# Checkers AR

Nikolas Chaconas
Jenna Cryan
Sharon Levy

## Abstract

*This paper describes Checkers AR, a real-time augmented reality iOS application built for the Iphone 6/6s. Using board corner tracking, it allows users to augment virtual checker pieces onto a real checkerboard and play a game of checkers. Checkers AR utilizes Swift[1] and Objective-C[2] for iphone development, as well as OpenCV 2[3] for camera calibration/board tracking and OpenGL ES 1.1[4, 5] for rendering the game pieces. In addition, the application also suggests a move for players, thus helping those who are not quite familiar with the game.*

## 1. Introduction

Augmented Reality is a form of technology in which elements of real life are layered over by computer generated sights and sounds. Unlike virtual reality, which completely replaces the real world with a simulated one, it is a link between the real and virtual world and combines elements of both into a single application. It also lets users interact with their environment in real time by overlaying new information as it is received.

This project combines multiple well known algorithms from OpenCV's C++ library and OpenGL ES for Swift and Objective-C to create a realistic augmented reality checkers game. Our goal was to create an experience for two players to play a game of checkers without the need for board pieces, that could perform in real time with realistic attributes. To interact with the game, users can tap on board pieces to highlight them, and tap onto valid board spaces to move the highlighted piece to that spot. Suggestive arrows will show where the player can make their next move, always giving moves which capture opponent pieces higher precedence. This precedence is simply due to the checkers rule that when a player is able to capture an opponent's game piece in a given turn, that move **must** be taken. We mention this to demonstrate that no moves suggested by our program necessarily show the player a best move to take, simply one that can (or must) be taken. All checker rules are followed with these hints, including a piece being queened once it reaches the other end of the board. We also would like to note that we used a board size of 8x6, which is not the standard for a checkerboard. Our reasoning behind this is explained during section 3.3.

Swift was chosen to be the primary language for iPhone development due to members having prior knowledge of it. We will go into more detail later as to how this caused difficulty during development. OpenCV was chosen due to its support for tracking checkerboard corners. OpenGL was chosen for its effectiveness in displaying realistic 3D objects. Our work contributes a unique approach to building an iphone application that creates a satisfactory augmented reality checkers experience.

## 2. Related Work

Augmented Reality has become very popular in recent years, with many applications created such as Pokemon GO[6] which lets players capture pokemon based on location and battle with other users. Star Walk[7] is an application allowing users to point their phones at the sky and receive information regarding astronomical objects in real time. In addition augmented reality devices are being produced such as the Microsoft HoloLens[8], a headset using voice, sight, and hand gestures to interact between the real and virtual world. Snapchat Spectacles[9] are a recently released set of smartglasses, allowing users to record videos through cameras located on the sides of the glasses and upload them to Snapchat.

## 3. Setbacks

Throughout our development, we encountered many setbacks due to our choice of language for core development, processing limitations due to mobile development, and difficulties maintaining proper board orientation.

## 3.1 Integrating OpenCV with Swift

Due to our combined limited knowledge on the technologies we would be using, our initial approach to this project was to learn as much as we could about iPhone development, board tracking in OpenCV, and fixed function pipeline rendering with OpenGL. Our first challenge was figuring out how to have OpenCV communicate with Swift, as OpenCV does not have any support for the Swift language. To counteract this, we were forced to create bridging header files in a mixture of Objective-C and C++ to allow our main Swift controller to communicate with OpenCV. We soon realized that not only can Swift not communicate directly with OpenCV, but any public functions made available to Swift through our Objective-C++ bridging headers could not include any variables/methods defined within OpenCV's framework. This added another level of complexity to defining our public methods that Swift could invoke for camera calibration and tracking. Due to these setbacks, the majority of our code ended up being written in Objective-C++, despite our core language being Swift.

## 3.2 Processing Limitations on the Iphone

After overcoming difficulties having Swift communicate with Objective-C, we noticed our application was far too slow for real time when the camera lost view of the checkerboard, due most likely to processing limitations on the iPhone. Besides limited hardware, we believe that using Swift and its camera delegates for Image capturing is another contributes to why our application initially had poor performance.

As Swift was our main design language, we naturally decided to control all camera input through our Swift controllers. The consequences of this was that all Swift UIImage frames captured through our camera delegate had to be converted to matrices for OpenCV to compute upon, and converted back to UIImages following computation. We believe this introduced an unnecessary amount of computation, which could have been avoided using OpenCV's built in camera delegate for capturing photos. This gave us strong motivations for switching to Objective-C. However, due to time constraints and the amount of work it would take to convert our entire project, we kept swift as our backbone.

Poor performance was only evident when the iPhone would not find the checkerboard. When the board is found on a frame, OpenCV's *findChessboardCorners* can stop searching for the board. When the corner tracking algorithm cannot find the board, it continues searching until completion of a searching an image, causing a significant amount of lag and a drop in frame rate. To overcome this, we set a small delay every time the board was not found by OpenCV. Doing this caused the frame rate when the board was not found to be reasonable enough to continue gameplay. Of course this also meant having to wait a short delay before the board would be tracked again after losing it, however we found that amount of time to be unnoticeable, and that a delay of 20ms weighed both of these considerations well. Overall, we believe the benefits of adding the delay outweighed the negative consequences.

## 3.3 Maintaining Proper Board Orientation

The last major hindrance we met was properly tracking the orientation of the board. We found it difficult to maintain a proper orientation of the board when we used a classic 8x8 checkerboard. This was due to corner tracking finding a different orientation of the board after each frame, due to *findChessboardCorners* not properly distinguishing the width from the height of the checkerboard. Our solution to overcome this was to instead use a 6x8 checkerboard, which allowed our board tracking to maintain the proper orientation of pieces up to a reasonable angle around one side of the checkerboard (nearly 180°). We recognize that this is not the most elegant solution, and that we could have used methods involving the difference in checkerboard corner piece colors to maintain a proper orientation of the board. Currently our board size is a macro, and further calculations are based off of that macro, making it a trait that could be modified easily in future iterations.

## 4. Approach

After overcoming our main setbacks, we were able to focus on core functionalities our application would need to incorporate in order to meet our goals mentioned previously.

## 4.1 Camera Calibration

In order to effectively draw pieces which could match the depth of the checkerboard regardless of orientation/distance of the iphone relative to the board, we required camera calibration. The user is instructed upon opening the app to begin camera calibration, and cannot begin the game until they do so. Camera calibration involves the user taking ten photos of the checkerboard from different angles. After taking each photo, the user is prompted with its success by the calibrated image with the checkerboard corners drawn flashing onto the screen. A poor calibration would result in poor board tracking, as seen in Figure 1. Calibration was obtained using *findChessboardCorners* for the initial corner tracking, *cornerSubPix* to increase the accuracy of found corners, and *calibrateCamera* to take the corner data from 10 images, and find the intrinsic parameters within the user's camera. Calibrate Camera yielded a camera matrix as well as distortion coefficients, which described the different qualities and distortion specific to the users camera. As the qualities found through calibration are qualities which are unique to the camera itself, it is not necessary nor useful for a user to recalibrate their camera after obtaining a successful calibration.

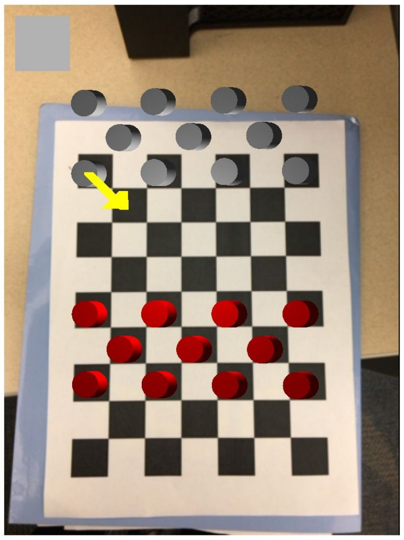
Figure 1: A poorly calibrated checkerboard

## 4.2 OpenCV to OpenGL

After a successful calibration, our next step was using *findChessboardCorners* to obtain corner locations during gameplay. We found that simply drawing shapes in OpenCV did not give a realistic effect of game pieces being on the board, as we will describe in more detail in 4.3. Thus, we turned to OpenGL for our rendering. To convert our two dimensional corner locations to three dimensional locations in OpenGL, we had to create both a perspective matrix from our intrinsic parameters, and a transformation/rotational modelview matrix from our extrinsic parameters. To build these, we had to take into account many factors:

1. OpenCV's coordinate axis is rotated 180 degrees around the x-axis compared to that of OpenGL.
2. OpenCV stores matrices in column major order, rather than row major order, requiring a transposition of OpenCV matrices.
3. OpenGL ES 1.1 for Objective-C expects matrices to be 32 bit float format, while OpenCV generally uses 64 bit double format.

The first two requirements were somewhat trivial (though did still take a fair amount of contemplation) to implement using OpenCV rotations and transpositions. The latter of the three cost us quite a bit of debugging in order to realize that we could **not** implicitly convert between these two data types - especially when working with matrices. Our initial implicit conversions caused all sorts of strange behavior and memory corruption, causing us a fair amount of headache. After solving these challenges, we were left with circular opengl elements which could remain stationary on the board regardless of camera orientation or distance, however we were not completely satisfied with their appearance.

## 4.3 Realistic Checker Pieces

As stated in the previous section, our initial checker pieces were drawn using OpenCV and were later created with OpenGL. Initially, these pieces drawn with OpenGL were flat circles that scaled accordingly and stayed on the checkerboard. These circles were created by drawing 360 vertices and connecting them into a filled shape. However, to give them a more realistic 3D look, we decided to generate multiple circles for each checker piece, stacking them along the z axis towards the camera. This technique did improve the look of the checker pieces, but since they were a uniform color, certain camera angles made them look deformed and not like cylindrical pieces.

In order to solve this issue, we decided to add lighting to give the checker pieces a more lifelike impression. This needed to be done by calculating the normals of the cylindrical pieces. However, our implementation of the checkers did not work for calculating the normals on the sides of the pieces. Thus, we were forced to create the checkers differently.

Our final approach for creating the checker pieces was to draw two circles, one on the checkerboard at z = 0 and another at 0.5 units above the checkerboard. Then, we drew a series of triangle strips connecting the vertices of the two circles together to form the sides of the checker pieces. In the end, our checkers looked exactly the same as our previous implementation.

To add in lighting, we now calculated the normal vectors of each vertex so that OpenGL could determine how the lighting should reflect off of each piece making up a checker piece. First, these normals were determined for the top circle, which all had the same values since all the vertices face upwards. To calculate the normal vectors of the sides, we created an array with the x and y coordinates equal to the cosine and sine of the current index for each index on the triangle strips. The z coordinate was equal to zero for these normal vectors because they were horizontal vectors. Lighting was then enabled for OpenGL and a light was set up two units above the center of the checkerboard, where one unit is equal to the side of one checker square. With the lighting set up , the checker pieces now looked realistic as seen in Figure 2, with diffuse lighting creating a gradient throughout each piece. This also made sure that the checker pieces still looked cylindrical when the camera was moved at different angles over the board.

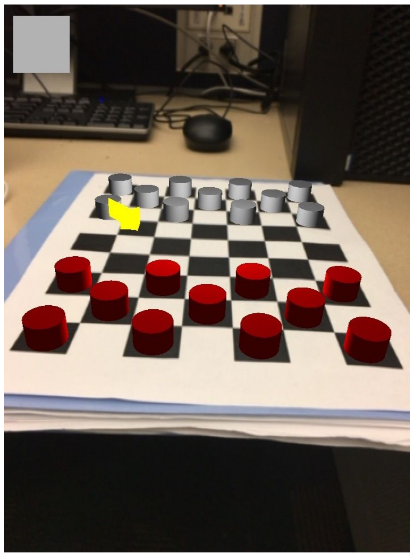
Figure 2: Realistic checker pieces with lighting

### 4.4 Two Dimensional Taps to Three Dimensional Space

In order to move the pieces and create a usable checkers game, we recognized the need for user interaction. Of course without any sort of user interaction to move pieces, the game was not playable. We first looked into using hand interaction to move pieces on the board, however any sort of occlusion on the board renders OpenCV's *findChessboardCorners* method useless. We considered an idea involving hand tracking and rendering pieces based on their last known position, and even rendering the hand to a depth buffer, thus making it possible for a user to use hand interaction. In this way, a quick hand gesture could occlude the board momentarily, but the pieces could still be rendered below the hands depth buffer into their last known location, giving the illusion of pieces positioned below the hand. However, we opted for user interaction with the game pieces to be done with taps on the iphone screen. In order for this work, we had to translate two dimension taps on the screen to three dimensional points onto the board.

Our first proposed methodology for this was to use our already built perspective and modelview matrices from 4.2 to translate our two dimensional taps to three dimensional space, similarly as to how we transformed our corner locations. However, we decided to employ another method, which utilized our already calculated two dimensional corner positions.

Our chosen methodology involved creating a data structure holding all of the current two dimensional corners found by OpenCV. Relevant corners were then offset using an approximated or exact distance based off of their distance from neighboring corners to translate their positions into the centers of each black square as seen in Figure 3, and non-relevant corners were discarded. Every time a tap on the screen occurred, an iteration through this data structure occurred and measured the euclidean distance between the tap and the center of each black square. If the tap was within a certain distance of the center of a black square, that tap was validated and associated with the corresponding black square. We used a minimum distance of 12 pixels between tap location and center piece location, however this value is a macro and could be easily adjusted to increase/decrease touch sensitivity. This method proved successful in recognizing a user tapping on a game piece, without the use of any complex two dimensional to three dimensional projections.

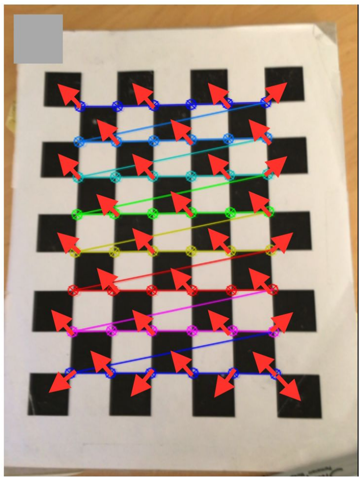
Figure 3: Offsetting Corners found in *findChessboardCorners*

### 4.5 Game Hints

Our last main feature was to integrate game hints to the user of where to make their next move. This was done by first computing what would be a valid move for the current user to make. For example, if there is a jump available, the player must take it instead of a single space move. In addition, once a checker piece makes its way to the other side of the board, it is "crowned" and can now move backwards as well as forwards. This logic was used to restrict users from moving pieces to invalid checker squares. Furthermore, for each turn, a yellow arrow appeared to give the player an option of a valid move to make, as seen in Figure 4. This hint is not necessarily the best move for the player to make but will always be a valid move. Players can know whose turn it is by looking at the colored square at the top left corner of the screen, showing the respective teams color when it is their turn.

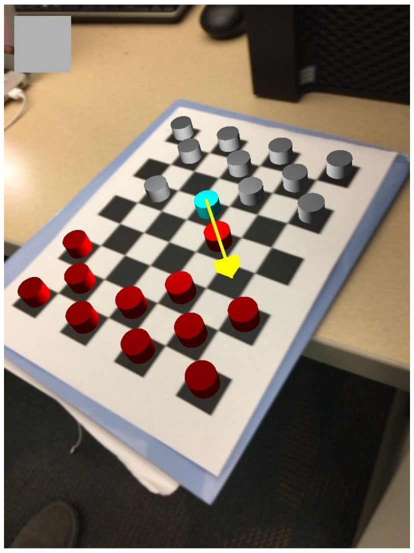
Figure 4: Move hint shown during a turn

## 5. Assessment

### 5.1 Ground Truth Comparisons

Before performing any kind of user study, we wanted to assess the accuracy of the appearance of the game, to ensure users were viewing as close to a real game of checkers as possible. We examined images and video of real life checkers games being played to establish what we wanted for our acceptable baseline. Since this resembles a game used by human subjects, the rendering didn't need to be perfect in terms of placement of the game pieces, but users need to easily identify the state of the game (i.e. shouldn't guess which tile a game piece sits on). Accordingly, we decided the checkers didn't need to be perfectly centered in the tiles. To remain easily distinguishable, though, we ensured the checkers in the application remained completely contained within the edges of a single tile, even as the device moves around the game board at different angles. Additionally, as the device moves around the board, we performed testing to check for realistic appearance (i.e. the game pieces stay on the board and don't begin appear as floating checkers). The application performed well for this metric with proper camera calibration for the checkerboard, and very poorly without, suggesting an acceptably accurate rendering of the game in 3D space.

### 5.2 User Study

To assess practical usability for real users, we performed a user study where subjects used the application followed by answering a series of questions about their experience. The study included data from 22 participants total, selected by availability (random selection of friends and fellow students). Most questions were on a numeric rating scale (1-7) to provide us with some qualitative data, with an opportunity at the end to provide open feedback on what the subjects liked, didn't like, and how they would improve it.

Questions asked include:

- How well did the application overall simulate real gameplay experience?
- How familiar are you with the game of Checkers?
- How useful are the suggested moves via arrow?
- How accurate does the scene appear in terms of the lighting?
- How easy is the application to use?
- How could the application improve in terms of accurate replication of the real-life experience?

Overall from the data we gathered, we determined good usability of the application. All subjects successfully calibrated the camera and checkerboard before playing a full game of augmented checkers against one of our team members. The results are summarized in Table 1. The general feedback received was positive, with lots of suggestions for future work. Many users enjoyed the augmented game experience, but some

Table 1: Summary of User Study

| Metric | Average Score (max. 7) |
|---|---|
| Simulate Real Gameplay | 6.2 |
| Familiarity to Checkers | 6.8 |
| Usefulness of Arrows | 5.5 |
| Accurate Lighting | 6.4 |
| General Ease of Use | 4.3 |

users reported missing out on some of the aspects of playing with physical pieces (i.e. physically jumping checkers, hearing the 'clack' of pieces on the board). Additionally, the suggestive arrows did help those subjects who forgot how to play checkers, which agreed with our expectations. We didn't expect though, but discovered through this user study, that users who *did* remember how to play the game found the arrows to get in the way of a more enjoyable user experience. We considered all user feedback as ways to improve the application, but due to the timing of the user study at the end of development period, a lack of time prevented implementing the improvements. Instead, we examine these suggestions in next section.

## 6. Looking Forward

At the end of our development period, we came out with an operable iOS application for the iPhone. The application runs in real time with very little lagging, or delay. Some small delays remain due to the continuous checkerboard detection, but should have a minimal effect on user experience. Due to the limitations of the screen size, the usability suffered. As an iOS application, we experimented with the effects of running the application on an iPad in an attempt to overcome the hindrance of a small display surface area. Contrary to expectations, the rendering of graphics did not scale up to make use of the larger display. This remains an area to further explore in the future.

Alternately, instead of two users sharing one small screen, we propose another plan for future work that would rely on sensors within the device (i.e. compass, accelerometer) to indicate which player is holding the device and adjust orientation accordingly. Even further, we believe this game would work well as an application across a network that would allow multiple devices to connect and see an individualized view of the game. In lieu of two players, another area for potential work includes implementing an opponent using artificial intelligence to allow a single player to use the application.

To improve the appearance and experience for the users, implementing sound effects could decrease the artificial feeling the application currently gives off. Also, changing the suggestive arrows to be optional for the users would help remove clutter from the screen for those who already know how to play. Moreover, offering strategic move suggestions, instead of just legal moves, would better benefit these users. Due to short time, these features were left as future work.

## 7. Conclusions

This application successfully renders augmented gameplay for a two person game of checkers played on an iPhone. The application achieves an acceptable level of realistic gameplay, as determined by our user study when all subjects were able to play a full game. While there remains many areas for future work, this mobile application establishes a good baseline for which to build upon.

## References

[1] API Design Guidelines.
https://swift.org/documentation/api-design-guidelines

[2] Objective-C.
https://developer.apple.com/reference/objectivec

[3] Camera Calibration and 3D Reconstruction.
http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html. 2016.

[4] OpenGL ES 1.1 Reference Pages.
https://www.khronos.org/opengles/sdk/1.1/docs/man/

[5] OpenGL ES for iOS and tvOS.
https://developer.apple.com/opengl-es/

[6] Pokemon GO.
http://www.pokemongo.com/en-us/explore/

[7] Star Walk
http://vitotechnology.com/star-walk.html

[8] Microsoft HoloLens
https://www.microsoft.com/microsoft-hololens/en-us

[9] Snapchat Spectacles
https://www.spectacles.com/