Dynamic Simulation of Virtual Objects for Augmented Reality Applications. Development of an Augmented Reality Chess

by

Marios-Fotios Bikos

A thesis submitted in partial fulfillment for the diploma of Electrical and Computer Engineering

in the
Department of Electrical and Computer Engineering

June 2015
In order to enhance immersion in augmented reality systems, solutions must not only present a realistic visual rendering of the virtual objects, but also allow natural hand interactions. The main goal of this project was to utilize and introduce advanced techniques for the superimposition and manipulation of virtual objects over the view of the real world for mixed reality simulations. In this work, a board of markers was used for computing the camera pose seamlessly and a pinch gesture detection algorithm was implemented, employing users thumb and forefinger to interact with the virtual content, using an RGB-D camera. Ultimately, a Mixed Reality Chess was developed, focused on providing an immersive experience to users, so that they are able to manipulate virtual chess pieces in front of a real table and play against a chess engine.
## Contents

Abstract

1 Extended Summary

1.1 Introduction .................................................. 1
  1.1.1 Problem Description and Related Work .................... 2
  1.1.2 Objectives ............................................. 3
  1.1.3 Development Tools and Frameworks ....................... 3
1.2 Experimental Setup ........................................... 5
1.3 Marker Tracking .............................................. 5
1.4 Pinch Gesture Detection ...................................... 8
1.5 Chess Engine Integration ..................................... 11
1.6 Gameplay Programming and Rendering ......................... 13
1.7 Occlusion Handling .......................................... 19
1.8 Limitations and Drawbacks ................................... 21
1.9 Future Work .................................................. 22

Bibliography .......................... 24
Chapter 1

Extended Summary

1.1 Introduction

Augmented Reality (AR) applications turn devices with cameras into magic lenses, through which real and virtual objects appear to coexist in the same space. This illusion is achieved by superimposing virtual content on top of the real world. To improve the feeling of presence in mixed reality worlds, a certain level of immersion should be provided by the system. This can be achieved through the interaction with virtual objects, which is a challenging area for further research. A variety of interaction techniques have been used to manipulate virtual objects in AR applications. However, most of the approaches are based on 2D touch screen pointing and clicking, and these methods suffer from having limited input area, using 2D input for 3D interaction. All these break the illusion that the users can interact directly with the virtual objects in the real world and are not pushing towards seamless interactions between the user and the virtual content.

Recent research has shown that allowing physical interaction between free-hand and virtual objects enhances user experience and increases the realism of virtual content in the real world. Compared to conventional device-centric interaction methods, natural gesture-based interfaces offer a more intuitive experience for AR applications. Combined with depth information, gesture interfaces can extend AR interaction into full 3D space. There have been different approaches using computer vision and image processing techniques to detect gestures, however, in the last few years, low-cost depth sensing devices as Kinect or Leap Motion have offered great opportunities to track hand gestures and detect natural free-hand interactions that can be applied to mixed reality environments.

In this thesis, we attempt to use 3D gesture interaction as an alternative input technique for AR. We propose and implement a solution to integrate a pinch gesture detection technology
within the augmented reality environment to offer a more robust and seamless integration of
the user gestures with the virtual objects, specifically using the Intels Realsense RGB-D sensor
device. We demonstrate our tracking system in an augmented reality chess game allowing a
user to interact with virtual chess pieces.

1.1.1 Problem Description and Related Work

Augmented reality chess games have been implemented on previous works. One approach
[1] used a handheld pen prop with a marker cube on top of it in order to interact with the
chess pieces. As the writers admit, the tracking of interaction props was inaccurate and slow
to provide unencumbered and natural use. Another one[2] used finger tracking techniques that
allow gestural interaction with the chess pieces. In this approach, manipulation of virtual objects
is possible using grab and release gestures, as well as image processing techniques to detect hand
gestures, using a single camera. The finger tracker that is implemented is using a hands 3D
model that can determine enough information to robustly track the position, orientation and
pose of the users index finger. However, this solution resorts to using a marked glove with
retro-reflective spheres on top of the forefingers joints, something that may disturb users and is
definitely not a natural way to interact with content. Other approaches utilize mobile markers
that correspond to a specific type of chess piece and users had to move the markers to different
positions of a chessboard in order to play the chess game. Of course this would take quite an
amount of effort to correctly print and create the configurable chessboard and is really similar
to just using a real chessboard with real chess pieces.

The current possibilities that low-cost color and depth sensors like Kinect or RealSense 3D
cameras offer, can help us track gestures, eliminating the need for non-natural gloves and pen-
markers, which obstruct the gesture naturalness of the users interaction with the virtual content.

The hand/finger interaction within augmented environments faces two major challenges:

- The users fingers should be able to physically interact with virtual and real objects in an
  almost seamless way.

- The mutual visual occlusion between virtual and real elements has to be of convincing
  quality. The correct occlusions between the users fingers and the virtual objects in the
  AR space should be as correct as possible.

On this project we focus on these 2 goals considering the detection of a pinch gesture to interact
with the content and we try to manipulate virtual objects as physical ones, e.g. when we try to
grab a virtual pawn, the pawn should be rendered between our thumb and other fingers based
on the depth of each element.
The problem of interaction with virtual objects with physical gestures in augmented reality environments is a challenging task that faces several problems that break the user experience. This task becomes even more difficult, in the context of the development of an augmented reality chess. Many problems arise during the game of chess, since:

1. Users have to be able to see all the virtual chess pieces at the same time, in order to figure out what their next move is going to be

2. While a user makes a move, his hand occludes the chessboard which may be a marker or a number of markers and this may lead to incorrect camera pose estimation or no marker tracking at all.

### 1.1.2 Objectives

The main contribution of this thesis will be the integration of a color and depth sensors capabilities into an AR environment in order to provide better interaction with the virtual content. In our approach we used the Realsense Dev Kit has the advantage of being smaller compared to the Kinect. Since the Realsense 3D camera was recently released, there are not many studies related to the use of it, which is an advantage to conduct future studies with this technology.

The pinch gesture is one of the most common gestures for interaction with digital interfaces. It is defined as the movement of expansion and contraction of a finger spread. It has been used for different purposes depending on target applications, e.g. the zooming metaphor by contracting and expanding, scaling or picking. It resembles a grabbing or picking action and offers natural signal to select or move an object in an interactive system and due to the nature of the thumb and index fingers, the pinch grabbing is precise and has high performance. While interacting with real objects, it is used in chess games from almost every average player. For our purpose, the pinch is a pivotal gesture to implement interactions within our Augmented Reality space. The gesture-based interaction will be narrowed to design and use a single pinch gesture detection algorithm. The goal of this work is to study, implement and integrate a pinch gestures tracking technique in the AR space, using a color and depth sensor device, ultimately aiming for the optimization of the interaction between the user and the virtual objects in an AR environment, focusing on the interaction with a single pinch gesture. The implemented algorithms will be tested using a chess game application and occlusion handling techniques will be implemented to provide an immersive experience to the users utilizing the depth data.

### 1.1.3 Development Tools and Frameworks

For this thesis, the following tools and frameworks were utilized:
Extended Summary

- **OpenGL.** A cross-language, multi-platform application programming interface (API) for rendering 2D-3D graphics.

- **ArUco.** Fast, reliable, cross-platform, minimal library for augmented reality applications based on OpenCV.

- **Realsense SDK.** The Intel Realsense SDK provides access to color and depth stream data of the Realsense 3D camera as well as blob detection, contour extraction, hand tracking and speech recognition algorithms.

- **OpenCV.** Library of programming functions aimed at real-time computer vision and video and image processing.

- **Qt.** C++ cross-platform development framework for application, UI & device creation. Only the QProcess class features were used which allows the communication with executables.

- **Windows 8.1 Pro OS,** running on a laptop MacBook Pro with Intel Core-i5(2.6GHz) processor and 16Gb of RAM memory.

- **Visual Studio 2010 Professional Edition.**

- **Realsense Dev Kit 3D Camera,** connected through USB port 3.0. An Intel developed depth camera in this Creative designed peripheral device supports full VGA depth resolution, full 1080p RGB resolution, and includes dual microphones

- **Markerboard(8x8):** Same size as a real chessboard

Prior to the design and during a process of testing, the libraries and frameworks to use was an essential aspect to consider. As explained previously, the Realsense 3D camera was chosen because of its capabilities against other devices, the Realsense SDK features and due to its size. Among a variety of libraries for Augmented Reality, the Aruco Library has been chosen due to some main features it offer, such as the detection of markerboards(markers composed of several markers), its simplicity, its robustness and the trivial integration with OpenGL. ArUco is well documented and its API provides a comprehensive guide through its different capabilities. Moreover, we used OpenGL for rendering, although OpenGL works with low level functions, and nontrivial tasks to interact with graphics increased the programming complexity. Of course, a 3D Games Engine could have been used, such as Unity3D, however we wanted to emphasize on the basic aspects and theory behind augmented reality techniques and therefore we opted to write code from scratch.
1.2 Experimental Setup

Our applications design and architecture has been built so that the Realsense 3D Camera can be mounted on top of an Oculus Rift HMD. Through the HMD user would be able to see the color stream from Realsense camera, creating a video see-through display. However, due to time and complexity constraints, we decided that the integration of Oculus rift within our application would be excessive. Hence, it has been decided to work in an experimental setup which would simulate the parameters of height and viewpoint, proving that our approach could work with Oculus rift and Realsense 3D Camera seamlessly.

In order to interact with the content and evaluate the pinch gesture interaction, we setup a testing area, on which the markerboard is placed on a table, easily reachable by a user sitting in front of it. The Realsense 3D Camera stands on a tripod in a height of approximately 60cm pointing its field of view towards the markerboard target in order to encompass the area of interaction above the target. Finally, the laptop is placed in front of the users view where the augmented video captured by the camera is displayed. The Realsense 3D camera is connected to the laptop which runs the application.

1.3 Marker Tracking

Augmenting the reality is the process of adding virtual information to images or videos. To do so, we need to know where to render the virtual information. While some approaches seek natural features such as key points or textures, fiducial markers are still an attractive approach because they are easy to detect and high speed, as well as precision may be achieved. An AR marker, like the ones used by ArUco, is a very distinctive element that can be easily detected. It can be used to calculate the extrinsics(pose) of the camera in relation to the marker so that we’ll be able to render in 3D knowing where is the (0,0,0) of the world reference system. Since the AR marker is very easy to detect, you can process frames and do rendering in real time. Detecting a single marker might fail for different reasons such as poor lightning conditions, fast camera movement, occlusions, etc. To overcome that problem, ArUco allows the use of markerboards. A markerboard is a marker composed by several markers arranged in a grid. Markerboards present two main advantages.

1. Since there have more than one marker, it is less likely to lose them all at the same time.

2. The more markers are detected, the more points are available for computing the camera extrinsics. As a consequence, the accuracy obtained increases.
Since the game of chess is a tabletop game that always uses a chessboard as a base for placing the chess pieces on, we considered that using a markerboard would be really similar to the use of a chessboard and the occlusion of the markerboard from the users hand would not affect the rendering of virtual chess pieces during a piece movement. That is why, we created a markerboard which consists of 64 markers in a 8x8 grid, just like a board that is used in chess games, as the main marker of the system. A paper published by the founders of ArUco described the methodology that can be used in order to create specific marker IDs for the markerboard which will have a high number of bit transitions so that they are less likely to be confused with environment objects. While previous works impose fixed dictionaries, the latest version of ArUco library proposes an automatic method for generating a board of marker with the desired number of markers and with the desired number of bits. We used the ArUco samples and created a board of highly reliable markers which corresponds to the size and dimensions of a real chessboard as described by the World Chess Organization.

![8x8 ArUco Markerboard](image)

Through the use of ArUco computer vision algorithms, we can detect the markerboard with a specific probability value. When we develop augmented reality applications, we have to display OpenGL graphics superimposed on the realtime video feed that we get from the camera. To do this, we must first calibrate our camera as an offline process to determine the intrinsic parameters of the camera. The perspective matrix conventionally stays the same for the duration
of the application. In our application, we used the samples provided by OpenCV and a simple chessboard marker, where each square had a size of 2.5cm., that is usually used for calibration. After the calibration, we acquired an intrinsics .yml file which describes the camera parameters and can be used to undistort the image and help us determine the projection matrix that is going to be used in OpenGL. Thankfully, ArUco library takes care of the calculation of the perspective matrix, given the .yml file with the intrinsic matrix we got from the camera calibration.

![Camera Calibration](image.png)

Figure 1.2: Camera Calibration

On every frame of the real-time video feedback, the augmented reality application uses the intrinsic matrix, and in correspondence with the markerboard that is tracked, can give us the rotation and translation (model-view matrix) of the OpenGL frame. Therefore, for drawing a virtual object in our scene, we used the current model-view matrix and the perspective matrix and we got the position of the center of the markerboard. To achieve the augmented reality effect, we need the color stream from the Realsense 3D camera in our openGL scene. We created a texture from the color pixels of the video stream and applied this texture to a quad that was placed on the back of our scene. After that, we can render virtual objects with respect to the center of the markerboard. In OpenGL, we wrote code which loads .obj files with models of chess pieces and we placed them correctly in the markerboard. Also we rendered a virtual chessboard which consists of cube primitives correctly scaled. We decided not to use a chessboard 3D model, because using cubes we can change their color according to the features of our program. For example we could change the color of cubes during a move by the user, to visualize the possible moves a chess piece can make.
1.4 Pinch Gesture Detection

In this thesis, we implemented a pinch gesture recognition algorithm that triggers a grabbing action to move and release virtual objects in the AR scene. Andrew Wilson [1] presented a computer vision technique to detect when the user brings his thumb and forefinger together (a pinch gesture) for close-range and relatively controlled viewing circumstances. This technique avoids complex and fragile hand tracking algorithms by detecting the hole formed when the thumb and forefinger are touching. This hole is found using a simple analysis of the connected components of the background segmented against the hand.

We figured out that using advanced hand tracking techniques such as the ones that are used in latest publications, or the ones provided by the Realsense SDK would be an exaggeration for this application and also would not match, since part of the hand and fingers are not visible from the camera when a user moves a chess piece. Therefore, we dont really need to track the whole hand nor all the fingers. That is why, our approach, inspired by the paper of A.Wilson was based on the blob data and the hole formed when forefinger and thumb come together. A blob, in this context, is a shape identified within an image, which represents a single object.

More specifically, we utilized the blob detection capabilities of the Realsense SDK and we implemented an algorithm to detect when a pinch gesture happens and where this gesture happens in 3D space, so that we could use it later for correct piece selection and manipulation. The SDK interfaces allow us to detect objects in front of the camera (i.e. blobs) and extract the segmentation images, contour lines, and points of interest for these blobs. A segmentation image is an image representing a single blob, in which the blobs pixels are white (255) and the background pixels are black (0). Contour lines are the outer and inner border lines that separate the object from its background. The blob module is a convenient alternative to the hand tracking features of the SDK, since our application does not specifically require hand detection and identification. Each blob has an external contour line, and optionally one or more internal contour lines, depending on the blobs shape. Each contour line is represented by a series of points. Outer contour lines (i.e. external borders) are defined by an array of points in counter-clockwise order. Inner contour lines (i.e. "holes" in the mask), if any, are defined by arrays of points in clockwise order.
Based on the above definitions of blob data and contour data, the following algorithm was developed:

During each frame, we get the color and depth streams from the Realsense 3D Camera. Due to the misalignment and the physical offset between the color and depth sensors on the camera, we also needed a way to map the color pixels to the depth pixels and vice-versa. That is why we get the UVMap from the color and depth streams that will help us later in the implementation. We initialize the blob detection feature of the Realsense SDK, setting specific parameters, such as the maximum number of blobs we want to detect and the smoothing of segmentation and contours. We detect the nearest blob of the depth image, since the object that is closer to the camera and persons viewpoint during a chess game is the hand of the player. Once the nearest blob is recognized, we copy the data to a familiar data structure of computer vision called Mat(Matrix) and the number of contours is obtained from the SDK. If the number of contours is lower than 2, this automatically means that there isn't any hole formed in our depth image. Therefore, our program knows for sure that the user didn't use a pinch gesture. On the other hand, if the number of contours is 2, then there may be a probability that a pinch gesture occurred. To check if a pinch gesture really occurred, we acquire the points of the inner contour that may be the points of the hole formed by the users forefinger and thumb. If the number of contour points is below a certain fixed threshold, then we concede that the
current contour data are not related to the hand blob and therefore we move on to the next frame. However, if the number of points is above that threshold, we continue and we calculate the left-most and right-most points of the inner contours points. Once we find these points, we calculate the straight line that is defined by these two points. Next, we create a neighborhood of a (configurable) number of points, that belong to the straight line we calculated and are located on the left of the left-most point of the inner contour. Of course, the more points we get, the more accurate the result, yet the slower the program. Once we estimate this neighborhood of points in the depth frame, we have the depth values for each of these points and therefore, we determine the average depth value of the pixels of the neighborhood that have a valid depth value. This depth value will later be used as the depth value of the pinch gesture of the user in 3D space. What we do next, is that utilizing the UVmap we mentioned previously, we map each of the neighborhood points to the color frame. We measure the average x and y values of the color frames pixel-points that have a valid value (since during mapping some points may not be mapped correctly). However we must project these color pixels to the camera coordinate system to get the correct x and y values in real world units, in our case meters. Once this procedure is complete, we have the x,y,z values for 1 specific point in 3D world coordinates (meters) which is considered the point where a pinch took place, or else the point in 3D space where the user decided to pinch. Of course a a flag is activated, indicating that a pinch has been performed. At the end of the day, what we have is the 3D pinch-point with respect to the color camera and the fact that a pinch gesture has been detected. Based on these facts, we can build the rest of our application and the implement the logic of the gameplay of our application.
1.5 Chess Engine Integration

After the implementation of the pinch gesture detection algorithm, we decided that for our applications purposes, user should be able to play against the computer. This way, a natural sequence of chess events would take place and the application would simulate a real chess game against an opponent, something that would help with the evaluation of the system. In order for the user to be able to play against a computer, normally, we need to implement artificial intelligence algorithms so that our program could think the next move based on the past moves made by the user. However the implementation of these algorithms for our chess game was
out of the scope of this thesis and a simple way for AI functionality integration had to be investigated.

This is where the chess engines emerge. A chess engine is a computer program that that receives a board position as input, analyzes the position of chess pieces and calculates the best move based on that board within a given amount of possible effort (in most cases a time limit). The chess engine estimates the next moves, but typically does not interact directly with the user. Most chess engines do not have their own graphical user interface (GUI) but are rather console applications that communicate with a GUI via a standard protocol. This allows the user to play against multiple engines without learning a new user interface for each, and allows different engines to play against each other. The GUI, in this case, is our application so far. What we need is a way to connect it with the chess engine properly.

Nowadays, the most used way of communication with a chess engine is the Universal Chess Interface (UCI) Protocol. The Universal Chess Interface (UCI) is an open communication protocol that enables a chess program’s engine to communicate with its user interface through a set of specific commands. In order to integrate a chess engine to the augmented reality chess game, we need to exchange commands(strings) with a chess engine. A chess engine receives commands via standard input from an application and outputs its responses to standard output. So it has no graphical user interface, no mouse input, no pictures, just a plain console window, it is nothing more than an executable.

To communicate with the engines executable, we decided to use a feature provided by Qt, through a class named QProcess which allows to start an executable file and easily read and write string commands from and to it. The interaction with the engine starts with a uci command that tells the engine to identify itself. It then receives commands that may change the values of options and the way output is presented by the engine. Afterwards, the engine takes as an input from our application the users completed move, and outputs the next best move for the enemy pieces. The chess engine is told to spend a specific amount of time to search for a best move. It starts its search and considers the best move before the time limit expires. For our application we considered that the time limit should be really small(40ms) so that we can get immediate feedback from our program.

Usually, chess engines dont have the ability to know whether or not a move command by the user is a valid one or not, based on the pieces type and the state of the board. However, we used the iCE Engine[link] which stores the possible moves for every chess piece and when an invalid move is taken as an input from the user, it returns the string Invalid chess move. This feature really helped us in changing the architecture of our code logic and makes it even easier for the user not to execute a wrong move at all. What is more, the engine we used can output the outcome of the game, so that our program can detect if the user won or lost the game. So
we implemented the basic functionality to send and receive strings to and from the engine and according to these strings the correct actions can be taken in our program.

### 1.6 Gameplay Programming and Rendering

Now that he know when the user performs a pinch gesture, the position of the pinch point in 3D world space coordinates and how to integrate A.I functionality, we can implement the logic of our chess game that will lead to the correct rendering of virtual objects during each action. It is widely known, that during a chess game between 2 players, consecutive moves are made by each player. A player can move a chess piece based on the type of the piece, its abilities to move to specific squares and whether or now another piece occupies a square. Also there are special moves, such as castling, where 2 pieces may move simultaneously and the game ends when a check-mate is detected(or a draw).

More specifically, when a user tries a pinch gesture having as a goal to move a chess piece, we must render the chess piece with respect to the 3D pinch-point so that the piece will move according to the pinch-point. If the user decides to make a move, he will translate the chess piece on top of another square of the chessboard and will pinch-out(forefinger and thumb not touching each other anymore), so that the piece can occupy a new square that may be empty, or may be occupied by another enemys piece. Once a move is completed by the user, our program should query the engine for a new best move based on the artificial intelligence algorithms. Once we have an enemy move, we can update the chessboard and the position of pieces on it. The above sequence of events is rather complicated to code, so we decided for our program to execute different commands based on the state of the game and whether or not a pinch gesture has been detected.

More specifically, based on the flag that is raised when a pinch gesture is detected on each frame, we designed an algorithm which will change the current state of the system. There are 5 basic states in our implementation, namely the FREE, PINCH-IN,PINCH-CONTINUOUS,PINCH-OUT and ENEMY-MOVE states. As you can see from the figure below, the states change according to the previous state of our program. If we analyze the move of a player carefully, we will realize that the system status can be one of these states:

1. **Free**: User doesnt do any pinch gesture, or our system cant detect any pinch gesture at that frame.

2. **Pinch-In**: A pinch gesture has been detected and so if the previous state of the system is free, now we have a new state which is called Pinch-In.
3. Pinch-Continuous: A pinch gesture has also been detected on the current frame and the previous state of the system was pinch-in. This means that the user may want to move a chess piece.

4. Pinch-Out. On the current frame, there hasn't been any detection of a pinch gesture. However the previous state was Pinch-In or Pinch-Continuous, therefore we might have a possible complete move by the user.

5. Enemy-Move: Our system enters this state after every pinch-out and if the move made by the user was valid. Also the system remains in this state while the animation of the enemy's move is running.

![Figure 1.5: Change of Game States](image)

Once we know what the current state of the frame is, we can continue with the second part of our algorithm which takes specific actions, such as which virtual objects are going to be
rendered or not and what calculations should take place for the correct update of the position
of pieces in the chessboard.

More specifically, if the current state is FREE-HAND, then we only need to render each chess
piece that exists in our board representation, which is a data structure of an array of pointers.
Based on the type of each chess piece, we can render the corresponding 3D model of the chess
piece over the correct square of the chessboard.

When the current state is Pinch-In, then user may want to select a chess piece. However we
have to find out which chess piece is the user trying to select. To achieve this, we get the 3D
position of the pinch point, based on the procedure we described previously and we calculate all
the distances between this pinch-point and each center of all the squares of the board that have
a piece on them and that piece belongs to player 1 which is our user. From all these distances,
we estimate the square which has the minimum distance from the pinch-point, as well as the
distance in meters. To prevent selection of a chess piece when the user uses a pinch gesture too
far away from the chessboard, a condition must be met, where the distance must be lower than
a threshold which was estimated based on tests. If there is a valid selection of a chess piece,
then during the rendering pipeline, we render all the chess pieces of the board, except for the
one that was selected. Instead, we render the selected piece not on top of its square, but instead
in the position of the pinch-point. If not, we have an invalid piece selection, because the pinch
was detected too far away from any chess piece and therefore we render all the pieces of the
board.
When we have a PINCH-CONTINUOUS state, then if we had a valid piece selection during the pinch-in state we re-render all the pieces of the board, apart from the one which is moving according to the pinch point, otherwise we keep rendering the pieces of the board.
In the PINCH-OUT state, if a valid chess piece was selected during the previous states, then we have to calculate where the pinch-out takes place in the 3D space, so that we can assign a new position to the moving piece. To do this, we first calculate the distances between the last 3D pinch-points 2D projection on the chessboard and each squares 2D center point. Afterwards, we get the coordinates of the square which is closer to the pinch-points projection. This square will be considered as the destination square of the chess piece. Once we have this square, we have a move that is attempted by the user. However, this move may not be valid, based on the chess rules. For example, the user may try to move a chess pawn more than 2 squares away from its initial position. To overcome this problem, we have to check for the validity of the move using the chess engine. We must transform the move which is actually 4 numbers which represent the initial and final coordinates of the square in the 8x8 grid of the chessboard into a UCI compatible string (e.g. 0102 becomes A2A3) and get feedback from the engine on whether or not the move is valid. If the move is not valid, then we dont change the internal board representation and the moving chess piece goes back in its initial square. Otherwise, we have a valid move by the user, so we activate a flag which starts the ENEMY-MOVE state, we update the position of the moving chess piece in the internal board representation and we render all the chess pieces that belong the updated board.
Finally, during the ENEMY-MOVE, what we do is that we ask from the engine for a move. This move is in UCI format, so we have to transform it to a series of numbers which corresponds to a move in the grid of our chessboard. Once we do this, we start the animated rendering of the enemys piece based on the interpolation between the initial square and the final square. In the end we render the pieces of the updated board.
1.7 Occlusion Handling

For a user to successfully perform tasks in mixed reality environments, a certain level of immersion should be provided by the system. In most approaches occlusions are not taken into consideration, so virtual objects are always rendered on top of physical objects. However, to create an immersive and realistic experience, the occlusion between virtual and real objects has to be managed correctly, so that users can look at a scene where virtual content blends with the natural environment. Such an approach allows us to achieve high level of user immersion since the augmented objects occlude the users hands properly; something which is not possible with conventional AR.

In OpenGL, when an object is rendered, the depth of a generated pixel (z coordinate) is stored in a buffer, called the z-buffer or depth buffer. This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the method compares the two depths and overrides the current pixel if the object is closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one. In the end, the z-buffer will allow the method to correctly reproduce the usual depth perception: a close object hides a farther one.
The main problem when dealing with occlusion is that usually there is no depth information of the real scene. In order to overcome this problem, the depth of the real world from the users viewpoint has to be employed. In the current work, we use the Realsense 3D cameras depth sensor to aid in the process of acquiring the depth map of the environment scene. The Realsense 3D Camera can measure the distance of everything it sees, creating a depth map. This way the depth of every pixel of the depth frame can be estimated. The depth image is basically a 640x480 (although it can be smaller) matrix where each pixel value represents the distance from the camera to that point in space (expressed in mm).

When a video card renders a virtual scene it computes occlusion from the depth buffer, i.e. the z distance of every object in the virtual scene. Usually when another virtual object must be rendered in the same pixel of the z-buffer, the method compares the 2 depths and overrides the current pixel, if the object is closer to the observer, which in this case is our camera.

The trick is to initialize the Z-Buffer of OpenGL with the depth values taken from the Realsense 3D Camera depth image before rendering any 3D virtual object and after rendering the quad which shows the color video stream. By doing so, when the chess pieces are rendered, they will be occluded by what it appears to be real-life objects such as our hand. Its like simulating a render of the entire environment in 3D and using the resulting z-buffer values.

So, in order to handle occlusion in our application, we had to map the whole depth image to the color image, transform the values in millimeters to meters and then write to the OpenGLs z-buffer. Before we jump into writing in the z-buffer we need to consider what kind of projection we use (ortho or perspective). So we have to modify the data based on the perspective view that the virtual objects are rendered. In our case virtual objects are drawn using perspective projection.

For a perspective projection, the relationship between Z and depth is non-linear. Specifically, it’s of the form

\[ depth = \frac{A}{Z} + B \]

\[ A = \frac{zFar \cdot zNear}{zFar - zNear} \]

\[ B = \frac{zFar}{zFar - zNear} \]

Also we have to bear in mind that points in front of the viewpoint have negative Z coordinates. After applying this transformation to the depth data we can write the data to the z-buffer.
Once we write our depth data to the buffer, we can see that virtual object and real objects (such as our hands) blend naturally together.

![Occlusion Handling](image)

**Figure 1.11: Occlusion Handling**

### 1.8 Limitations and Drawbacks

During the simulation tests, some limitations and drawbacks emerged, that made us redesign some aspects of our applications. First of all, during blob detection, when the user decides to select and manipulate a virtual chess piece, his hand might be too close to the table, and this has as a result for our program to detect the hand and the table as a single blob, hence the pinch gesture detection fails. These problems are a common thing when using other modern sensors such as the Leap Motion. However, we decided that since, changing the way a blob is detected by the RealSense SDK would be excessive, we could simulate the visual design of a real chessboard which is usually a cube with a specific height above the tabletop surface. Therefore, in this application, the chessboard is rendered as a grid of multiple cubes with a height of approximately 3.5cm, since this was the minimum value that our program could correctly detect the hand blob.

Another problem that has caught our attention during the test simulations, is that due to the wrong detection of a pinch point in 3D space, the user may accidentally move a chess piece in another location than the one he intented to. For example, he may want to move one of his chess pawns 2 squares away from its initial position, but during the translation, the pinch may not be detected and therefore the pawn may move only 1 square away from its initial position. We considered that there may be several solutions for this problem. For instance, instead of the current approach, we could implement a timer effect before a valid move takes place. Then,
users would have to pinch for a specific amount of seconds before a move could be considered as valid. However, we opted not to do this, because we wanted the user to play chess just like in real life and also we could measure and evaluate the wrong and right moves he played during a chess game.

1.9 Future Work

Many prototypes implemented with sophisticated computer vision algorithms to robustly recognize hand gestures have demonstrated that gesture recognition is rather complex and computationally expensive. That is why our methodology is better, since we don’t use many resources and it is not computationally expensive.

Based on the results obtained, future work may include several improvements to the current prototype and additional features to manage a seamless interaction in AR and an even more immersive experience for the users.

The whole pinch gesture detection algorithm has been designed and developed, so that it would be possible to integrate the Realsense Development Kit Camera on top of an Oculus Rift Device. Our algorithms take into account the egocentric viewpoint of the user and his distance from a real table, thus they can work correctly without any changes. In order to achieve the integration of Oculus into our program, one would have to render whatever the application renders, including the video stream from the Realsense Camera, into the texture of a Framebuffer object of openGL. Afterwards, this texture could be applied in a quad and this quad could be placed in a virtual scene made by OpenGL. This virtual scene and the quad that is located inside it would then have to be also copied in the texture of a framebuffer object and passed into the Oculus SDK for modifications, such as distortion and creation of a stereo view from different viewpoints per eye.

In order to get more accurate and robust results on the exact 3D position of the pinch-point which is detected when thumb and forefinger come together a filtering of frames could be implemented to infer gestures in time and this could improve effectively the interaction errors.

One could also improve the visual feedback to enhance the users perception of the virtual world. The use of shadows and light source estimation for correct illumination of the scene and virtual objects can greatly affect the realistic rendering of the virtual content in augmented reality. Furthermore, in order for the user to better perceive the depth of the scene, enhanced visualizations can be utilized such as projections in all planes of the scene. In our application we have already used one such visualization by rendering a straight line of projection from the moving chess piece to the chessboard and it was obvious that users can move virtual chess pieces and better understand where the new position of the object is going to be. Finally, since we
use 3D models for the virtual chess pieces, a broad collection of different figures could be used, so that user would be able to play chess with different sets each time, ranging from dragons to soldiers. Adding multiple sets of chess models could definitely improve user experience, but in order to further improve the visual experience attack and death animations could be implemented. Instead of pieces disappearing, destruction through an explosion or decapitation would be even better.

Taking everything into consideration, our approach seems to work pretty nicely and the user is able to play a chess game with virtual objects from the beginning to the end without any serious problems.

Figure 1.12: Demo Screenshot
Bibliography


