Dreaded Dragons: Wearable Augmented Reality Entertainment

Andrew Dalia, Brian Dellon, Natanel Fuchs

May 10, 2011

Abstract

The goal of Dreaded Dragons: Wearable Augmented Reality Entertainment (DD:WARE) is to build and execute an easy to use augmented reality (AR) system that allows a user to interact with virtual objects. The emphasis of the project is placed on creating a system that would be simple to implement and cost-effective. The system is designed to be immersive. It should create virtual objects that interact with elements of the real environment. An interactive game is used to display the system capabilities.

User immersion and user interaction with virtual objects is accomplished using video eyewear and marker detection. The eyewear overlays virtual objects defined in the DD:WARE system onto video input from a camera, which is attached to the eyewear. Interaction is accomplished using the position of user controlled markers, which are detected in the DD:WARE system using camera input. Virtual objects are also allowed to interact with each other, using physics simulation, and fall due to gravity or collide with each other. This increases user immersion.

Marker detection is accomplished using an AR library for the C programming language called ARToolkit. Marker detection allows abstract representations of virtual objects to be overlayed on the image of the real world based on detected marker position and orientation while user controlled markers allow for interaction. Physics simulation is accomplished using the Havok physics engine, a physics simulation library for the C++ programming language. Object movement and interaction were ascertained using abstract representations of objects. The position and orientation of these objects were manipulated by the physics engine and overlayed images of the objects were changed accordingly. Drawing of the camera input and system-defined objects onto the video eyewear was accomplished using GLUT, which is the OpenGL utility toolkit, a graphics library for the C programming language.

Contents

List of Figures

1 Introduction

We intend to create a wearable AR application utilizing the open source C++ ARToolkit framework [2] and the Havok physics engine [13]. We will use these tools to develop a virtual interactive environment for users to explore that will be broadcast directly to the eyes with video see-through using the Vuzix VR920, an image of which is shown in Figure 1 [20]. The environment will include the ability to create and interact with virtual objects that react to user input in a realistic pseudo-physical manner. The applications of this project lie in the realm of entertainment, such as video games, or possibly even interactive cinematic experiences. Our project will combine various available tools to create a unique environment for users.

Figure 1: Vuzix VR920 goggles

Integrating the virtual realm into reality has been a subject of interest in the technology and entertainment industries for decades. The rise of the science fiction genre and films that delve into fantasy realms, such as Star Trek, explored the fascination with virtual realities. Modern implementations of augmented reality in video gaming such as the Playstation Move and the Xbox Kinect, by adding virtual elements to reality, have proven that this fascination is not just a dream. From the minds of movie directors, artists, scientists, and the general public, virtual reality became a hot concept and soon much more than that. The idea of AR was directly motivated by virtual reality, which has been experimented with since the 1970's. If entire virtual systems could be created, immersing a user into a completely non-

physical environment, why couldn't a hybrid "mixed reality" system be produced, one that projected virtual images and objects through some kind of "portal" into the real world? Fiduciary markers, simple objects used in imaging systems, provide exactly the kind of "portal" needed in AR, since they act as reference points in the real world where virtual objects will be displayed.

AR technology has yet to see widespread implementation, and has much room for growth. It is not only a novel concept generally, but also in the context of virtual reality implementations, research, and development in academic settings. Much of the motivation of an academic project of this nature is in demonstrating the application of AR with modest resources in terms of both time and funds. A key challenge of AR lies in the lack of substantial available details and literature on the topic, but we believe its potential uses are reason enough to explore what can be accomplished and what we can contribute given the constraints that academia affords. To understand a little more about the development of AR, it is important to consider the question, "What kinds of things does mixed reality allow us to do when compared to a pure virtual reality?" While it is obvious that combining the real with the virtual affords more flexibility and more interesting behavior compared to purely reality based settings, what is the purpose of AR and why is it an important concept? One of the major advantages of AR is the wide range of interactivity that can be implemented. Through AR, humans can physically interact with or create interfaces to build interactions between real objects (e.g., a user's hand) and virtual objects (e.g., a virtual block image displayed as an overlay to a physical environment). With pure virtual reality, the system can be expected to behave relatively flawlessly on its own, since the environment is internally defined. However, AR is much more complex, and subject to different kinds of inputs, since parts of the system are real and react to virtual input in varied and unexpected manners. With virtual reality, the user interacts with nothing real and is immersed into an environment that is virtual and conceptual. This limits its usefulness for practical objectives since the user's impact on the system is restricted. In comparison, in AR, the user is still immersed

in an environment that is predominantly real in most cases. The introduction of virtual objects gives the user the flexibility to perform actions that would be impossible in the real world. This has applications beyond that of entertainment, such as implementing images to enhance or provide details as overlays on physical objects. Imagine a head-tracking device that uses AR to identify and overlay the name and information of a person. While such ideas are mostly displayed in fiction such as the Japanese animated series, "Death Note", research into related topics such as face-tracking have been done, such as the work of Chris Mitchell [15]. Such a device would be useful in different situations such as for use by spies or detectives.

The exact breadth of the specifics of AR is still unfolding as AR develops into a more mature and understood group of technologies. Even the term AR itself is ambiguous and has several different interpretations. However, today there are two widely accepted definitions of AR that most people with some knowledge or experience on the matter generally subscribe to: Azuma's definition of AR and Milgram's Reality-Virtual Continuum. Azuma's definition says AR must incorporate the following three components: the combination of real and virtual into a continuous environment, a system that is interactive in real-time, and a three dimensional (3D) interface [14]. Milgram's definition places AR into a continuum with the extreme ends representing a fully real environment and a fully virtual environment [9]. In the continuum, AR is positioned closer to the real environment extreme, since it involves a stronger focus on the physical environment than the projected virtual components. In fact, he introduces another term to explain the more virtual components of this continuum, known as "augmented virtuality." However, in the end, the final output is always the ultimate goal and mark of progress in any AR system. This final output is what the user actually experiences. AR displays can manifest themselves in three primary ways: head mounted displays (HMD), hand-held displays, and spatial displays [11]. HMDs use video see-through to provide an immersive experience for the user, since the mapping of virtual images onto the physical landscape and the real landscape are processed externally and then sent to the

display, which is all that the user sees. This is distinguished from optical see-through, where the display is partially transparent, and virtual images are generated on top of the naturally viewed landscape. Hand-held devices also use video see-through, but the image is constrained to the device's small display. These have been the most successful commercially, as they have been used in handheld devices such as mobile phones. Smartphone applications, such as on the Android and iPhone platforms, have the advantage of being portable because they run on small, easily transported computing devices. Some examples of such applications include Layar, which uses GPS, and stored points of interest to superimpose information about the world as you view it on an iPhone [7], and fairy trails, which superimposes fairies you can catch on the world [4]. The most novel direction that AR has taken has involved the use of spatial displays. With this kind of AR, the users and display are separated functionally and physically as part of the system, which makes it great for AR applications that involve collaboration in augmented space. Spatial display utilizes digital projectors for displaying virtual graphics on physical objects. Due to spatial display's reliance on projectors it can be used by multiple individuals without the use of HMDs and does not have the kind of display resolution problems that HMDs and hand-held devices generally have. An example of such an application of AR can be seen in [18].

2 Background

AR systems superimpose computer generated images onto the real world, generally with some relation to what is being observed. AR has long been conceived and has even been implemented within the last two decades, but it has seen very little commercial penetration. The major practical developments have been in military HUDs [19]. For nonmilitary enterprises, the cost can be prohibitive. AR can require highly complex image detection and recognition algorithms and the wearable video or transparent displays required for proper immersion tend to be unreasonably expensive. The cost of high-end display systems can be mitigated by using lower end wearable video displays and solving resultant image recognition issues with robust coding. The complexity of image detection can be marginalized with freely available software libraries, particularly software that uses markers. While we will not eschew all knowledge of correlation algorithms like RANSAC as seen in [16] and relatively simple error correction methods, we believe that creative use of the more limited marker recognition will be equally effective with a lower development cost.

We began by doing a cursory overview of various available AR frameworks, including: "The Way In," an AR SDK developed by Vuzix, which we rejected due to the dependency on the Vuzix iWear VR920 Consumer Video Eyewear (VR920) and lack of open source [20], the Armedia Plug-in for Google Sketch-Up [1], which we rejected because it is neither free nor open source, and FlArToolkit [5], which we rejected due to the unwieldy nature of Adobe Flash, which it relies on. Ultimately we chose ARToolkit because it is open source, free, has extensive documentation, and appears relatively simple to implement. [2] ARToolkit uses any number of physical markers made up of black square borders with user defined identifying symbols in the center to orient the position of virtual objects and interpret the physical space. This simplifies image recognition significantly at the cost of requiring the markers to be present. An example of a marker we have created for use in testing can be seen in Figure 2. ARToolkit uses the OpenGL Utility Toolkit (GLUT), an open source C graphics library, to draw virtual objects.

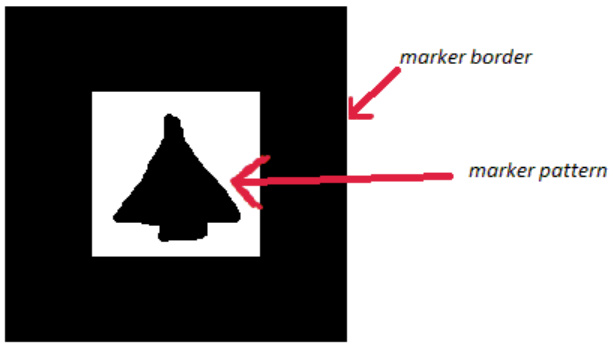


Figure 2: ARToolkit marker

To simulate a real-world physics environment we chose the Havok Physics Engine [13]. This was due to its extensive use in modern gaming, proving its versatility and reliability. In addition, the noncommercial version is both free and open-source. However, the physical environment and virtual objects that reside within it are represented differently across these platforms, and research must be done to find ways of reconciling this issue. One possible workaround is described in [10] and uses camera position as a reference frame which provides common information to both the ARToolkit-based and Havok-based software. In addition, although preliminary research into implementing both Havok and ARToolkit has been done, we must expand our understanding of these resources through extensive examination of both source code and documentation.

For display of the virtual world, initial experiments used a laptop or computer screen, but eventually we decided to use the VR920 goggles mentioned previously. Other goggles made by Vuzix were considered, including the Wrap 920AR, though that model was eliminated from consideration because it lacks a VGA connection without additional cost. We chose the VR920 due to the comparatively low cost (less than half the average for video goggles we found), and the previous testing and satisfactory performance evidenced by its use in [15]. In our initial experiments with ARToolkit, webcams built into laptops we owned and webcams available at The Cooper Union were used with their associated software. We now use the

Logitech Webcam Pro 9000, an image of which can be seen in Figure 3 due to its high resolution and auto-focus features [8]. This camera was also previously tested and found to work well in [15], which has similar video processing needs. Initially, the Vuzix iWear CamAR was considered because it is designed to attach to the VR920 goggles, but we decided that the Logitech Webcam Pro 9000 was a better choice because it offers a higher quality camera with a potentially advantageous auto-focus feature for a lower price. There was some discussion pertaining to the use of a smaller portable computing device to implement the project, but due to concerns about processing power and data transfer between peripherals and the computing platform, this idea was not feasible to implement and personal computers were used instead. Furthermore, the use of this device was not essential in demonstrating the use of AR technology and thus was not a priority.

Figure 3: Logitech Webcam Pro 9000

Augmented reality is a type of artificial reality that is still a novel concept in most areas. Along with its cousin, virtual reality, AR has yet to mature in use and has not yet become a common technology that pervades our lives. However, with modern powerful processors and new image processing techniques, the integration of virtual objects into real environments

in real-time is becoming more reliable. Virtual images themselves were generally unheard of outside science fiction until the 1970's when Myron Krueger completed his artificial reality laboratory called Videoplace [16]. In his lab, this artificial reality was not implemented through the use of goggles or headsets as we generally envision it today. Instead, it utilized devices like projectors, video cameras, and customized hardware to create an interactive environment for the user. The user's movements would be captured and projected as a silhouette representation in an artificial environment.

"Augmented Reality" was coined as a term in 1992, which was the year the first major paper on an AR prototype was published [2]. The first AR specific software development kit was developed in 1999 by Hirokazu Kato at the Human Interface Technology Laboratory (HITLab) at the University of Washington [19]. Since then, AR has been used in a variety of applications including: ARQuake, the first outdoor mobile AR game, and various Android and iPhone platform based applications.

ARToolkit is still prolifically utilized for builiding AR applications and the software library is occasionally updated. ARToolkit is called "the most widely used AR library in the world" [3]. Since its initial development in 1999, ARToolkit has been constantly updated with new extensions, functionalities, and improvements at HITLab and by open source developers who have contributed to building the framework over time. Some of the early accomplishments of ARToolkit involved creating a working environment that was compatible with both Windows and Linux operating systems. By ARToolkit 2.33, camera calibration and camera distortion functionality was significantly enhanced. The next major release was ARToolkit 2.61, which had enhanced tracking accuracy, some application of the Virtual Reality Modeling Language (VRML) and MacOS support. Next, ARToolkit 2.65 featured full VRML support as well as a new video library called DSVideoLib 0.04. Furthermore, it was the first to come distributed with examples, which made it far easier for users to begin building applications and understanding the different features of the development kit. Finally, ARToolkit 2.68 (the current version) introduced a new rendering library called gsub_lite [2].

ARToolkit has become the software library of choice for most AR developers due to its adaptive nature which is exemplified by the previously discussed updates. In turn, many developers have contributed to the project to make it a robust, well-supported platform for designing AR applications. Since its community is large and growing rapidly as AR becomes a more popular subject for research and development purposes, we decided it would be the ideal choice as the framework of our AR-based project. Documentation is also readily available for ARToolkit and there are numerous publications on projects and research using this library. This has helped us to understand what kind of functionalities have been demonstrated to work and what may be feasible to implement. Finally, ARToolkit is a free software library, which allowed us to limit the cost of the project. Since we have budget constraints, and purchasing the hardware required for the project has already depleted a large portion of this budget, we found great use of this free toolkit designed to make AR applications simpler to implement.

Before going into detail on how specific actions in ARToolkit can be performed, it is important to have a basic understanding of how the software development kit actually operates. First of all, a camera is used to capture video information about the real world. The camera is connected by USB to a computer which uses the ARToolkit software to interpret any square shapes in each video frame, which are possible candidates for markers. The image from the camera is converted to a binary image format to identify the dark marker borders. Next, ARToolkit determines the position and orientation of the camera relative to the marker. A symbol inside the border is then matched with existing marker images in memory. If there is a match, a 3D computer graphics model is drawn relative to marker position based on which symbol is identified. The model is drawn with an orientation based on the orientation of the marker. The image of the border of a detected marker in a video frame is no longer square if the marker is not directly facing the camera. The shape of the border, as well as the directionality of the image inside the border is used to calculate the marker's orientation, and the system uses this information to draw virtual objects accordingly. The

final output of these operations is then sent to the display, such as the goggles used in our project, so that the user can see the virtual model projected on a real marker. An example of an object drawn relative to marker position and orientation can be seen in Figure 4.

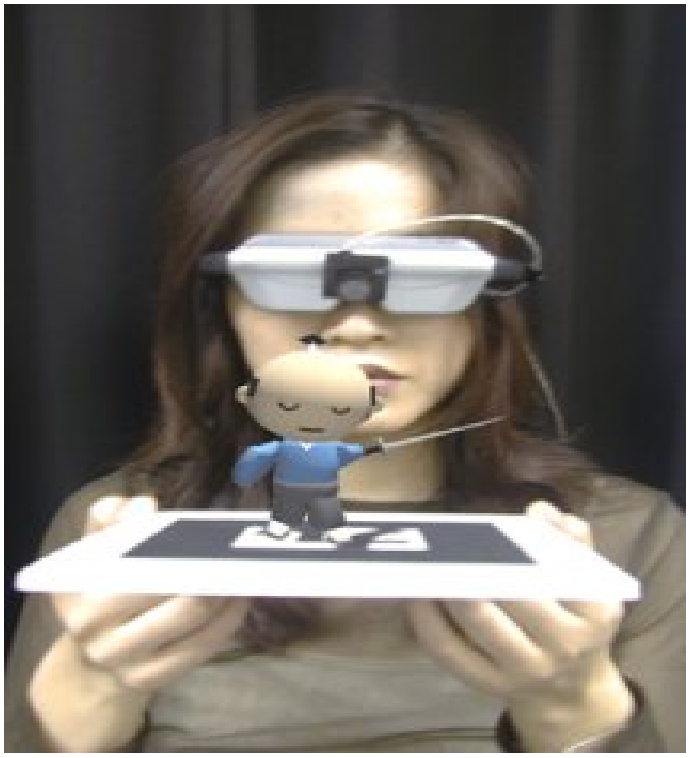


Figure 4: Depiction of a virtual object being displayed relative to a marker, as included in [2]

To add dynamic virtual objects to a user's display, the image must be continuously updated. To do this, our code makes GLUT and ARToolkit function calls within a loop that runs for the duration of the program. Information about object position is updated in the loop, and it draws objects onto the display accordingly. The ARToolkit function calls detect markers in video frames being sent from the input camera, and that information is used in the GLUT function calls to correctly position virtual objects. To avoid image flicker, and streamline image processing, newly drawn images are stored in a buffer. When it is time to update the display, the new image is already stored, and simply loaded from the buffer. The old image is then destroyed to prevent displaying old information and to avoid memory

issues. An example of the mainLoop code can be seen in Appendix I.

Images are understood in terms of orientation and position using OpenGL 3D transformation matrices. The reference frame of a particular virtual object is stored as a 4x4 matrix which includes a 3x3 matrix representing orientation in terms of its pitch, yaw, and roll, each of which has a value in each dimension, which is prepended to a 3x1 matrix that represents the actual x, y, and z coordinates of the position. Multiplying a 4x4 matrix corresponding to a particular reference frame by a 4x1 vertex transforms that vertex into the reference frame. Due to this, reference frames can be nested from most general to most specific simply by multiplying in that order, as can be seen in Formula (4)

$$frameParent * frameChild * frameNestedChild * vertexInNestedChildCoords \quad (1)$$

$$= frameParent * frameChild * vertexInChildCoords \quad (2)$$

$$= frameParent * vertexInParentCoords \quad (3)$$

$$= vertexInUniverseCoords \quad (4)$$

The general 4x4 matrix used to describe a reference frame or object can be seen in Formula (5). Figure 5 is a visual representation of possible reference frame information. In GLUT, ARToolkit, and Havok, the orientation of objects and reference frames are described by three orthogonal vectors. These vectors are the rows of the matrix in (5). Each vector defines the rotatation and scaling of a specific axis of an object or reference frame. The top vector's first three entries (marked $right_x$, up_x and $forward_x$ in the matrix) describe rotation and scaling of the x-axis, while the final entry ($position_x$) describes postion. The effect of the rotation vectors is more clearly understood when described in relation to an unchanging "master" reference frame. up_x describes rotation of an object's x-axis towards the y-axis of the "master" reference frame, $forward_x$ describes rotation of an object's x-axis towards the z-axis of the master reference frame, and $right_x$ describes scaling of an

object's x-axis. $right_y$ describes rotation of an object's y-axis towards the x-axis of the "master" reference frame, $forward_y$ describes rotation of an object's y-axis towards the z-axis of the master reference frame, and up_y describes scaling of an object's y-axis. $right_z$ describes rotation of an object's z-axis towards the x-axis of the "master" reference frame, up_z describes rotation of an object's z-axis towards the y-axis of the master reference frame, and $forward_z$ describes scaling of an object's z-axis. $position_x$, $position_y$ and $position_z$ describe an object's or reference frame's position in the master reference frame. The fourth row of the matrix contains no information, is always 0, 0, 0 and 1, and is there for convenience in matrix tranformations. ARToolkit does no even allow you to manipulate this row, and the corresponding data structure only contains information about the first three rows. Havok, GLUT, and ARToolkit all use different structures to store this information. ARToolkit uses a 3x4 array of floats, GLUT uses a length 16 array of floats, and Havok uses a 4x4 array of floats.

$$referenceFrame = \begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

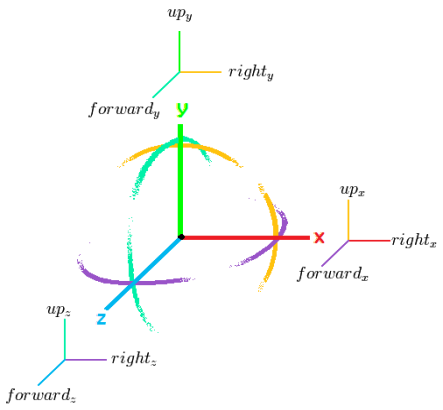


Figure 5: Illustration of possible transformations of a reference frame

3 Related Work

Many recent strides made in AR can be seen in recently developed smartphone applications. The portable video screens and cameras included in most smartphones are convenient for the purposes of AR. Additionally, smartphones often contain a variety of sensors, such as GPS and compasses, that provide alternative methods of determining how to orient and position virtual images onto the display. One example of a popular AR application for smartphones is Layar [7]. Layar uses GPS and compass information to identify businesses and services in the viewfinder of a smartphone and then overlays information about those businesses and services on the display. Marker-identifying AR applications have also been developed for smartphones, but they serve as more of a novelty. Markers are largely incompatible with the use-anywhere appeal of smartphones. This is a different paradigm from our program, which stresses immersion with a simple but necessary setup. The primary focus of many popular smartphone applications is providing information. AR simply serves as an intuitive presentation of that information. There are less attempts to allow user interaction with the virtual content. Of course, all AR applications are theoretically possible on smartphones, but projects of the nature we are undertaking have not gained backing.

The only requirement for a system to be considered an AR application is that it must overlay virtual objects onto some display of the real world. As such, an AR application may perform any processing desired on a computer associated with the AR display. These calculations may be constrained by the total processing power available but are by and large compatible with any AR system. This means that physics simulations can be used to enhance the capabilities of an AR system. Perhaps the best demonstration of this concept that we encountered is "Forked! A Demonstration of Physics Realism in Augmented Reality" (Forked) [10]. Forked uses ARToolkit with the Havok physics engine to create an AR display with applied physics, as seen in [10]. The end result is that a physical robot with a marker attached is mapped into the virtual realm and may thus collide with virtual blocks projected in the environment. The environment in Forked is a surface which is defined by multiple

redundant markers to allow multiple camera angles and movement of the robot which would obscure markers. Additionally, due to the fact that the system understands the dimensions, position and orientation of the robot in the real world, it is capable of visually occluding the blocks, enhancing the illusion. The robot can successfully lift and transport the virtual crates. In fact, the creators, interested in the believability of the AR system, performed a series of tests in which users performed tasks with real crates and then virtual crates. A small increase in completion time was observed when participants interacted with the virtual crates, but some of the performance was attributed to the discomfort of the video goggles used. We emulated the Forked system in many ways. In particular, implementation of simulated collisions is highly desirable for immersion purposes, and we believed this would greatly enhance our end product. Forked served as our inspiration in trying to adapt Havok to the project. We also implemented a multiple redundant marker system similar to the one used in Forked, although this idea was conceived prior to encountering Forked.

Featured at the "Technology Entertainment and Design" conference (TED) in 2009, "SixthSense" is a recent AR project whose goal was to overlay information about objects encountered in day-to-day life [18]. To achieve this goal, SixthSense uses a small camera, a processor and a projector mounted on the chest. Major features of SixthSense include the ability to use blank paper as a screen, to identify and display information about real objects, and to manipulate virtual content with your fingers. Colored rings worn on the forefinger and thumb allow the device to easily detect user input from those digits. SixthSense demonstrates several real-life applications of AR. These include the ability to copy text found in the real world into the virtual realm, to augment newspaper articles with video content, and to provide amazon.com ratings or pricing information for observed consumer products. There was much excitement surrounding the initial release of information about the SixthSense project, but as of now it remains in the prototype phase [18]. The project's use of a projector, and its stress on portability deviate from the ultimate goals of our project. However, their use of simple markers for the most intensive input from the real world or the user validates

Figure 6: Playstation Move controller

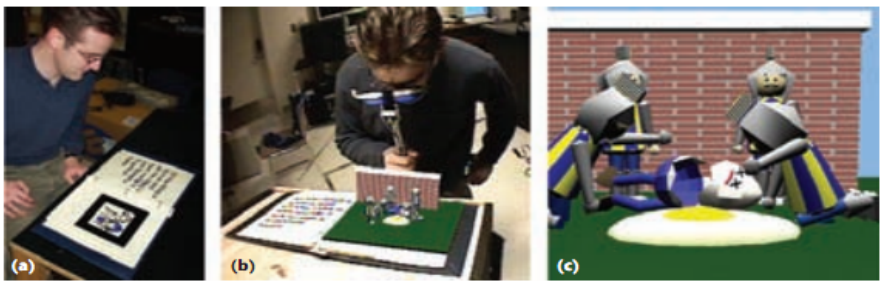
our decision to use a marker-based system in our project. It remains clear that markers are currently an important part of complex AR interaction.

Some of the major breakthroughs in AR technology have come about recently as peripheral equipment for video game entertainment systems. The Playstation Move released only recently for use with the Sony Playstation 3 gaming console is a motion sensing platform that translates a users actual movements to actions in the context of the game world [17]. It operates using two main devices; the Playstation Eye, which is the camera that obtains video information from the user's environment, and the Playstation Move motion controller, a wand-like device that allows for user interaction with the Playstation game. The motion controller has an orb-like sphere at its top that acts as a fiduciary marker that the Playstation Eye interprets. The uniform shape characteristic of the sphere allows it to be accurately tracked in 3-dimensions and its distance from the Playstation Eye to be accurately measured. The sphere even contains light emitting diodes, which change color to contrast sharply with the users environment; this allows it to be a very effective marker. A Playstation Move controller can be seen in Figure 6.

The wand itself contains a linear accelerometer, which can track movement in 3 dimensions and a rate sensor. These allow the wand to track movements that are rotational rather

than just translational. Finally, it uses inertial sensors for dead reckoning in the case where the camera loses track of the sphere marker. This can happen when the marker is obscured by the user or another object or if the marker moves out of the viewing range of the camera. Dead reckoning involves estimating the current position of an object using a previously determined position. Dead reckoning is particularly useful in applications such as this, since there needs to be a way to interpolate the location of the wand when it is no longer detected. Using the inertial sensors, the system uses information such as the velocity and acceleration of the wand to perform an estimate of the wand's location. The problem of accurately retaining object information during marker occlusion is a topic of major research, and the Playstation Move demonstrates one way of accomplishing this without the use of redundant markers [17].

Another video gaming system that makes use of AR technology is the Microsoft Xbox 360 through its Kinect software technology. Unlike the Playstation Move, the Kinect is completely controller free. Instead, it detects physical behaviors such as hand movements, body gestures and speech input for interaction. In this way, it involves a system of detection and recognition that does not use any kind of static marker. Instead it relies on facial dectection and feature extraction to track a moving object. It then detects motion of parts of the body like the face, the hand, or a specific joint to perform motion analysis of an individual [12].



(a)reality, (b) augmented reality, (c) virtual reality

Figure 7: Milgram's reality-virtuality continuum levels in MagicBook

Figure 8: Milgram's mixed reality continuum

The project seen in "The MagicBook: Moving Seamlessly between Reality and Virtuality" (MagicBook) [11] is one example of a major success of AR using the ARToolkit software library. It is one of the rare projects in the realm of artificial reality that manages to encompass many different degrees of integrating virtual reality according to Milgram's reality-virtuality continuum. A representation of the mixed reality contiuum can be seen in figure 8. The project satisfies a child's fantasies by bringing regular picture books to virtual life. Using wearable augmented reality supporting displays, individuals can watch animated models appear on their book. These models represent the scene shown in a given picture. The models appear attached to the real page of the book and can be perceived from many angles like any 3D model. In addition, MagicBook also supports a fully immersive virtual environment. In fact, in some examples of MagicBook stories, readers are able to become a character in the book and interact with virtual characters as the story unfolds. Furthermore, MagicBook supports multiple users so that children (or adults) can share the story as characters. In this case, multiple people are part of the same scene and can even interact through face-to-face conversation. The most interesting part of the MagicBook project, however, is that the story can actually interact with the user on three different levels of the Milgram reality-virtuality continuum simultaneously. This can be seen in Figure 7, which we have copied from the MagicBook paper [11]. Most obvious is the reality level, because the user can see a simple picture book. On the other extreme is the virtual reality level, where users interact as part of the scene. Finally, in between is the augmented reality level,

where spectating viewers can see a scene unfold between humans acting as virtual characters in the virtual context of MagicBook. MagicBook uses a hand-held display for augmented reality, a small attached camera and a processing computer for graphics and ARToolkit, and the picture book. The pictures that allow this virtual interaction are surrounded by black borders to behave as markers, which ARToolkit uses for head-tracking. When the users head position is determined it is registered within the context of the virtual reality, which is generated on the real picture book page. MagicBook is only one example of ARToolkit's potential. Many other applications have been built using ARToolkit since its release.

4 Project Description

Using AR, we have developed a game in which a user is able to aim and fire balls at virtual structures and knock them down to destroy virtual "dragons". Virtual objects emulate actual objects by exhibiting behavior with simulated physics. The user is able to solve small puzzles and interact with the virtual environment to defeat the dragons. Pictures of the game in action can be seen in figures 9 and 10. Other demonstrations of a different nature have been implemented, including a proof-of-concept application wherein the user can use markers to draw three-dimensional figures that appear in the display and seem to be in the real world.

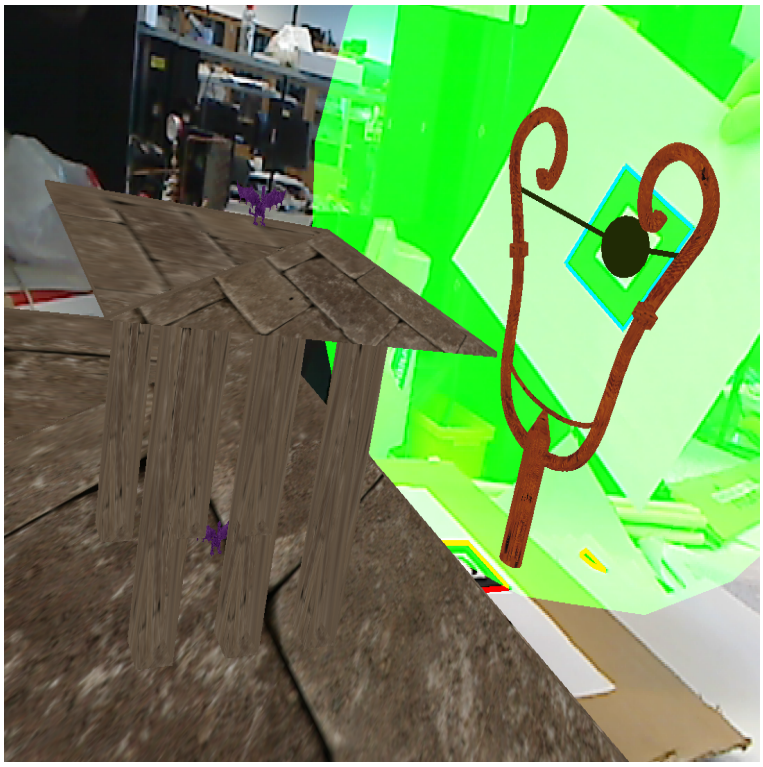


Figure 9: DD:WARE in action

Physical interactions between virtual objects and marker defined real objects are recognized. Specifically, phenomena present in real world physics have been implemented,

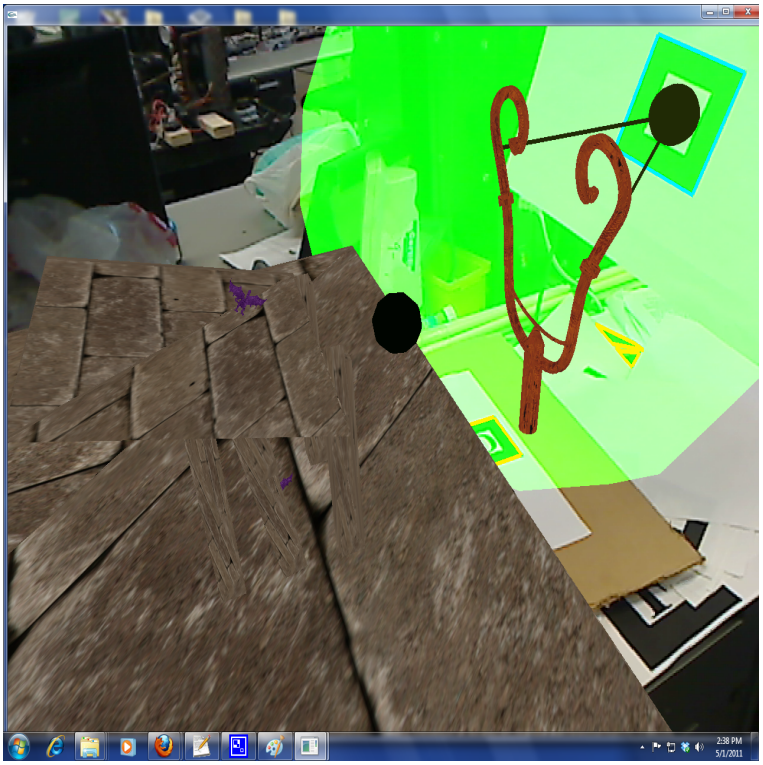


Figure 10: DD:WARE in action

including gravity, friction and collisions. In this way, all the different hardware and software tools work together to create an interactive AR based environment. When the user tells the game to fire using a mouse click, the position of a "slingshot" marker manipulated by the user is used to control the direction and velocity of a ball launched throught the slingshot. The marker's effect is defined in relation to a virtual "slingshot" whose position is defined by a group of redundant markers separate from the "slinghot" marker. The redundant markers also define the position of a virtual structure and "dragons" to be destroyed by the user. In any given simulation, the combination of physical markers and different functions implemented in the ARToolkit framework create virtual objects that are interpreted by the processing computer and displayed with the VR920 goggles. The Logitech Webcam Pro 9000 is mounted on the top of these goggles to approximate normal vision and achieve video see-through. The camera-goggle amalgamation can be seen in figure 11 The objects interact

with each other through simulated physics, using the Havok physics engine. Throughout this process, ARToolkit works to interpret the presence and orientation of markers in order to correctly understand how to display virtual objects. This creates a back and forth methodology, where ARToolkit identifies markers to determine virtual object position and orientation, the data is sent to Havok to determine the next step of how the virtual objects would react physically, and finally the now updated position and orientation information is sent back to ARToolkit to be drawn with GLUT functions.

Figure 11: Webcam attached to Vuzix goggles

Our first goal was to create our own ARToolkit-based program to interpret markers and place objects. In addition we incorporated the webcam, computer, and goggles together so that visual output is sent to the goggles. We created simple applications that detected markers and drew virtual objects in relation to those markers. After some experimentation we saw some issues with the robustness of marker detection. One step we took towards solving this was the use of redundant markers.

Our first progress was made in understanding the use of ARToolkit. The program is installed on multiple consoles and its linkage with GLUT, the graphics rendering toolkit, is set. While the underlying complexities of ARToolkit and GLUT remain unknown to an extent, and as a result were often treated as "black boxes" both libraries provide many built in functions that can be utilized. These have been heavily explored and suit a wide variety of purposes.

In testing ARToolkit's recognition capabilities, we have observed markers at a wide variety of angles, positions, and transitions between different angles and positions. ARToolkit successfully updates the drawn virtual objects in real time, but it is not uncommon to encounter situations where ARToolkit is outpaced and there is observable lag in the observed marker position update. Moving the camera or marker was seen to be equivalent. More concerning than this is the tendency to lose sight of markers altogether. This was encountered mainly as a result of significant changes in lighting. Lighting is more of a concern than we initially expected for several reasons. User position can block a lighting source, markers can suddenly move to a shadowy area, and some environments are simply too dark for reliable marker detection. Camera calibration has mitigated detection problems associated with lighting. However, the improvement in reliability is difficult to quantify and lighting problems remain a concern.

Multiple redundant markers are used to provide spatial information about one object or a related group of objects. One of the major challenges faced was observed in initial experiments. If a marker was slightly obscured ARToolkit would not find it, and as a result virtual objects would disappear. Many orientations of the hand will obscure the marker and thus eliminate the virtual copy, which was an issue compounded by the fact that the camera situated on the eyes was moving. We may have been able to solve this problem by storing information about marker position, which would retain information and possibly even calculate movement in the absence of a visible marker. Doing such however would lead to innevitable inaccuracies as the camera moves relative to the undetected marker. Such

a system would complicate matters by conceivably retaining objects that were intended to be removed, which would require testing of loss conditions and proper retention time limits. Additionally and perhaps more importantly, this could prove a great increase on the processor load, which in turn slows the refresh rate and breaks immersion. A more satisfactory solution was previously discussed. This solution is the use of more than one marker to describe the position of the same object or group of objects. This solution requires the relative position of the redundant markers to be known in advance so that the program understands the offset necessary to draw the object in the same physical position. This limits the flexibility of our setup and adds a need for physical precision, but the negative aspects of this solution are outweighed by the increased resiliency of the system to occluded and undetected markiers. This redundant marker system is already included in ARToolkit, and although each individual redundant marker is less likely to be detected using this system, it is still more likely that any one of several redundant markers will be detected than any independant marker on its own. Redundant markers increase the ability of the user to move and observe from multiple positions and angles. When the camera loses one marker, or when a marker becomes obscured, the user can still see virtual objects as long as one or more of the redundant markers is detected. The use of ARToolkit also creates an issue in that the markers must be rigid. Small bends in a marker can prevent ARToolkit from properly identifying it. Although we did not include a system for recognizing non-rigid surfaces, implementation of such a system is possible, and ideas for such a system can be found in [16]. Non-rigid markers are highly useful for their ability to accurately map complex two dimensional (2D) surfaces and are less likely to fail to detect markers that are partially obscured. This would be ideal for mapping a complex real playing surface into the virtual reality. The system in [16] also contained an implementation simulating advanced lighting and texture, which could greatly enhance realism. The redundant markers system remains a much more feasible method of ensuring the virtual world is maintained and is aided by a built-in ARToolkit implementation. All markers were backed with cardboard to ensure rigidity and minimize

detection issues.

In addition, we experimented with various kinds of user input, such as garnering information through user manipulation of markers and the use of a keyboard and mouse. We prefer marker manipulation to keyboard or mouse manipulation because it provides a greater sense of immersion. Unfortunately, the hand itself is not well suited to being labeled with a marker. However, marker labeled instruments for the hand to manipulate have been developed. Although intially we wanted to avoid the use of keyboard or mouse input, the difficulties of creating intuitive and seamless user input using only markers was prohibitive, and we decided that using a limited amount of mouse input would allow the game to be easier to play and understand.

ARToolkit requires interaction with the GLUT graphics library to draw objects. GLUT is capable of drawing simple objects such as cubes, spheres and triangles. Drawing more complex 3D objects necessitates drawing composite objects out of these simpler objects. To do this, complex 3D objects are created in 3D design applications such as 3DS and Solidworks, and saved as 3DS files. 3DS files define objects in text chunks which can be interpreted as vertices on the surface of these complex 3D objects. These files are read and interpreted, and we use the vertice information in conjuction with the GLUT library to draw many smaller polygons (specifically triangles) which constitute the much larger 3D object. An example of an object drawn in this method can be seen in Figure 12. This method of drawing complex objects circumvented what would would have been intense coding work. Display of complex 3D objects is thus vastly simpified. In addition, using this method we could find existing, freely available 3D models and incorporate them into our game.

A secondary element of preventing markers from becoming obscured is calibration. Even when in plain sight, a marker might not be properly understood by ARToolkit. Alternatively, the marker may be understood incorrectly and give the wrong orientation. The symbols that differentiate the markers may even be confused, leading to the wrong virtual objects being displayed. To minimize these issues, the camera was carefully calibrated and various lighting

Figure 12: Complex 3D model of a Dragon

conditions were tested. New calibrations were undertaken in each area where we planned to play the game.

ARToolkit includes two programs for calibrating a camera for use within its own framework. One of these programs is just a simplification of the other. The simpler calibration process involves taking images at various distances and angles of a sheet of paper with a 4x6 grid of dots, each with a radius of 3.5mm, with centers placed 40mm apart from each other. The person calibrating the camera then tells the calibration program where in the image each dot is located using a simple point and click interface. Both programs have yielded positive results, but the improvement is limited, and definite improvement is hard to measure. In addition to the previously described calibration process, the more complex process requires tools for mounting the camera at precise vertical distances from a 32cm x 24 cm grid with grid lines spaced 4cm apart. Vertical and horizontal lines are overlaid on each grid line in an image taken by the camera of the grid. After grid lines are overlayed on the image, a new

one is taken with the camera 10 cm further away from the 32cm x 24 cm grid. This process is repeated at least 5 times before the calibration is complete. The apparatus used to hold the camera at precise distances from the grid is shown in Figure 13

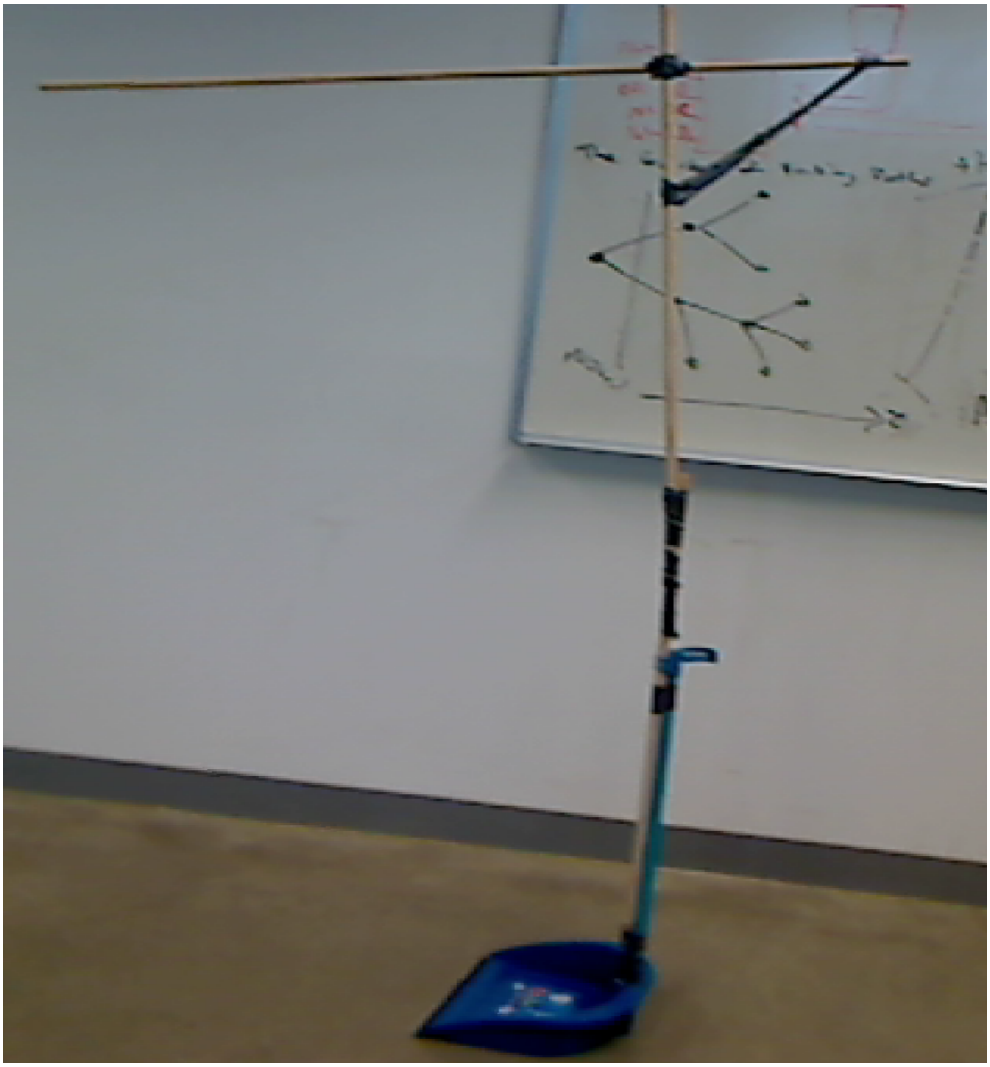


Figure 13: Calibration Apparatus

Another major problem we faced is that of occlusion. ARToolkit fails to recognize objects such as hands that should be in front of virtual objects but do not obscure the marker, and as a result strange, incorrect overlaps occur. The system cannot identify the dimensions or positions of real objects without markers that are associated with those objects. This problem can be mitigated by having virtual representations of objects that would commonly

obscure a virtual object (such as hands), using markers to find the position of those obscuring objects, and using that information to draw virtual objects differently. The major challenge in this is that the shape of a hand is not well suited to a rigid marker.

Using software included in ARToolkit we have defined our own markers, which allows us to easily create virtual objects by our own definition, and to create markers containing environmental information. A large number of markers can be defined (we have tested up to nine different markers at once, and use six in the final game), and no performance loss has been noted when increasing the number of markers used. Marker symbols can take any form within the marker square. ARToolkit includes programs for taking input images and creating files that ARToolkit uses for marker recognition. We used these programs, which required a camera image of the marker patterns we created. There is a slight degradation in performance of marker recognition for markers we have defined when compared to the default markers supplied by ARToolkit. Most commonly, false positives are observed. Camera calibration improved this problem to some extent.

In addition we have been able to use multiple markers to define single objects. This solves issues caused by the limited view of an input camera. When markers are partially obscured or outside the view altogether virtual objects can still be defined by redundant markers and displayed accordingly. This system is simpler, more robust, and is less taxing on memory resources than the alternative of maintaining positional information about markers that have been temporarily obscured from view. The redundant marker system is implemented in ARToolkit. There are some issues with it that must be discussed. Primarily, marker recognition for some part of a redundant marker system is inferior to a single marker implemented normally. The reliability increase we observed however more than made up for the failure to detect individual markers.

ARToolkit allows simple to use references for each marker defined in the system. Using these references, it can be checked in the code if a given marker has been identified in the field of view of the camera. This allows particular graphics to be associated with the marker. First

experimented with was the ability to draw multiple shapes associated with a single marker. This endeavor was successful showing that the simplest means of creating complex virtual objects with GLUT is the combination of several simpler objects. These objects can either be 2D or 3D. GLUT provides built in functions for both, as well as functions for correctly positioning and orienting those objects relative to the marker.

It is possible to import 3D models from various graphic design programs into GLUT. Specifically we imported 3ds files into our own ARToolkit programs. 3ds files contain blocks of information describing vertex locations for many small triangles. This information can be read and then used by GLUT functions. We successfully integrated 3ds file loading software into our own programs and were able to associate complex graphics models with specific markers, as seen in Figure 14.

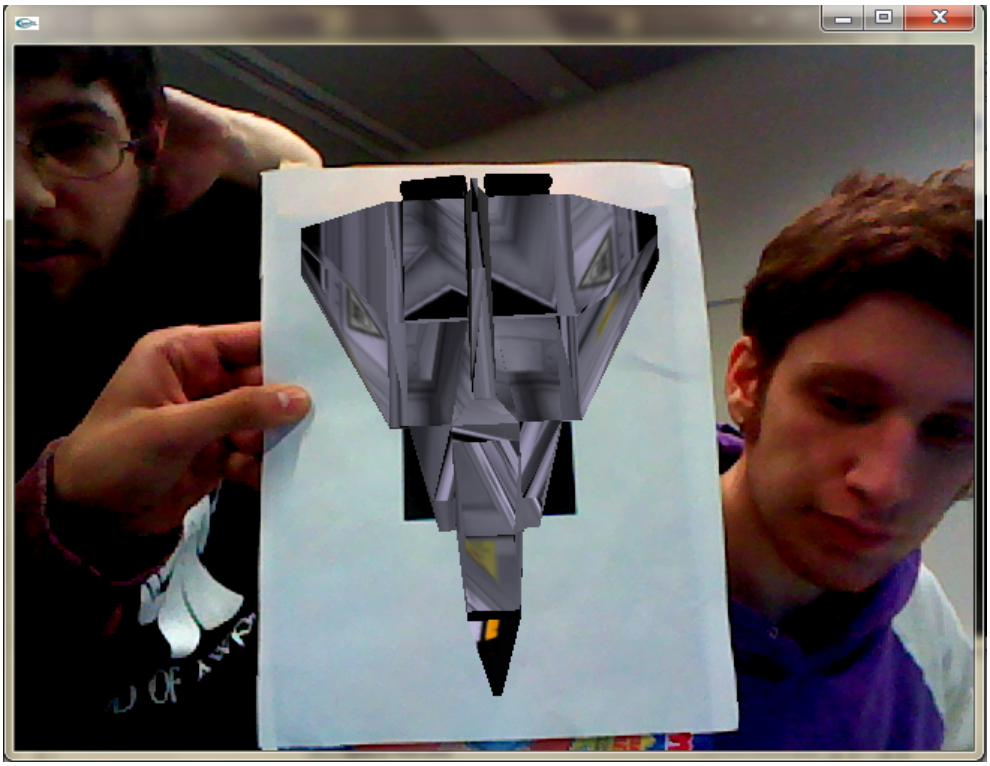


Figure 14: Complex 3D model of a sci-fi spaceship

There is another aspect to displaying complex graphics. Textures must be associated with each object. By assigning points in a bmp file to each vertice of the object a texture is

overlayed onto the object. Texture wrapping and stretching occurs when the bmp texture is not tailored specifically for an object.

While imported textures may be necessary for some complex objects, the simple colors afforded by GLUT's simple functions are still viable. In experimenting with these, we found that even these functions are capable of more than solid colors. It was found that when a color is specified in GLUT it is actually associated with a vertex rather than a face. The color of the face is the gradient of colors moving from one vertex color to another. As such, rainbow like effects are very simple and various aesthetically pleasing color mixes can be experimented with without the overhead of creating and importing a texture. A few colorations of this type have been experimented with, though none have been selected for use in the final product. This can be seen in Figure 15.

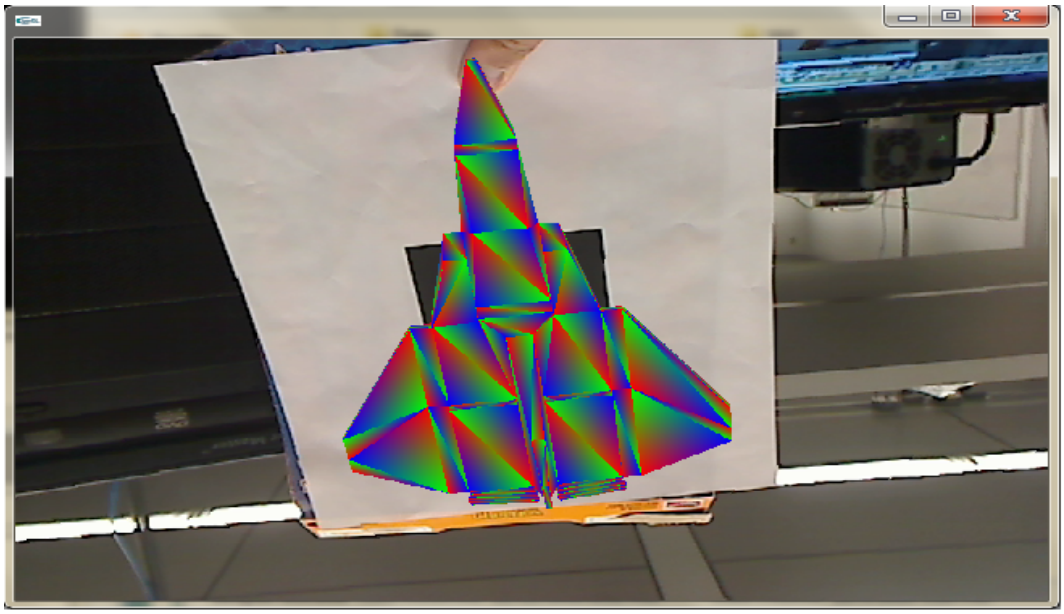
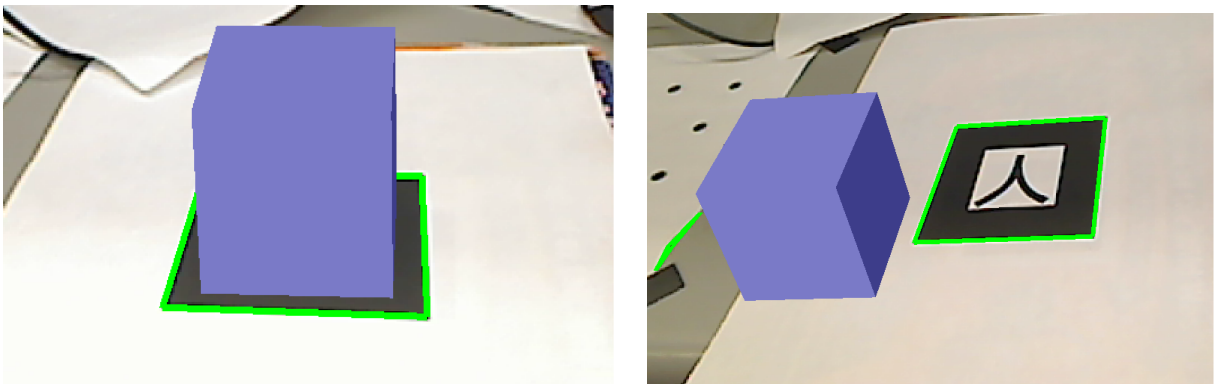


Figure 15: Vertex associated object coloration

The GLUT drawing functions are called inside of a loop that continues to run as long as the program is running. This means that virtual objects are continually redrawn and can be updated regularly. This allows for virtual objects to be animated. No degradation in performance for marker detection or display quality is observed as a result of animated virtual objects.

Virtual object animation can be preset in code or controlled by user inputs. The simpler forms of user input include keyboard and mouse commands. We added the ability to change the virtual objects drawn based on user input. We have experimented with user inputs that control position, color, and shape. Figure 16 shows screen captures for an ARToolkit based program that allows directional input from a keyboard to move a virtual object. All aspects of the virtual objects can conceivably be controlled by the user. We experimented with more complex inputs such as mouse position, and simultaneous keyboard inputs (e.g. two keys being pressed at once). Users can also manipulate markers to provide additional input information. In particular, users can obscure a marker as an input trigger. We successfully altered the virtual objects through the absence or partial occlusion of a marker.



(a) Unshifted Cube (b) Shifted Cube

Figure 16: Virtual object manipulation through user input in ARToolkit

Our focus at the beginning of the second semester was on making ourselves familiar with the Havok physics engine. During the first semester we had decided that integrating an already established physics engine to the AR environment would be a better option for

implementing physics when compared to creating our own physics engine. This is due to the fact that implementing collision based physics is not a trivial task and a custom physics engine would be limited in its scope. For instance, collisions would probably be perfectly elastic, and friction would be unincorporated. Furthermore, we were aware that Havok had a wide array of usable functionality beyond just collisions and gravity, which we felt would allow more flexibility in our final product and possibly more complex functionality. Finally, our hope was that Havok would also be much more efficient since implementing APIs to interact with Havok has been performed in a variety of projects before. In other words, creating our own physics engine would be like "reinventing the wheel" with the ultimate result being less flexible and much less useful than that of a project using Havok.

In learning how to use Havok, we first experimented by simply trying to create rigid bodies and applying simple physics like gravitational effects to them. Havok is a rigid body based physics engine. This means that Havok does not allow objects to be deformed physically at all. This effectively simplifies a multitude of actions which the physics engine performs since the way any rigid body physically behaves does not change due to its interaction with other rigid bodies. For example, a Havok object representing a sphere will never change its characteristics and so will always behave and react like a sphere. This is one fundamental limitation in Havok: the objects are not dynamic in terms of shape and its associated properties.

While researching how to deal with the creation and implementation of characteristics of rigid bodies, we realized that the standard shapes that are used in Havok include spheres and rectangular prisms. Other standard geometries include triangles, cylinders, and capsules. In addition to this, convex shapes can be made using defined vertices. Building other three dimensional objects involves a more complex method that combines the different standard shapes into a compound object. The process of defining a sphere and rectangular prism rigid body are similar and are the simplest to implement in Havok. It involves the following: defining the shape properties of the object (type of shape, radius, side dimensions, etc.),

specifying the characteristics of the object (mass, type of motion, position in Havok world, etc.), adding the object to the Havok world, and finally defining any conditions the object should have after creation (friction characteristics, angular damping, etc.).

Creating simulations using Havok involves creating a timestep that indicates the length of frames in which to step through. Our time steps were of a discrete, constant value so that each frame was one sixth of one second. This indicates how often the Havok environment changed state based on the motions occurring in each simulation. To view the result of our physics simulations in Havok, we used the provided Havok Visual Debugger program. This program shows all entities added to the Havok world and their behavior according to the simulation, which is updated based on the frame rate.

Since we were using the C++ Visual Studio 2010 Express version integrated development environment as our framework for building ARToolkit based solutions, we decided that it was necessary to find a method of creating Havok applications using Visual Studio. The motivation behind this was that it would mitigate make the challenge of integrating the C++ Havok source code into the C based ARToolkit applications that we were building. To accomplish this, we discovered a tutorial that explained the linking process and code behind a standalone Havok application built in Microsoft Visual Studio [6]. The result of the successfully built tutorial application was the creation of an instance of a Havok world, which contained a fixed platform as the only entity, a method for stepping the simulation using time steps, and certain physics characteristics that were not immediately apparent until further work was done to add non-fixed rigid body entities to the environment.

The next step in understanding the mechanics of Havok was creating other rigid body entities that had motion characteristics that were non-fixed. Using an example in the Havok documentation [2], we were able to create a sphere with given properties such as mass, position, radius, and collision type. The result of one simulation in which we added two identical spheres at different distances (but directly normal from one position on the fixed surface) with gravity implemented to accelerate the spheres towards the fixed surface is

shown in figure 17. At first, it seemed odd that the top sphere would stack on the bottom sphere. However, when considering that these were identical spheres that were directly in line with one another along the axis of gravitational force, this behavior is actually physically accurate.

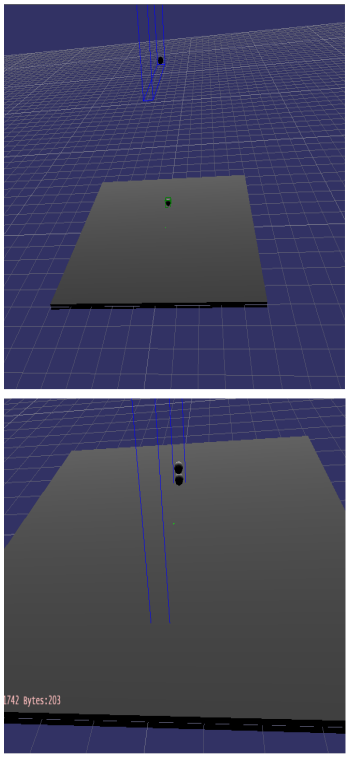


Figure 17: Depiction of Ideal Physics in Havok Visual Debugger

After we had a good understanding of how Havok works with frames, how to implement basic physics within Havok, and how to create and manipulate both fixed and non-fixed rigid bodies, we began the process of integrating Havok with ARToolkit. The tutorial we found was a great resource in this case since it contained certain C++ classes and related functions that we used to step simulations, update the Havok Visual Debugger, initiate Havok, initialize the actual world environment, handle the removal of objects that go beyond the defined borders of the Havok environment, and manage memory buffers and thread pools. This meant that we had to mainly focus on defining rigid bodies and their associated properties, initializing the mechanics of those bodies, adding them to the Havok world, and handling all of the

physics in an application loop.

While understanding the tutorial made using Havok a much more feasible task, there were several difficult challenges we had to face in integrating Havok with ARToolkit. First of all, we had quite a difficult time in dealing with linker errors while configuring the ARToolkit Visual Studio project to handle the same libraries and external dependencies that Havok utilized. The tutorial gave us a good starting point on which libraries needed to actually be added as well as the paths to additional library directories and the include directories needed from the Havok source code. However, the fact that ARToolkit is a software library written in C presented a problem since Havok makes use of C++ classes and other C++ specific functionality. To solve this issue, we changed the ARToolkit files to C++ files since we did not notice any aspect as this conversion that would result in errors. For the most part, this seemed to work fine, but we did encounter an issue in which one of the dependencies written in C produced compiler errors after we began integrating Havok into the project. To resolve this issue, we had to add the dependency as an extern "C" header file to the main project file since this was a non-system C header file that was now being included in C++ code.

Another major issue that we needed to address was creating a consistent interpretation of the environment between Havok and GLUT. One of the main things we noticed was the different interpretation of axes between Havok and GLUT when the marker is situated on a flat surface such as a table. In this case, Havoks negative y-direction corresponded with GLUTs positive z-direction, which we had to take into account whenever we made a call to Havok. Furthermore, using consistent and accurate positional and rotational information throughout our program was a challenge. ARToolkit, GLUT, and Havok store quaternion information in different kinds of matrices which made conversions between the different formats at all steps of any simulation necessary. In fact, during each step of the application loop we had to retrieve updated position and orientation data from Havok after which we converted them to ARToolkit and GLUT style quaternions to move the images.

In DD:WARE, most of the objects that define the field, which we create using GLUT,

were either rectangular prisms or spheres. These shapes were used primarily because their counterparts in Havok were primitives that could be easily created and defined with physics. The rectangular prisms were created in GLUT by using a contiguous set of triangles. Each side of the prism would be defined by two congruent right triangles that met on their diagonal, which yields a rectangular side. In this way, we were able to implement a system in which each figure we defined in our program using GLUT was actually represented physically as a rigid body in the Havok physics engine. Therefore, we could simply set some initial conditions for these objects after which Havok took care of the simulations. During these simulations we would repeatedly make calls to Havok from our game to update the representations of the objects in terms of their locations and orientations with respect to the field. An example of one instance of our program in which we display the program alongside its physical representation in the Havok Visual Debugger can be seen in Figure 18. The only major exception to exclusively drawing spheres and rectangular prisms in GLUT were the dragons featured in our game and the slingshot. The slingshot did not have any actual counterpart in Havok as it was not an object that needed to be physically dealt with. Its main purpose was to act as a reference for where the projectile would be fired from. However, the dragons did have to actually be mapped to some object in Havok as they were required to exhibit physical behavior and register collisions. To this end, we represented the dragons as spheres as seen in Figure 18. While this is not a perfect solution, we noticed no major problems since there was little chance that the projectile would pass through a part of the dragon in GLUT and not collide with the sphere on the Havok end.

Detecting collisions with the dragons in our game was important in order to register whether the dragon was "defeated" or not. To make this work, we added collision detection listeners to each dragon on the field. The listeners would check for any collisions with the dragon during each application loop. If a collision with any dragon was detected, we determined the object or objects that were the sources of the collision. We then located the specific points on the dragons rigid body representation that were affected by the collision.

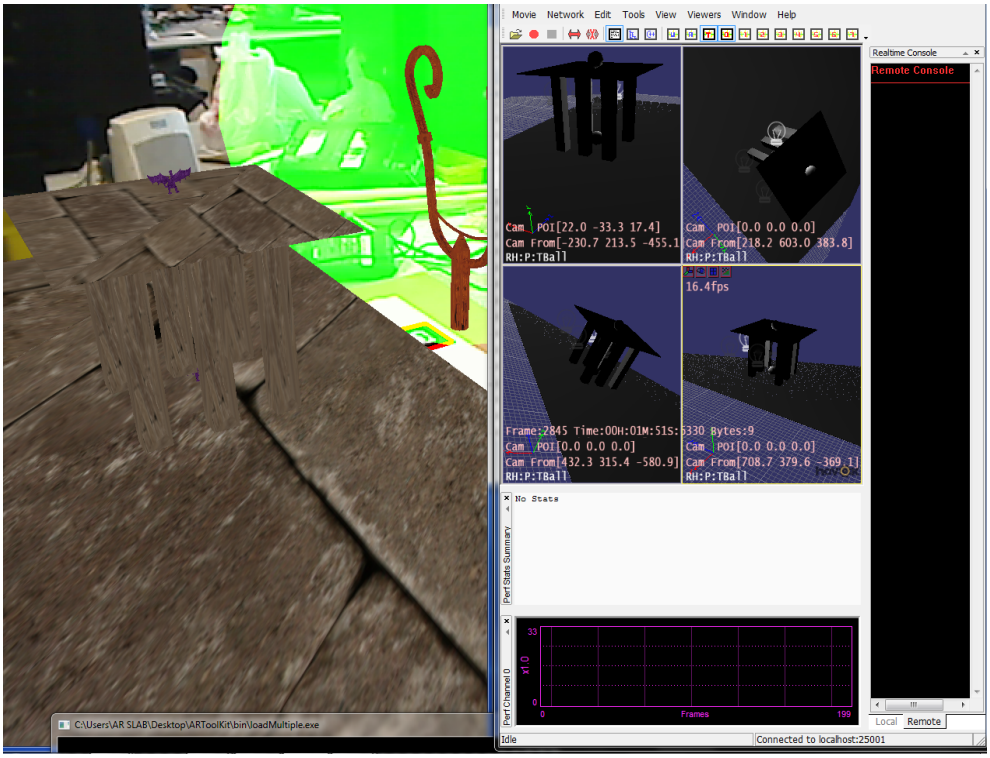


Figure 18: Side by side comparison of Havok Visual Debugger and game field

Finally, we calculated the total impulse delivered to the dragon during any moment and decided whether it is defeated or not by simply comparing it to some threshold minimum impulse value. If the impulse was greater than the threshold, the dragon and its rigid body counterpart in Havok were then removed from the game. Once all dragons are removed, the game ends and the player is victorious!

There were difficulties encountered due to the black box nature of the Havok libraries. When objects move outside of some range of the center of Havoks virtual representation of the field, Havok reuses the memory these objects took up for new objects. The information stored at that memory however, was not completely correct. The result of this memory reuse was that, when drawing objects in GLUT, often the older objects and newer objects referred to the same memory location and thus were drawn together. This often resulted in spheres that had cubes overlayed upon them or vice versa. We charmingly dubbed the result of this bug “sphubes. Using an explicit lazy deletion of these older objects we were able to overcome

this bug.

Ultimately we successfully integrated Havok, GLUT, and ARToolkit, and the game created was both novel and fun.

5 Future Work

There are several directions in which this project can be expanded. These additions range from aesthetic to functional improvements to drastic changes in user interaction. One of the most appealing to us is the integration of a user draw function into the final game. Definite success is seen in allowing the user to define shapes and sizes through marker movement and occlusion, as one of our demos clearly shows. The ability to interact with the world and the users options as a whole are expanded with the addition. Actual integration into the game, such as allowing the user to dynamically define the projectiles used, is only restricted by time to add the feature. The implementation is largely clear to us, and the biggest challenge would probably be simply maintaining game balance so it does not make the game too easy or too hard.

Another aspect cut largely for time constraints is visual additions. The number of textures and shapes currently used within the game is relatively small. While this may seem like an inessential measure, one of the main issues experienced while playing is difficulty judging depth and relative positions. It is possible that more complex patterns, varied landscapes, and additional work with virtual lighting, shadow, and perhaps even reflection could greatly enhance the users ability to judge distances. Aside from aiding in ease of use, these features would clearly enhance the aesthetic appeal of the game and would almost certainly lead to a further sense of immersion. This definitely seems like a path worth pursuing, even though it is not as closely tied to the technical aspects of the project as other elements.

The other half to improving the visuals is including animations. While object positions are updated according to the physics simulation and marker positions, no preset or otherwise triggered motions occur in the current incarnation of DD:WARE. Improvements in this area increase the number of events that can be simulated. Moving obstacles and targets were briefly considered, but the addition seemed inessential to the game. Projects more interested in expressing the virtual environment than the user interaction would put more consideration into animated portions. Additionally, animation creates concerns for interaction. Realisti-

cally reconciling the preprogrammed movement and physics dependent movement created by the user can be very complex. This could be part of a larger series of tests into maximizing user immersion.

In terms of future work expanding on the project, there are many options available from the framework of marker detection with integrated physics. Rather than the goal of entertainment that we achieved, more functional goals could be pursued with this technology. Possible uses include practicing tasks virtually and various immersive learning environments. The future experimenters could even run tests to analyze improvement in real tasks through interaction with the virtual environment. These are just a few examples. The framework has very few restrictions.

One of the most ambitious goals a followup project could investigate is visual processing more than rigid markers to improve user interaction. One possible method we considered is integrating hand-tracking. Particularly, there exists a low cost solution involving wearing a glove patterned with jagged, disparate colors to visually approximate hand positions [21]. Allowing the user to interact with the environment using the whole hand would severely increase immersion and allow a whole new field of applications. This would be much more complex than marker recognition, though, and even if successful would probably require careful allocation of processing power. Balancing the use of hand-tracking with the less intensive marker detection framework we provide could be profitable, especially if future experimenters wish to maintain the physics simulation element.

Finally, flexible markers in general are a definite possibility for expansion even without hand-tracking. By correctly interpreting non-rigid markers the valid angles from which a marker can be used are increased and the restrictions on a markers condition are decreased. Virtual objects would leave view less often and it would be possible to use the markers to understand complex surfaces they are placed on. Experiments could be done on retexuring or virtually representing existing objects. To put it simply, flexible markers improve flexibility.

6 Conclusion

In many ways, AR has grown and developed through its short history. We can consider the birth of AR to be when it was officially named "augmented reality" for the first time in 1999. Like any eleven year old, while it has a substantial amount of progress to cover before it can be considered well-developed or mature, AR has made significant strides within the last decade. Currently, it is on the verge of a period of rapid growth with the advent of AR applications in motion based video gaming systems such as the Playstation Move and the Xbox Kinect. Furthermore, it is rebellious, as it seeks to find its place among its more prominent relatives in the area of artificial reality. Unlike virtual reality, AR has yet to be accepted by a wide variety of applications and has had relatively little exposure. However, as it continues to be nurtured to maturity, people will begin to recognize its advantages and utility for a wide variety of purposes from entertainment to hand-held aides to military HUDs. ARToolkit in particular has been an invaluable resource to AR developers and anyone intending to use AR technology. By handling the difficult hurdles of image processing and model generation in the process of AR development, users can concentrate on building applications that creatively mix virtual components into reality. As the name suggests, this technology effectively "augments" reality in that the basis of the environment is the physical world. The virtual elements are projected onto this environment with the fiduciary markers used as reference points for the creation and display of the virtual elements. ARToolkit uses a system of transformation matrices for obtaining the appropriate perspective of any image, video frame buffers to store captured data, and GLUT to draw the object as intended into 3D space. Therefore, viewed in context of an AR application, ARToolkit can be seen as the black box between the video input and the output display. Since it is open-source, there is also plenty of progress that can be made in the areas ARToolkit handles by simply modifying the source files.

In our venture into the field of AR we created a virtual environment with integrated physics that the user could view and manipulate through fiduciary markers. Through these

elements we implemented a game wherein the user can choose the power and direction of a virtual slingshot to hit and destroy virtual targets. Collisions and gravity allowed realistic falling and crashes that expanded the interactivity and improved the feeling of immersion. While certain aspects of control could have been more fine tuned for ease of use and the field of view at times became cluttered, we feel our project demonstrated AR's capability. The addition of objects was flexible, physics translated well to the virtual world even in the presence of real objects, and redundant, rigid markers maintained the environment for the user.

A particular success of the project is the use of the video goggles. More than anticipated, the video see-through implementation with the camera mounted on top of the goggles was very intuitive. Aside from periods of lag dependent on processing capability, the human brain seems to adapt well to viewing the world through a camera filter. Using the video goggles both maintained the presence of the virtual world, as we desired, and to our pleasant surprise, made using the game more intuitive. We highly recommend this relatively simple method of enhancing AR.

Maintaining the proper virtual environment is what this project is all about. The markers must be identified and understood. The virtual objects must act and look like real objects would. The virtual objects must react to the relevant physical objects and the user must be able to interact with them intuitively. Creating this kind of well maintained mixed reality world is what guided our research and experimentation, and it is the same spirit we hope to see in any future derivative works.

References

[1] Armedia plugin 2.0 for google sketch-up. `http://www.inglobetechnologies.com/en/new_products/arplugin_su/info.php`.

[2] Artoolkit documentation. `http://www.hitl.washington.edu/artoolkit/documentation`.

[3] Artoolworks. `http://www.artoolworks.com/`.

[4] Fairy trails. `www.freeverse.com/fairytrails/`.

[5] Flartoolkit. `http://www.libspark.org/wiki/saqoosha/FLARToolKit/en`.

[6] Havok physics tutorial. `http://piotrpluta.opol.pl/programming/havok-physics`.

[7] Layar. `www.layar.com`.

[8] Logitech webcam pro 9000 specifications. `http://www.logitech.com/en-us/webcam-communications/webcams/devices/6333`.

[9] Ronald T. Azuma. A survey of augmented reality. *In Presence:Teleoperators and Virtual Environments*, 1997.

[10] David Beaner and Brian MacNamee. Forked! a demonstration of physics realism in augmented reality. Dublin, Ireland, 2009.

[11] Mark Billinghurst, Hirkazu Kato, and Ivan Poupyrev. The magicbook- moving seamlessly between reality and virtuality. May/June 2001.

[12] Adam Bunker. Exclusive: How does microsoft xbox kinect work? August 2010.

[13] Havok physics. `http://www.havok.com/index.php?page=havok-physics`.

[14] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kushino. Augmented reality: A class of displays on the reality-virtuality continuum. *SPIE*.

[15] Christopher Mitchell. Applications of convolutional neural networks to facial detection and recognition for augmented reality and wearable computing. Master's thesis, Cooper Union for the Advancement of Science and Art, New York, New York, 2010.

[16] Julien Pilet. Augmented reality for non-rigid surfaces, 2008.

[17] Luke Plunkett. Playstation move: Everything you need to know. March 2010.

[18] Sixthsense. `http://www.pranavmistry.com/projects/sixthsense/`.

[19] Jim Vallino. *Introduction to Augmented Reality*.

[20] Consumer video eyewear. vuzix: View of the future. `http://www.vuzix.com/consumer/index.htm`.

[21] Robert Y. Wang and Jovan Popovic. Real-time hand-tracking with a color glove.

7 Appendix I: Example Code

```
1  static void mainLoop(void)
2  {
3      ARUint8          *dataPtr;
4      ARMarkerInfo     *marker_info;
5      int              marker_num;
6      int              i,j,k;
7    double           err;
8
9      /* grab a video frame */
10     if( (dataPtr = (ARUint8 *)arVideoGetImage()) == NULL ) {
11         arUtilSleep(2);
12         return;
13     }
14
15     if( count == 0 ) arUtilTimerReset();
16     count++;
17
18   /*draw the video*/
19     argDrawMode2D();
20
21   //red if can't shoot yet
22   if(mainloopcount > 200)
23   {
24     glColor3f( 1.0, 1.0, 1.0 ); //reset after drawing
25   }
26   else
27     glColor3f(1.0, 0.0, 0.0);
```

Figure 19: Example of mainLoop code

```
1       argDispImage( dataPtr, 0,0 );
2     glColor3f( 1.0, 0.0, 0.0 );
3     glLineWidth(6.0);
4
5     /* detect the markers in the video frame */
6     if(arDetectMarker(dataPtr, thresh,
7       &marker_info, &marker_num) < 0 ) {
8       cleanup();
9       exit(0);
10    }
11
12    for( i = 0; i < marker_num; i++ ) {
13      //visual confirmation
14      argDrawSquare(marker_info[i].vertex,0,0);
15    }
16    glColor3f( 1.0, 1.0, 1.0 ); //reset after drawing
17
18    /* check for known patterns */
19      for( i = 0; i < objectnum; i++ ) {
20      k = -1;
21      for( j = 0; j < marker_num; j++ ) {
22              if( object[i].id == marker_info[j].id) {
23
24          /* you've found a pattern */
25          glColor3f( 0.0, 0.0, 1.0 );
26
27          //visual confirmation
28          argDrawSquare(marker_info[j].vertex,0,0);
29  if( k == -1 ) k = j;
30                else /* make sure you have the best pattern (highest confidence
                        factor) */
31              if( marker_info[k].cf < marker_info[j].cf ) k = j;
32        }
33      }
34      glColor3f( 1.0, 1.0, 1.0 ); //reset after drawing
35
36      if( k == -1 ) {
37        object[i].visible = 0;
38        continue;
39      }
```

Figure 20: Example of mainLoop code

```
/* calculate the transform for each marker */
    if( object[i].visible == 0 ) {
            arGetTransMat(&marker_info[k],
                          object[i].marker_center, object[i].marker_width,
                          object[i].trans);
        }
        else {
            arGetTransMatCont(&marker_info[k], object[i].trans,
                          object[i].marker_center, object[i].marker_width,
                          object[i].trans);
        }
        object[i].visible = 1;

  }
  arVideoCapNext();

  /* draw the AR graphics */
    draw( object, objectnum );

  //step the simulation and VDB
    hkpWorldCinfo testInfo;

    havokUtilities->stepSimulation(timestep);
    havokUtilities->stepVisualDebugger(timestep);
    havokUtilities->getWorld()->getCinfo(testInfo);
    hkVector4 testVect = testInfo.m_gravity;
    hkVector4 gravs = havokUtilities->getWorld()->getGravity();

  mainloopcount++;
  pigsRemain = false;

//detect collisions with pigs
  for(int i=0; i<1024;i++)
  {
    if(objectArray[i].type == PIG)
    {
      pigsRemain = true;
      forceDet = detectCollisions(objectArray[i].body, i);
      if(mainloopcount > 200 && forceDet)
      {
        havokUtilities->getWorld()->removeEntity(objectArray[i].body);
        objectArray[i].type = IGNORE;
      }
    }
  }
```

Figure 21: Example of mainLoop code

```
1    if(pigsRemain == false)
2    {
3      keyEvent('r',0,0);
4    }
5    else if(numVelBalls>=10)
6    {
7      keyEvent('r',0,0);
8    }
9
10   if( (err=arMultiGetTransMat(marker_info, marker_num, config)) < 0 ) {
11         argSwapBuffers();
12         return;
13     }
14
15   if(err > 100.0 ) {
16         argSwapBuffers();
17         return;
18     }
19
20   /* draw the multimarker pattern */
21     for( i = 0; i < config->marker_num; i++ ) {
22         if( config->marker[i].visible >= 0 )
23         multidraw( config->trans, config->marker[i].trans, 0 );
24         else
25         multidraw( config->trans, config->marker[i].trans, 1 );
26     }
27   argSwapBuffers();
28 }
29 }
```

Figure 22: Example of mainLoop code