvRealScalar(0.1) );

            count = 0;
            isAccurate = false;
      }

      ~PtFilter() {
            cvReleaseKalman( &kalman );
      }

      osg::Vec3f getPrediction() {
            // Return the predicted position of the tracked point

            const CvMat *prediction = cvKalmanPredict( kalman, 0 );
            pPredict.x() = prediction->data.fl[0];
            pPredict.y() = prediction->data.fl[1];
            pPredict.z() = prediction->data.fl[2];
            return pPredict;
      }

      void update( osg::Vec3f pIn ) {
            // Update the tracking with a measurement

            measurement->data.fl[0] = pIn.x();
            measurement->data.fl[1] = pIn.y();
            measurement->data.fl[2] = pIn.z();

            cvKalmanCorrect( kalman, measurement );
      }

      void update() {
            // Update the tracking based on internal state (no
measurement)

            state = kalman->state_post;
            cvMatMul( kalman->transition_matrix, state, state);

            CvMat *predict = cvCreateMat(3,1,CV_32FC1);
            predict->data.fl[0] = state->data.fl[0];
            predict->data.fl[1] = state->data.fl[1];
            predict->data.fl[2] = state->data.fl[2];

            cvKalmanCorrect( kalman, predict );

            cvReleaseMat( &predict );

            count++;
      }
      int getCount() {
            return count;
      }

      void resetCount() {
            count = 0;
      }

      bool isAccurate;
      osg::Vec3f pPredict;


private:
      CvKalman *kalman;                             // kalman filter
data structure
      CvMat *state;                                       // state
variable
      CvMat *measurement;                                 // measurement
variable
      int count;
};
```

Main.cpp

```
#define NUM_CAMS 3

#include "Cam.h"
#include "IntersectionTester.h"
#include "ManipulatorWidget.h"
#include <sstream>

#include <osgViewer/Viewer>
#include <osgViewer/ViewerEventHandlers>
#include <osgGA/TrackballManipulator>
#include <osg/Geometry>
#include <osg/Node>
#include <osg/LineWidth>
#include <osg/TextureRectangle>
#include <osg/TexMat>
#include <osg/Shape>
#include <osg/ShapeDrawable>
#include <osg/PositionAttitudeTransform>
#include <osg/MatrixTransform>

#include <osgDB/ReadFile>

// Function declarations
osg::ref_ptr<osg::Camera> getVideoBackground(int, osg::Group *,
osg::Image *);
```

```
osg::ref_ptr<osg::Geode> getForegroundRect(cv::Mat *, osg::Vec3,
osg::Vec3, osg::Vec3, osg::Vec3);

// Global Variables
Cam * arrCams[NUM_CAMS];
HANDLE hth[NUM_CAMS];
unsigned uiThreadID[NUM_CAMS];
int intCurrentCam;

// Event handlers
class KeyHandler : public osgGA::GUIEventHandler
{
public:

      KeyHandler() {}
    ~KeyHandler() {}

    bool handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& aa)
    {
        osgViewer::Viewer* viewer =
dynamic_cast<osgViewer::Viewer*>(&aa);
        if (!viewer) return false;

        switch(ea.getEventType())
        {
            case(osgGA::GUIEventAdapter::KEYUP):
            {
                if (ea.getKey()=='b')
                {
                            // suspend threads
                            for(int i = 0; i < NUM_CAMS; i++) {
                                  SuspendThread( hth[i] );
                            }

                            // reinit background
                            for(int i = 0; i < NUM_CAMS; i++) {
                                  arrCams[i]->resetBackground();
                            }

                            for(int i = 0; i < NUM_CAMS; i++) {
                                  ResumeThread( hth[i] );
                            }
                      }
                else if (ea.getKey()=='r')
                {
                            // suspend threads
                            for(int i = 0; i < NUM_CAMS; i++) {
                                  SuspendThread( hth[i] );
                            }

                            // reinit matrices
```

```
                              for(int i = 0; i < NUM_CAMS; i++) {
                                    arrCams[i]->resetCamMatrix();
                                    arrCams[i]->resetBackground();
                              }

                              for(int i = 0; i < NUM_CAMS; i++) {
                                    ResumeThread( hth[i] );
                              }
                        }
                        else if (ea.getKey() == ']')
                        {
                              // Reset view to the next camera
                              intCurrentCam = (intCurrentCam + 1) %
NUM_CAMS;
                              viewer->getCamera()->setProjectionMatrix(
arrCams[intCurrentCam]->getProjectionMatrix() );
                              viewer->getCamera()->setViewMatrix(
arrCams[intCurrentCam]->getViewMatrix() );
                        }
                        else if (ea.getKey() == '[')
                        {
                              // Reset view to previous camera
                              intCurrentCam = (NUM_CAMS + intCurrentCam
- 1) % NUM_CAMS;
                              viewer->getCamera()->setProjectionMatrix(
arrCams[intCurrentCam]->getProjectionMatrix() );
                              viewer->getCamera()->setViewMatrix(
arrCams[intCurrentCam]->getViewMatrix() );
                        }
                    return false;
                }
                case(osgGA::GUIEventAdapter::PUSH):
                case(osgGA::GUIEventAdapter::MOVE):
                case(osgGA::GUIEventAdapter::RELEASE):

                default:
                    return false;
            }

                viewer = NULL;
        }
};

int main() {

      // Set up OSG viewer
      osg::ref_ptr<osg::Group> root = new osg::Group();

      osgViewer::Viewer viewer;
     viewer.setThreadingModel(osgViewer::Viewer::SingleThreaded);
      viewer.setUpViewInWindow(500,100,800,600,0);
      viewer.setSceneData( root.get() );
```

```
    viewer.addEventHandler(new osgViewer::StatsHandler);
    viewer.addEventHandler(new osgViewer::WindowSizeHandler);
    viewer.addEventHandler(new osgViewer::ThreadingHandler);
    viewer.addEventHandler(new osgViewer::HelpHandler);

      // Tracking vectors
      std::vector<osg::MatrixTransform *> vecModels;
      std::vector<bool> vecHasWidget;
      std::vector<ManipulatorWidget *> vecWidgets;


      // Create object interface
      //osg::ref_ptr<osg::Node> bigboxGeode =
osgDB::readNodeFile("Models/827_3389_200.3ds");
      osg::ref_ptr<osg::Box> bigbox = new osg::Box(osg::Vec3(0,0,0),
35);
      osg::ref_ptr<osg::ShapeDrawable> bigboxDrawable = new
osg::ShapeDrawable( bigbox.get() );
      osg::ref_ptr<osg::Geode> bigboxGeode = new osg::Geode();
      bigboxGeode->addDrawable( bigboxDrawable.get() );
    bigboxGeode->getOrCreateStateSet()->setRenderBinDetails(50,
"RenderBin");

      osg::ref_ptr<osg::MatrixTransform> matModel1 = new
osg::MatrixTransform( osg::Matrix::translate(0,0,100) );
      matModel1->addChild( bigboxGeode.get() );
      root->addChild( matModel1.get() );

      // Add objects' transforms to tracking vector
      vecModels.push_back( matModel1.get() );
      vecHasWidget.push_back( false );

      // Create scale button
      osg::ref_ptr<osg::Box> boxButton = new
osg::Box(osg::Vec3(0,0,0),25,20,10);
      osg::ref_ptr<osg::ShapeDrawable> boxButtonDraw = new
osg::ShapeDrawable( boxButton.get() );
      osg::ref_ptr<osg::Geode> boxButtonGeo = new osg::Geode();
      boxButtonGeo->addDrawable( boxButtonDraw.get() );
      boxButtonGeo->getOrCreateStateSet()->setRenderBinDetails(50,
"RenderBin");

      osg::ref_ptr<osg::MatrixTransform> matButton1 = new
osg::MatrixTransform( osg::Matrix::translate(200,-50,5) );
      matButton1->addChild( boxButtonGeo.get() );
      root->addChild( matButton1.get() );

      // Set up intersection calculation thread
      IntersectionTester * IntTest = new IntersectionTester();
      HANDLE hthIntersect;
      unsigned uiThreadIDIntersect;
      hthIntersect = (HANDLE)_beginthreadex( NULL,         // security
                                                             0,
// stack size

IntersectionTester::threadedEntry,
                                                             IntTest,
// arg list

CREATE_SUSPENDED,

&uiThreadIDIntersect );

      // Set up loop
      for(int i = 0; i < NUM_CAMS; i++) {
            arrCams[i] = new Cam(i);

            hth[i] = (HANDLE)_beginthreadex( NULL,                   //
security
                                                 0,
            // stack size
                                                 Cam::threadedEntry,
                                                 arrCams[i],
            // arg list
                                                 CREATE_SUSPENDED,
                                                 &uiThreadID[i] );
            if ( hth[i] == 0 )
                  printf("Failed to create thread %d\n", i);

      }


      // set start cam position
      intCurrentCam = 0;
      viewer.getCamera()->setProjectionMatrix( arrCams[intCurrentCam]-
>getProjectionMatrix() );
      viewer.getCamera()->setViewMatrix( arrCams[intCurrentCam]-
>getViewMatrix() );


      // Set up video background
      osg::ref_ptr<osg::Image> imgBackground = new osg::Image();
      osg::ref_ptr<osg::Camera> bckgnd =
getVideoBackground(intCurrentCam, root.get(), imgBackground.get() );


      // Create finger sphere
      osg::ref_ptr<osg::Sphere> unitSphere = new
osg::Sphere(osg::Vec3(0,0,0), 5);
      osg::ref_ptr<osg::ShapeDrawable> unitSphereDrawableA = new
osg::ShapeDrawable( unitSphere.get() );
      unitSphereDrawableA->setColor( osg::Vec4(0.0,0.0,1.0,1.0));
      osg::ref_ptr<osg::Geode> geoSphereA = new osg::Geode();
      geoSphereA->addDrawable( unitSphereDrawableA.get() );
      geoSphereA->getOrCreateStateSet()->setRenderBinDetails(50, 
"RenderBin");

      // Vector to contain position transforms for the spheres
      std::vector< osg::ref_ptr<osg::PositionAttitudeTransform> > 
vSpheres;

      // Execute threads
      for(int i = 0; i < NUM_CAMS; i++) {
            ResumeThread( hth[i] );
      }
      ResumeThread( hthIntersect );


      std::vector< std::vector< osg::Vec3 > > vStart, vEnd;

      IntTest->setThreshold(4);
      std::vector< osg::Vec3 > vPts;
      std::vector< int > vDelPts, vAInd, vBInd;

      osg::ref_ptr<osg::Geode> arrForegrounds[] = {NULL, NULL, NULL, 
NULL};


      // Display loop
      viewer.addEventHandler( new KeyHandler() );
    viewer.realize();

      std::vector<cv::Mat> vOutROI;
      std::vector<cv::Rect> vOutRects;
      cv::Mat tmpMat;
      bool bScaleFlag = false, isIntersected = false;

      double dErrorTerm = 5;

      while( !viewer.done() ) {

            // update video background
            cv::Mat tmpImg = cv::Mat(arrCams[intCurrentCam]-
>imgOutput);
            cv::Mat tmpMask = arrCams[intCurrentCam]->imgForeMask;

            imgBackground->setImage(arrCams[intCurrentCam]->imgOutput-
>width,
                                                arrCams[intCurrentCam]-
>imgOutput->height,
                                                arrCams[intCurrentCam]-
>imgOutput->depth,
                                                3,
                                                GL_BGR,
                                                GL_UNSIGNED_BYTE,
                                                (unsigned 
char*)arrCams[intCurrentCam]->imgOutput->imageData,
                                                osg::Image::NO_DELETE,
                                                1 );

            // Get intersection output
            IntTest->getOutputs(&vPts, &vDelPts);

            // Get foreground rects
            vOutROI.clear();
            vOutRects.clear();
            arrCams[intCurrentCam]->getCrit();
            //vOutROI = arrCams[intCurrentCam]->vOutROI;
            vOutRects = arrCams[intCurrentCam]->vOutRects;
            arrCams[intCurrentCam]->releaseCrit();

            // Get 2D transforms of points
            std::vector< osg::Vec3 > vPts2D;
            //std::vector< osg::Vec3 > vPtsPartialTransform;
            for(int i = 0; i < vPts.size(); i++) {
                  osg::Vec3 pt2D;
                  osg::Matrix mat = arrCams[intCurrentCam]->matView 
*arrCams[intCurrentCam]->matProjection * arrCams[intCurrentCam]-
>matViewport ;
                  pt2D = vPts[i] * mat;
                  //vPts2D.push_back( osg::Vec2( pt2D.x(), pt2D.y()));
                  vPts2D.push_back( pt2D );

                  //pt2D = vPts[i] * arrCams[intCurrentCam]->matView;
                  //vPtsPartialTransform.push_back( pt2D );
            }

            // Process foreground rectangles
            std::vector<float> fZvalues;
            for(int i = 0; i < vOutRects.size(); i++) {

                  float zCount = 0;
                  fZvalues.push_back( 0 );
                  cv::Rect tmpRect = vOutRects[i];

                  // Get average z value of rectangle
                  for(int j = 0; j < vPts2D.size(); j++) {
                        // see if point lies within rectangle on 2D 
plane
                        if( vPts2D[j].x() >= tmpRect.x - dErrorTerm && 
vPts2D[j].x() <= (tmpRect.x + tmpRect.width+dErrorTerm) &&
                              vPts2D[j].y() >= tmpRect.y - dErrorTerm && 
vPts2D[j].y() <= (tmpRect.y + tmpRect.height + dErrorTerm) ){

                                    //Average in points z value
                                    fZvalues[i] = (fZvalues[i] * zCount + 
vPts2D[j].z())  / (zCount+1.0);
```

```
                              zCount++;

                        }
                  }

                  // Create ROI images
                  cv::Mat tmp;
                  cv::getRectSubPix(tmpImg, cv::Size(tmpRect.width*2, 
tmpRect.height*2), cv::Point2f(tmpRect.x*2 + tmpRect.width, tmpRect.y*2 
+ tmpRect.height),tmp,-1);

                  cv::Mat roi = cv::Mat(tmp.rows, tmp.cols, CV_8UC4, 
cv::Scalar(0,0,0,0));
                  // changing channels,so can't use copyto op
                  for(int j = 0; j < tmp.rows; j++) {
                        for(int k = 0; k < tmp.cols; k++) {
                              if( tmpMask.at<uchar>(tmpRect.y+(j/2), 
tmpRect.x+(k/2)) > 0 ) {
                                    cv::Vec3b tmpPt = 
tmp.at<cv::Vec3b>(j,k);
                                    cv::Vec4b tmpPt2 = 
cv::Vec4b(tmpPt[0], tmpPt[1], tmpPt[2], 255);
                                    roi.at<cv::Vec4b>(j,k) = tmpPt2;
                              }
                        }
                  }

                  vOutROI.push_back( roi );
            }

            // Draw ROI rectangles
            for(int i = 0; i < 4; i++) {
                  if( i < fZvalues.size() ) {
                        if( fZvalues[i] != 0) {
                              tmpMat = vOutROI[i];
                              cv::Rect tmpRect = vOutRects[i];

                              //Update size/position
                              osg::Matrix mat;
                              mat.invert( arrCams[intCurrentCam]-
>matView * arrCams[intCurrentCam]->matProjection * 
arrCams[intCurrentCam]->matViewport);

                              // Get rect corners
                              osg::Vec3 UL = osg::Vec3(tmpRect.x, 
(tmpImg.rows/2)-(tmpRect.y), fZvalues[i]) * mat;
                              osg::Vec3 UR = 
osg::Vec3(tmpRect.x+tmpRect.width, (tmpImg.rows/2)-(tmpRect.y), 
fZvalues[i]) * mat;
                              osg::Vec3 LR = 
osg::Vec3(tmpRect.x+tmpRect.width, (tmpImg.rows/2)-
(tmpRect.y+tmpRect.height), fZvalues[i]) * mat;
```

```
                              osg::Vec3 LL = osg::Vec3(tmpRect.x,
(tmpImg.rows/2)-(tmpRect.y+tmpRect.height), fZvalues[i]) * mat;


                              // Create rectangle
                              if( arrForegrounds[i] != NULL ) {
                                    root->removeChild(
arrForegrounds[i].get() );
                                    arrForegrounds[i] = NULL;
                              }
                              arrForegrounds[i] = getForegroundRect(
&tmpMat, UL, UR, LR, LL );

                              root->addChild( arrForegrounds[i].get() );
                        }
                  } else {
                        if( arrForegrounds[i] != NULL ) {
                              root->removeChild( arrForegrounds[i].get()
);
                              arrForegrounds[i] = NULL;
                        }
                  }
            }


            // Draw spheres
            for(unsigned int i = 0; i < vPts.size(); i++) {
                  if( vSpheres.size() <= i ) {
                        osg::ref_ptr<osg::PositionAttitudeTransform>
tmp;
                        tmp = new osg::PositionAttitudeTransform();
                        tmp->setPosition( vPts[i] );
                        tmp->addChild( geoSphereA.get() );
                        tmp->setDataVariance(osg::Object::DYNAMIC);

                        vSpheres.push_back( tmp );
                        root->addChild( vSpheres[i].get() );
                  } else {
                        vSpheres[i]->setPosition( vPts[i] );
                  }
            }

            //Remove remaining spheres
            for(int i = vSpheres.size()-1; i > vPts.size()-1; i--) {
                  root->removeChild( vSpheres[i].get() );
                  vSpheres.erase( vSpheres.begin() + i );
            }

            // update intersection test
            vStart.clear();
            vEnd.clear();
            for( int i = 0; i < NUM_CAMS; i++ ) {
                  arrCams[i]->getCrit();
                  if( arrCams[i]->vNear.size() > 0 ) {
                        std::vector< osg::Vec3 > tmp = arrCams[i]-
>vNear;
                        vStart.push_back( tmp );

                        std::vector< osg::Vec3 > tmp2 = arrCams[i]-
>vFar;
                        vEnd.push_back( tmp2 );
                  } else {
                        std::vector< osg::Vec3 > *tmp1 = new
std::vector<osg::Vec3>;
                        std::vector< osg::Vec3 > *tmp2 = new
std::vector<osg::Vec3>;
                        vStart.push_back( *tmp1 );
                        vEnd.push_back( *tmp2 );
                  }
                  arrCams[i]->releaseCrit();
            }
            IntTest->setInputPts( vStart, vEnd );

            // Update widget index lists
            for(int i = 0; i < vDelPts.size(); i++) {
                  for(int j = 0; j < vAInd.size(); j++) {
                        if( vAInd[j] > vDelPts[i]) {
                              vAInd[j] = vAInd[j] - 1;
                        } else if( vAInd[j] == vDelPts[i]) {
                              vAInd[j] = -1;
                        }

                        if( vBInd[j] > vDelPts[i]) {
                              vBInd[j] = vBInd[j] - 1;
                        } else if( vBInd[j] == vDelPts[i]) {
                              vBInd[j] = -1;
                        }
                  }
            }


            // Update & Check widgets for expiration
            for(int i = 0; i < vecWidgets.size(); i++ ) {

                  // Update widget lock values ->if gone, remove lock
condition

                  if( vecWidgets[i]->bLock == true) {
                        if( vAInd[i] < 0 || vBInd[i] < 0 ) {
                              vecWidgets[i]->bLock = false;
                        } else {
                              vecWidgets[i]->vLockedA = vPts[vAInd[i]];
                              vecWidgets[i]->vLockedB = vPts[vBInd[i]];
                        }
                  }

                  vecWidgets[i]->update();

                  time_t now;
                  time( &now );
                  double t = difftime(now, vecWidgets[i]->tUpdate );

                  if( t > 5 ) {    // expiration time in s
                        // clear parent's widget
                        for(int j = 0; j < vecModels.size(); j++ ) {
                              if( vecModels[j] == vecWidgets[i]-
>matParent ) {
                                  vecHasWidget[j] = false;
                              }
                        }

                        // suicide
                        root->removeChild( vecWidgets[i]-
>matTransform.get() );
                        delete vecWidgets[i];
                        vecWidgets.erase( vecWidgets.begin() + i);

                        vAInd.erase( vAInd.begin() + i);
                        vBInd.erase( vBInd.begin() + i);
                  }
            }



            // Check for model intersections
            for(int k = 0; k < vecModels.size(); k++) {
                  for(int j = 0; j < vPts.size(); j++) {

                        if( vecModels[k]->getBound().contains( vPts[j] )
&& !(vecHasWidget[k]) ) {
                              // intersected, create a widget -
currently trans/rot
                              ManipulatorWidget * mwNew1 = new
ManipulatorWidget(bScaleFlag);
                              mwNew1->updatePosition(vPts[j],
osg::Vec3());

                              mwNew1->matParent = vecModels[k];
                              root->addChild( mwNew1->matTransform.get()
);

                              vecWidgets.push_back(mwNew1);

                              vecHasWidget[k] = true;

                              // push back indices as well
                              vAInd.push_back( -1 );
                              vBInd.push_back( -1 );
```

```
				}
			}
		}


		// Check widgets for entering lock condition
		for(int i = 0; i < vecWidgets.size(); i++ ) {
			if( vecWidgets[i]->bLock == false ) {
				bool bA = false;
				bool bB = false;

				// Check for intersections with ptA
				osg::BoundingSphere boundSphere = vecWidgets[i]-
>geoSphereA->getBound();
				boundSphere.set(boundSphere.center() *
vecWidgets[i]->matTransform->getMatrix(), boundSphere.radius());

				for(int j = 0; j < vPts.size(); j++) {
					if( boundSphere.contains( vPts[j] ) ) {
						bA = true;
						//vecWidgets[i]->vLockedA = &vPts[j];
						vAInd[i] = j;
						vecWidgets[i]->vOldA =
boundSphere.center();
						break;
					}
				}

				boundSphere = vecWidgets[i]->geoSphereB-
>getBound();
				boundSphere.set(boundSphere.center() *
vecWidgets[i]->matTransform->getMatrix(), boundSphere.radius());

				for(int j = 0; j < vPts.size(); j++) {
					if( boundSphere.contains( vPts[j] ) ) {
						bB = true;
						//vecWidgets[i]->vLockedB = &vPts[j];

						vBInd[i] = j;
						vecWidgets[i]->vOldB =
boundSphere.center();
						break;
					}
				}

				//Set color to know if you're intersecting
				if( bA ) {
					osg::Vec4 color = osg::Vec4(1.f, 0.f, 0.f,
1.f);
					vecWidgets[i]->shapeA->setColor( color );
				} else {
```

```
                              osg::Vec4 color = osg::Vec4(1.f, 1.f, 1.f,
1.f);
                              vecWidgets[i]->shapeA->setColor( color );
                        }

                        // Set color to know if you're intersecting
                        if( bB ) {
                              osg::Vec4 color = osg::Vec4(1.f, 0.f, 0.f,
1.f);
                              vecWidgets[i]->shapeB->setColor( color );
                        } else {
                              osg::Vec4 color = osg::Vec4(1.f, 1.f, 1.f,
1.f);
                              vecWidgets[i]->shapeB->setColor( color );
                        }

                        // Set lock condition
                        if( bA && bB ) {
                              vecWidgets[i]->bLock = true;
                        }
                  }
            }

            // Check for button intersections - need to modify so that
the intersection doesn't flip constantly
            bool isIntersectedFrame = false;

            osg::BoundingBox boundBox = boxButtonGeo->getBoundingBox();
            boundBox.set(osg::Vec3(boundBox.xMin(), boundBox.yMin(),
boundBox.zMin()) * matButton1->getMatrix(),
                  osg::Vec3(boundBox.xMax(), boundBox.yMax(),
boundBox.zMax()) * matButton1->getMatrix());

            for(int j = 0; j < vPts.size(); j++) {

                  if( boundBox.contains( vPts[j] )) {
                        isIntersectedFrame = true;
                  }
            }

            if( isIntersectedFrame == true) {
                  if( isIntersectedFrame != isIntersected ) {

                              bScaleFlag = !bScaleFlag;
                              isIntersected = true;

                              //change color
                              if( bScaleFlag )
                                    boxButtonDraw->setColor(
osg::Vec4(1.f, 0.f, 0.f, 1.f) );
                              else
```

```
                                        boxButtonDraw-
>setColor(osg::Vec4(1.f, 1.f, 1.f, 1.f) );
                        }
                } else {
                        isIntersected = false;
                }

                // Render
                viewer.frame();
        }


        // Wait for threads to resolve to close out.
        for(int i = 0; i < NUM_CAMS; i++) {
                arrCams[i]->setKey(27);
                WaitForSingleObject( hth[i], INFINITE );
        }

        for(int i = 0; i < NUM_CAMS; i++) {
                CloseHandle( hth[i] );
                delete arrCams[i];
                arrCams[i] = NULL;
        }

        // Close out intersector
        IntTest->setKey( 27 );
        WaitForSingleObject( hthIntersect, INFINITE );
        CloseHandle( hthIntersect );
        delete IntTest;
        IntTest = NULL;

 }


 osg::ref_ptr<osg::Camera> getVideoBackground(int intCam, osg::Group *
root, osg::Image * imgBackground) {

        Cam * cam = arrCams[intCam];

        int w = cam->imgOutput->width;
        int h = cam->imgOutput->height;
        int d = cam->imgOutput->depth;

        imgBackground->setImage(w,
                                                h,
                                                d,
                                                3,
                                                GL_BGR,
                                                GL_UNSIGNED_BYTE,
                                                (unsigned char*)cam->imgOutput-
>imageData,
                                                osg::Image::NO_DELETE,
                                        1 );

	// set up camera params for the vid background
	osg::ref_ptr<osg::Camera> videoBackground = new osg::Camera();
	videoBackground->setViewMatrix(osg::Matrix::identity());
	videoBackground->setRenderOrder(osg::Camera::NESTED_RENDER);
	videoBackground->setClearMask(GL_DEPTH_BUFFER_BIT);
	videoBackground->getOrCreateStateSet()->setMode(GL_LIGHTING, 
GL_FALSE);
	videoBackground->getOrCreateStateSet()->setMode(GL_DEPTH_TEST, 
GL_FALSE);
	videoBackground->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
	videoBackground->setProjectionMatrixAsOrtho2D(0.0f, (float)w, 
0.0f, (float)h);

	// create texture
	osg::ref_ptr<osg::TextureRectangle> txtImage = new 
osg::TextureRectangle( imgBackground );
	osg::ref_ptr<osg::TexMat> texmat = new osg::TexMat;
    texmat->setScaleByTextureRectangleSize(true);


	// create some geometry
	osg::ref_ptr<osg::Geode> vidGeode = new osg::Geode();
	osg::ref_ptr<osg::Geometry> geometry = new osg::Geometry();

	osg::ref_ptr<osg::Vec3Array> vcoords = new osg::Vec3Array();
	geometry->setVertexArray(vcoords.get());

	osg::ref_ptr<osg::Vec2Array> tcoords = new osg::Vec2Array();
	geometry->setTexCoordArray(0, tcoords.get());

	vcoords->push_back(osg::Vec3(0.0f, 0.0f, 0.0f));
	vcoords->push_back(osg::Vec3((float)w, 0.0f, 0.0f));
	vcoords->push_back(osg::Vec3((float)w, (float)h, 0.0f));
	vcoords->push_back(osg::Vec3(0.0f,  (float)h, 0.0f));

	tcoords->push_back(osg::Vec2(0.0f, 1.0f));
	tcoords->push_back(osg::Vec2(1.0f, 1.0f));
	tcoords->push_back(osg::Vec2(1.0f, 0.0f));
	tcoords->push_back(osg::Vec2(0.0f, 0.0f));

	geometry->addPrimitiveSet(new 
osg::DrawArrays(osg::PrimitiveSet::QUADS, 0, 4));

	vidGeode->addDrawable(geometry.get());

	videoBackground->addChild( vidGeode.get() );

	// set dynamic data variance
	videoBackground->setDataVariance(osg::Object::DYNAMIC);
	txtImage->setDataVariance(osg::Object::DYNAMIC);
```

```
      videoBackground->getOrCreateStateSet()->setMode(GL_LIGHTING,
                       osg::StateAttribute::OFF |
osg::StateAttribute::PROTECTED);

      videoBackground->getOrCreateStateSet()-
>setTextureAttributeAndModes(0, txtImage.get(),
osg::StateAttribute::ON);
    videoBackground->getOrCreateStateSet()-
>setTextureAttributeAndModes(0, texmat.get(), osg::StateAttribute::ON);
    videoBackground->getOrCreateStateSet()->setRenderBinDetails(25,
"RenderBin");

      root->addChild( videoBackground.get());

      return videoBackground;
 }



 osg::ref_ptr<osg::Geode> getForegroundRect(cv::Mat * roi, osg::Vec3
UL, osg::Vec3 UR, osg::Vec3 LR, osg::Vec3 LL) {

      int w = roi->cols;
      int h = roi->rows;
      int d = 8;       //roi->depth returns 0...
      int c = roi->channels();

      osg::ref_ptr<osg::Image> img = new osg::Image();
      img->setImage(w,h,d,c,
                                  GL_BGRA,
                                  GL_UNSIGNED_BYTE,
                                  (unsigned char*)roi->data,
                                  osg::Image::NO_DELETE,
                                  1 );

      // create texture
      osg::ref_ptr<osg::TextureRectangle> txtImage = new
osg::TextureRectangle( img );
      osg::ref_ptr<osg::TexMat> texmat = new osg::TexMat;
    texmat->setScaleByTextureRectangleSize(true);


      // create some geometry
      osg::ref_ptr<osg::Geode> vidGeode = new osg::Geode();
      osg::ref_ptr<osg::Geometry> geometry = new osg::Geometry();

      osg::ref_ptr<osg::Vec3Array> vcoords = new osg::Vec3Array();
      geometry->setVertexArray(vcoords.get());

      osg::ref_ptr<osg::Vec2Array> tcoords = new osg::Vec2Array();
      geometry->setTexCoordArray(0, tcoords.get());
      vcoords->push_back(LL);
      vcoords->push_back(LR);
      vcoords->push_back(UR);
      vcoords->push_back(UL);

      tcoords->push_back(osg::Vec2(0.0f, 1.0f));
      tcoords->push_back(osg::Vec2(1.0f, 1.0f));
      tcoords->push_back(osg::Vec2(1.0f, 0.0f));
      tcoords->push_back(osg::Vec2(0.0f, 0.0f));

      geometry->addPrimitiveSet(new
osg::DrawArrays(osg::PrimitiveSet::QUADS, 0, 4));

      vidGeode->addDrawable(geometry.get());

      osg::StateSet * ss = vidGeode->getOrCreateStateSet();
      ss->setTextureAttributeAndModes(0, txtImage.get(),
osg::StateAttribute::ON);
    ss->setTextureAttributeAndModes(0, texmat.get(),
osg::StateAttribute::ON);
    ss->setRenderBinDetails(50, "RenderBin");
      ss->setMode(GL_LIGHTING, osg::StateAttribute::OFF |
osg::StateAttribute::PROTECTED);
      ss->setMode( GL_BLEND, osg::StateAttribute::ON );
      ss->setRenderingHint( osg::StateSet::TRANSPARENT_BIN );
      ss->setMode( GL_DEPTH_TEST, osg::StateAttribute::ON );

      txtImage->setDataVariance(osg::Object::DYNAMIC);

      return vidGeode;

 }
```