The Graduate Review

Volume 7 Article 12

2022

Developing an iOS Game Application: Magnet Hockey

Trevor D. Wysong
Bridgewater State University

Developing an iOS Game Application: Magnet Hockey

TREVOR D. WYSONG
Bridgewater State University

Introduction

The Magnet Hockey app is built for iOS using, the SpriteKit framework in Xcode. iOS is an operating system that powers people to navigate Apple devices such as the iPhone, iPad, and iPod. Xcode is Apple's software development environment used to create iOS applications. Xcode is shown in Figure 1.

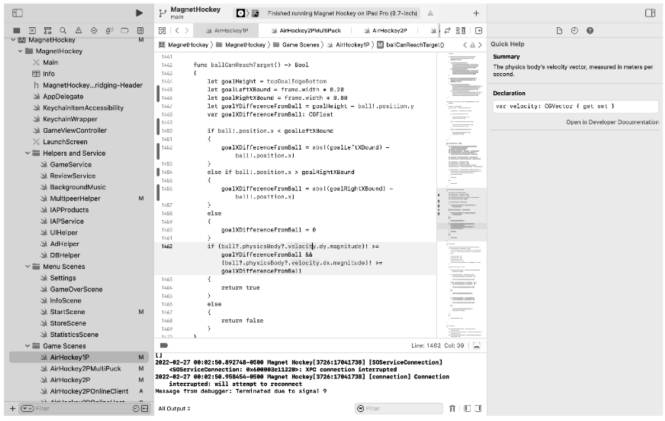

Figure 1
Apple's Development Environment Called Xcode

SpriteKit is a framework that can be imported within Xcode to make creating iOS game applications easier for a developer. Swift is a programming language that Apple designed that gives developers the ability to write logic and instructions to create an application. The Swift programming language defines the rules for writing code. Code in Swift is used to access the different functionalities offered by the SpriteKit framework, while working in the Xcode environment to create iOS applications.

Magnet Hockey is the title of the iOS game application I developed that has two game options for a player to choose from; each of which has its own set of game types. The Magnet Hockey app has a Magnet Hockey game option that includes two-player game types named "Standard Mode" and "Repulsion Mode." The objectives of the Magnet Hockey game "Standard Mode" are to avoid bumping into two or more magnets and to score the ball in the opponent's goal. In the "Repulsion Mode" game type, players can repel magnets away from them, if the magnets are travelling at a slow speed. In this game type, magnets moving at faster speeds will still attract towards the player. In the "Standard Mode" game type, the players do not have this "repulsion" property, and the magnets will always attract towards the player regardless of speed, so long as the magnets are within a defined proximity to the player. The other game mode option in the Magnet Hockey app is Air Hockey. The Air Hockey game option has one-player and two-player game types. The only way to earn a point in Air Hockey is to strike the puck into the opponent's goal; there are no magnets for the players to avoid. A player can challenge a robot (or a CPU opponent) in the one-player Air Hockey

game option. This robot uses algorithms to respond to different game scenarios to behave like a human player. In the two-player game option, a player can face a friend and choose between playing with one puck or two pucks. In both the Magnet Hockey and Air Hockey game modes, the players' sides are separated by the half line that is placed at exactly half of the device's screen height.

In the Magnet Hockey game modes, each player controls a mallet (like in Air Hockey) to strike a ball away from the circular goal on their side. One way to score is to strike the ball into the opposing player's goal. In the middle, at the intersection of both players' sides, there are three magnets spaced evenly across the width of the screen. These magnets move around the screen when the ball collides with them. The magnets utilize spring force physics to cling towards a player, who is within a defined proximity. If an opposing player's mallet contacts two or more of these magnets, then the player will earn a point. One objective of this game is to avoid colliding with the magnets. Whenever a point is scored by either player, the ball, the magnets, and both players' mallets are reset to their starting position for a new round.

The Magnet Hockey app's interface includes an info scene, a settings scene, a store scene, a statistics scene, a pause menu, an interactive tutorial, and more. Scenes are different views within the application. By using scenes or multiple views, the app does not always have to process everything. For instance, the main menu and store scenes are not loaded while playing in one of the Air Hockey or Magnet Hockey game scenes. The main menu shown in Figure 2 permits the user to navigate the various scenes throughout the app. The info scene describes the Magnet Hockey app. In the settings scene, a user can change game preferences. In the store scene, a user has the option to buy in-app purchases to improve their experience in the Magnet Hockey app. The user can see a history of their games played in the statistics scene. The pause menu is present when the user presses the pause button that appears on the screen during games. From the pause menu, the player can opt to view the interactive tutorial that will highlight game objects and explain what they do.

Figure 2: Main Menu for Magnet Hockey

Application Development

Scalable and Sustainable Development Practices

Responsive and intuitive design was a big focal point during the development of Magnet Hockey. A button to return to the main menu appears in various scenes. The designs for the info, store, settings, and statistics scenes are thematic. Each of the listed scenes has its own icon representing it. While in a scene, the icon is placed towards the top of the screen inside of

a red circle that matches the theme of other buttons in the menu. Additionally, each scene has an emitter in the background that uses this icon. Emitters continuously spawn the scene's icon in the form of small particles. As a result of these design choices, every one of these scenes is uniquely identifiable, but they are each intuitive to navigate, if the user has previously navigated any of the other scenes. All buttons in Magnet Hockey are programmed so that the user can visually tell if they have pressed down on a button, but the button's functionality is triggered only if the user releases their finger inside the button. This logic is handled inside of the touchesBegan() and touchesEnded() functions. These functions are included with the SpriteKit framework. A function is a block of code that can be used repeatedly. These functions help the developer to programmatically determine where on the screen a touch begins or ends. In programming, variables are used to store information for later reference. There are different types of variables depending on the information that is being stored. Variables of type Boolean can store values of true or false. A Boolean indicating that the user has touched a button is set to true, and the button's opacity is reduced when a touch begins inside of a button. If a touch ends in a button, that Boolean must be true, indicating that it started in the button to cause functionality. Otherwise, the button's opacity is returned to its untouched state, and the Boolean is reset to be false. A user must both start and end a touch inside of a button for it to work. This gives an extra level of confirmation to the user for each touch.

Responsive and adaptive design is important so that an app can run consistently on different devices. In the iOS ecosystem, devices have varying screen sizes, aspect ratios, and constraints. Aspect ratio is the measure of width relative to height for a screen. Art assets often need an iPhone version and an iPad version to avoid appearing warped. This is because an iPad is relatively wider than an iPhone. When an art asset does not have a horizontal and vertical line of symmetry, having separate art assets is usually necessary. Modern iPhones and iPads have safe areas around the notch where the camera is and at the bottom of the device where the home button used to be. These devices need to have different borders to restrict users from interacting with that portion of the screen. To make the design more interesting, information is sometimes placed in this safe zone for newer devices. For instance, each time a player collides with a magnet in the Magnet Hockey game mode, an indicator is placed in the top corner of their side (as seen in Figure 3). This magnet collision indicator cannot be interacted with; therefore, it is a good candidate to take up space in the safe area.

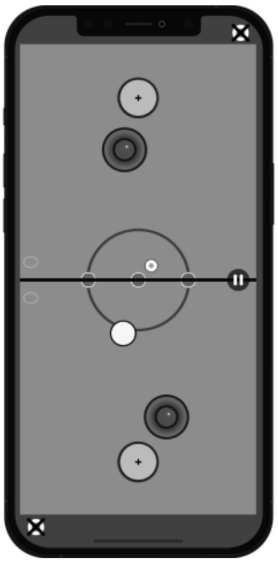

Figure 3: Magnet Hockey Game Mode

Apple is continuously introducing new devices to its ecosystem. App maintenance can be time consuming and tricky. It is important for a developer to program an app in a way that makes maintaining and updating it to support these new devices more sustainable. A common programming paradigm that is used to do this is Object-Oriented Programming or OOP. OOP is used to create blueprints of different objects in software. In Magnet Hockey, the three magnets share many of the same attributes. A blueprint of a magnet (or a class) can be used to quickly create these three magnet objects, rather than programming all the attributes of the three magnets individually. When the three magnets need to be altered in some way, the blueprint can instead be modified to alter all three magnets at once. This OOP paradigm is used throughout the Magnet Hockey app, causing there to be less duplicate code and making the code more readable and maintainable. Modular programming is used in conjunction with OOP. The process of modular programming breaks down a program or application into different smaller parts. These parts are usually distinguishable by functionality. Modules are abstracted from the main application, making the application easier to maintain. Modular programming is particularly useful for preventing duplicate code when implementing the same functionality within multiple areas of an app. The code for creating, writing, and reading from the database is abstracted from the scene in which it is being used, so that these actions can be executed from any scene in the Magnet Hockey app with only one implementation. Many other Magnet Hockey features follow a similar pattern.

In-Game Pause Menu

The player has the option to stop a game at any moment with the pause button on-screen. When this button is pressed, the pause menu appears. The player can navigate back to the menu, turn off the sound effects, or enter the tutorial mode from the pause menu. Players do not like to feel restricted or trapped within a certain part of an app. For instance, if a player starts a game with settings that they do not prefer, they may feel frustrated if they are forced to play through the entire game before being able to change the game settings. Giving players the option to change settings in-game or to return to the menu eliminates this issue.

When the pause button is pressed, the game objects' velocities are saved to temporary variables. The velocities for these objects are then set to zero vectors. Multi-touch is disabled when the pause menu is activated. This prevents glitches where the user could select multiple options at once, like resuming the game and launching the tutorial mode. If the player chooses to return to play, then the game objects' vectors will be restored back to what they were before the game was paused. Multi-touch is enabled again, so that both players can individually control their mallets with separate touches. The pause button and its functionality are designed to be unobtrusive to the player's experience. The pause button remains discreetly along the center line and is blended with the game, so that it does not block any game objects that pass underneath. This same button is used to resume the game while the game is paused. The icon switches from a pause image to a play image.

Magnet Hockey Game Tutorial

To inform players of the rules of the Magnet Hockey game option, a tutorial was created. This tutorial appears as an overlay on top of the Magnet Hockey game mode when the user starts their first game. The tutorial is also accessible through the pause menu at any time. The goals, the player's mallets, the magnets, and the score are all highlighted step by step to explain how to play (seen in Figure 4). Several functions were made to create a pulsing glow effect around highlighted game objects. The glow is created as a path around a game object. Functions can be designed to accept parameters or information that influence the effect or output. The path's size is calculated by taking in a game object as a parameter and making the path for the glow slightly larger than the radius of the game object. To accept different types of game objects, the create glow function is overloaded. An overloaded function is a function that is defined more than one time, but with different parameters in each definition to make the function more flexible. Another function creates a pulsing effect for the glow. This function is called in the update function. In SpriteKit, the update function is called at every frame or time unit. When called in the update function, this flashing glow function accepts a glow path as a parameter and then ranges up and down between two specified widths to create a pulsing effect.

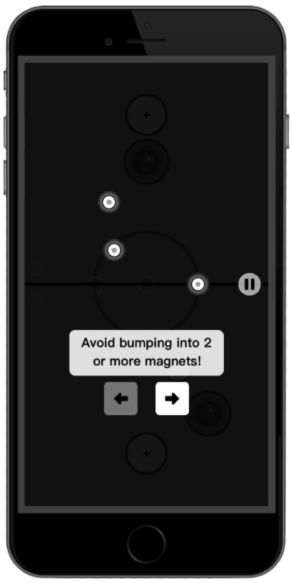

Figure 4: Tutorial Interface

The glows are hidden and do not flash when the user is not on a step for the specific game object being explained. A black image is placed to cover the entire scene except for the current tutorial objects being highlighted. As seen in Figure 4, the opacity of this black image is reduced, so that the user can still see their game in the background. Another invisible image is placed on top of the current game objects to prevent the player from moving game objects in the tutorial mode. The tutorial interface is placed above the invisible image so that the player can progress through the steps of the tutorial. The z-positions of the game objects, or the coordinate position used to determine how game objects are layered on top of each other, are manipulated for each step of the tutorial. If an object is being highlighted by the current step, then its z-position will be changed to be an integer that is between the z-positions of the black background and the invisible image that blocks touches.

The tutorial interface is strategically positioned on the screen for each step of the tutorial. When highlighting the magnets, the interface moves depending on the locations of the magnets. If most of the magnets are on one player's side, the instructions will move to the opposite side. Logic is in place for this dynamic positioning to work, even if one or two of the magnets have already been collected when the tutorial is started. Similarly for highlighting the player's mallets, the interface moves to the best location on screen to prevent the tutorial's description or buttons from covering either of them.

Mobile Revenue Models

To profit from app development, a developer can pursue a revenue model to extract value from other people enjoying their app. In recent years, the gaming industry has been progressing towards a free-to-play model. This model initially gained traction on mobile app stores before later spreading to other platforms, due to the success of games like Fortnite. With a free-to-play model, an enormous number of people can have access to an app or game with little to no barriers to play. Advertisements and in-app purchases (or microtransactions) are commonly incorporated to make free-to-play games sustainable to build and maintain. If a player enjoys a game, it may be tempting for them to pay a small amount of money from time to time to enhance their experience with new content. This model incentivizes developers to focus on supporting as many platforms as possible. It also incentivizes developers to maintain their app and release new content on a regular basis. If executed properly, this model is a win-win for the user, the developer, and the community.

Advertisements

The Magnet Hockey app implements Google's AdMob framework to deliver advertisements to users while in the menu and in between games. Ad units are registered on the AdMob website. Magnet Hockey uses banner ads and interstitial ads. Banner ads are rectangular and appear at the bottom of different scenes in the menu. Interstitial ads are full screen ads that appear when the game-over condition is met in the Air Hockey or Magnet Hockey game modes. Interstitial ads are called before the game-over scene is loaded. When the interstitial ad is closed, the transition to the game-over scene occurs. A developer can change preferences for the types of ads being delivered on the AdMob website. This is helpful for tailoring ads to your users based on the genre of the application. Additionally, AdMob uses the user's device ID to track activity across other apps to further personalize ads. The developer is compensated through the AdMob web platform when users click on the ads inside of the app. Having ads that are relevant to your app's typical userbase can help drive engagement.

In-App Purchases

Magnet Hockey also incorporates a store into its menu's interface. The user may be tempted to pay to remove all advertisements from the game or to unlock additional colors to change the way that the ball or puck looks. By using the payment transaction observer in Apple's StoreKit framework, it is possible to access a user's payment queue. By iterating the user's payment queue, the developer can act on different states of a specific transaction. The transaction's product can be accessed by the product identifier. Apple will handle

the interface so that the user can pay. If the state of the transaction is in the case of being successfully restored or purchased, then the developer should create a receipt to persist this purchase. Fortunately, Apple will prevent a user from paying for the same in-app purchase more than once. Products must be registered and approved online on Apple's developer platform. Each in-app purchase has a unique identifier. If a purchase is attempted by a matching Apple ID and product identifier, no charges will occur. It is important to include a restore purchases button, as per Apple's guidelines for in-app purchases. This ensures that users have a clear option to recover any lost in-app purchases. An app may be rejected by Apple if it does not have this option.

Data Persistence

Application data can be persisted in a variety of different ways. The method a developer chooses depends on the type of data, how secure the given data should be, and how long the data should be kept. UserDefaults, Keychain, and SQLite were each utilized for different purposes in the Magnet Hockey app. An SQLite database was implemented to store game history and statistics for the different game modes. Basic game preferences, like the number of rounds a player wants to play, and whether sound effects are enabled or not, are stored in the UserDefaults property list file. Secure receipts for in-app purchases are stored with Keychain.

Data Persistence with SQLite Database

An SQLite database is useful for storing small or large amounts of related data or lists for as long as the app remains installed on the user's device (Panchal, 2019). A local SQLite database is created when a person downloads the Magnet Hockey app. This database has four tables. One table includes a game history for all game modes, while the other three tables correspond with the following individual game modes: Magnet Hockey, 1-Player Air Hockey, and 2-Player Air Hockey. When a player finishes a game of either the Magnet Hockey game mode or one of the Air Hockey game modes, the database is opened. A new row is then inserted into the table with all game modes. A new row is also inserted into the table that corresponds to the game the user just played. The score for the top player, the score for the bottom player, and the ways in which points were scored (magnet collision or ball/puck in goal) for the top and bottom player are represented by four different columns in each row. After the game data have been inserted, the database is closed, and the game-over scene is loaded.

In the main menu, there is an icon that can bring the user to the statistics scene, when it is pressed. In the statistics scene, the user can choose which data from the database they would like to see. To see results, the user selects one of the four tables and chooses whether they want to see a history of games played or an overview of the table. An overview of the table will display to the user how many games have been played, how many games the top player has won, and how many games the bottom player has won. To retrieve this information, the appropriate table is iterated to append all row contents into a two-dimensional array. An array is a collection of elements. A two-dimensional array is a collection of arrays. This iteration process is done inside of the database helper file. This array is shared between the database helper file and statistics scene

file. Then, the array is iterated in the statistics scene. During each iteration loop (corresponds to each row in the database table), if the bottom score is greater than the top score for a given row, then the number of games the bottom player has won is incremented and vice versa for the top player. The number of games played is the number of iterations in the outer loop or the number of rows in the table.

The process is a little more complicated to visualize the results for the game history in the statistics scene. Five results are shown at a time, starting with the most recent. Buttons are placed on the screen to allow the user to go back and forth between pages to look through all the results. Above the icons responsible for changing pages, there is text that indicates which results are currently being displayed. The number of pages of results needs to be calculated, so that it can be dynamic with the number of games played. The logic for the buttons to switch between pages depends on this. The results query should work if there are zero results or 1000 results, and everything above and between. Modulus is a math operator in programming that gives the integer remainder for division. To calculate the number of pages of results, the total number of games or results modulus the number of results shown per page (five per page), is computed. If the modulus is equal to zero, then the number of pages is equal to the number of results divided by the number of results shown per page. If the modulus is not equal to zero, then the number of pages is equal to the number of results divided by the number of results per page, plus one.

Having the number of result pages makes it feasible to calculate the text that indicates which results are showing. Each time a page is displayed, it is determined whether the user is viewing the last page of results, based on the number of pages. If the user is on the last page, then the text will display the total number of results as the second number in the range. The first number in the range will be calculated by subtracting four (the number of results per page, minus one) from the product of the current page number and the number of results per page. In Figure 5, there are thirty-four results; therefore, the last results will be shown on page seven with a label "31 to 34" ('seven times five, minus four' to 'number of results').

Figure 5: Game History with SQLite Database

Formatting the order for which points were scored was the biggest challenge with visualizing the database in the statistics scene. The point order is stored in the database as an integer composed of ones and twos. A one digit represents a point scored by a goal, while a two digit represents a point scored by magnets. To get the point order, an algorithm calculates the point order integer modulus two. If this result

is equal to zero, then that means the last digit in the integer corresponds to a point caused by collision with magnets. If that result is not equal to zero, then the last digit in the integer corresponds to a point scored by a goal. After this is determined, a digit is removed from the end of the point order integer by dividing it by ten and then taking the floor value. This process is done repeatedly for each side's score (top and bottom), and for each result being displayed.

Because there is an option to change the number of rounds in each game in the settings scene, the number of icons displayed for each result's point order can vary from two icons to thirteen icons. Copies of a goal sprite and a magnet sprite are created as needed for each page. When the user changes the result page or returns to the menu, all the sprites are removed from memory. This approach ensures that only necessary sprites are saved in memory at a given time. Since the image, the goal sprite, and the magnet sprite use are parts of the statistics scene sprite sheet, creating and drawing new sprites on demand and using these images are very quick and efficient. Sprite sheets strategically combine images that are accessed simultaneously into a single larger image to reduce loading times and improve the app's performance.

Data Persistence with UserDefaults

UserDefaults is useful for storing preferences that do not need to be protected with encryption. UserDefaults is a property list file that is structured like a dictionary (Apple, n.d.). A dictionary is a data structure, where a given value can be accessed with a key value. When adding data to UserDefaults, it is necessary to have both a key and a value. In UserDefaults, information is stored in plaintext and can easily be manipulated, if a person uses computer software to access the contents of the app's folder. Basic game settings, if sound effects are enabled or not, are recorded into UserDefaults. It is not necessary for these preferences to be secure or for them to persist, if the application is deleted. Additionally, because these preferences only have one value at a time and have little relation to anything else, an SQLite database would not be appropriate to store them.

Data Persistence with Keychain

Securing data on an iOS device is possible with Keychain. Each iOS device has its own local and unique Keychain. Using the Keychain services API, developers can encrypt users' data (Apple, n.d.). Third-party wrappers can be implemented to make Keychain implementation very easy. 'KeychainWrapper' is a popular wrapper that makes adding data to the Keychain very similar to adding data to UserDefaults. With UserDefaults, the data type, key, and value are all similarly set in one line of code (Jason, 2022).

In Magnet Hockey, the preference for ball color is securely stored with Keychain, because it is tied to an in-app purchase that gives the player additional ball colors from which to choose. To prevent the player from modifying the color manually by editing the UserDefaults property list file, these are given an extra layer of protection. For a developer of a free app, it is important to not make it easy for people to steal the content that brings revenue. The encryption for Keychain is handled by the wrapper and the Keychain API (Sipila, 2017).

When the developer needs to retrieve these data for the user, they can simply do so by accessing the key name that was previously set. In Magnet Hockey, the receipt, stored as a Boolean, for purchasing to remove ads, was stored in Keychain. The presence of this receipt is verified in the app before loading and displaying advertisements. One advantage of using Keychain is that the data are persisted even if the app is deleted. While this may not be necessary for most preferences, it is helpful to prevent in-app purchases from needing to be restored manually. If the in-app purchase to remove advertisements was stored with UserDefaults, the advertisements would appear, if the app was deleted and reinstalled. This is because the UserDefaults property list file is removed when the app is deleted.

Review Request

Review request urges users to rate and review an application. Reviews and ratings help give an app exposure to more people on the app store. It is important to request a user to review an app at favorable times to prevent poor reviews. In the Magnet Hockey app, the user is prompted to rate the app in the game-over scene, after they have completed five games of any game mode. An alert view appears on screen, asking the user if they are having fun. If the user answers "No," then they are not prompted to rate the app. This response is also saved in UserDefaults to prevent the user from ever being prompted to review or rate the app again. If the user answers "Yes," then the prompt to rate the Magnet Hockey app appears as another alert view.

Apple permits developers to prompt a user to rate an app three times within a one-year period. Apple will not initiate the alert for rating an app if the user has already left a rating. In Magnet Hockey, on the second and third rating requests, the user may be prompted with Apple's rating alert view, if they have not left a rating the first time, and if they answered "Yes" to having fun playing the app during the first request. This second and third request is initiated, when a user finishes a game seven days or more after last being prompted to review. This time, the calculation is done by using time stamps stored in UserDefaults (Loco, 2019). Since the developer cannot determine if the user has already left a rating or review, the prompt asking whether the player is having fun will not appear on the second and third requests.

Player Retention and Challenges

Creating Bot Mode

Having a mode for a user to play without a friend is important for retention. Developing a one-player mode for the Magnet Hockey game mode proved to be a challenge. There were many unforeseen cases. Coding an algorithm that solves each case without breaking another case was very difficult and unrealistic without a machine-learning implementation. The bot must consider avoiding magnets, defending a goal that can be attacked from all directions, and striking a ball that may have a goal or magnets in its path. Avoiding all three magnets was challenging on its own. The bot must consider the speed, direction, and slope of the magnet's velocity. The bot must also consider its relative position to the magnet (above or below, to the left, or to the right) to determine which way it should navigate. Creating a system that prioritizes the different bot objectives depends on many different circumstances and is complicated. For instance, should

a bot collide with a magnet to reach the ball in time to save a shot on target? The answer to this question might depend on whether the bot already has picked up one magnet.

Two-Player Air Hockey Game Mode

An Air Hockey mode makes creating a one-player bot mode less intensive. Air Hockey is also a more familiar experience to lure new people into downloading the app (Schilling, 2005). People are more apt to embrace something if it has a connection to something, with which they are already comfortable (Thompson, 2018). The Air Hockey game option was first developed for two-Players. To create this, the circular goals and magnets from the Magnet Hockey game mode were removed. Then a goal was placed flush against the top and bottom borders of the screen. Air Hockey will need two player mallets (like Magnet Hockey), a puck, and one goal on each side. Semi-circles were attached to the goals on each side for design. Also, there is a new full circle in the middle that previously went carefully around the magnets in Magnet Hockey. To score in Air Hockey, a player needs to strike the puck into the opposing player's goal. A two-puck game mode was also added as an option for two-player Air Hockey. This two-puck game mode has new scoring and collision logic. For a round to end, both pucks must go into a goal, unless a player's score has already reached the game winning score when the first puck was scored. Additionally, there is collision detection added for the second puck, and when two pucks collide with each other.

One-Player Air Hockey Algorithm

Work began on the one-player Air Hockey bot mode once the two-player Air Hockey mode was fully completed. The algorithm for the bot was split into three separate functions or objectives: defend, attack, and mirror. These functions hold the logic for how the bot should behave. The mirror function dictates how the bot should position itself when the puck is on the player's side. The defend function is active when the puck is on the bot's side, and the puck's y-position is greater than the bot's y-position. The defend function is also active when the puck is moving on target to score a goal. The attack function is active when the bot is appropriately positioned to strike the puck towards the player's goal, and the puck is on the bot's side.

When the puck is in the player's half, the mirror function is acting on the bot's velocity to manipulate its position. The bot moves left and right, with the puck to follow it. If the puck is to the left or to the right of the goal frame, the bot waits at the goal post, on the side to which the puck is closest. This is done so that the bot is well-positioned to defend the goal. While the mirror mode is active, the bot retreats to a y-position near its goal frame, if it is not already there.

Linear algebra is used to aid the behavior of the bot for defending. A function named "ballIsOnTarget()" returns a Boolean, informing whether the puck is moving in the direction of the goal. This is calculated by using the slope of the puck's velocity vector to create a semi-infinite ray. If this ray crosses paths within the bot's goal frame, then the shot is on target, and the function will return true. Additionally, the magnitude or length of the puck's velocity vector is checked to determine if the puck will reach the goal. If the mag-

nitude is greater than or equal to the distance between the bot's goal and the puck, then the puck can reach the goal. If the puck can reach the goal, and the puck is on target, then the puck is considered dangerous by the bot, and the bot should defend. As pictured in Figure 6, the bot is programmed to move towards the puck's expected x-position (coordinate), when it reaches the goal frame. If the bot can get to this position on time (before the puck goes into the goal), then the bot will enter attack mode.

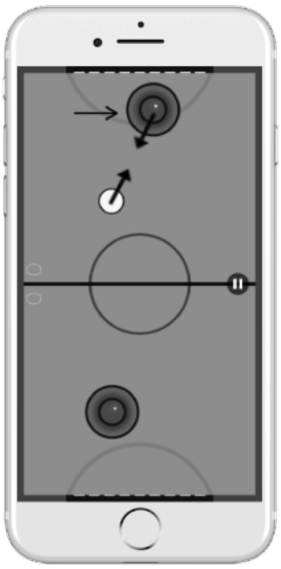

Figure 6: Explaining Bot Algorithm

The bot attempts to strike the puck, when the attack mode Boolean is true. In the instance, where the bot is located, at the point in the goal frame at which the puck is expected to travel, the bot will begin moving towards the puck. Since the bot is already on the puck's path, the bot can multiply the puck's velocity slope by negative one to travel towards the puck and strike it.

Conclusion

Magnet Hockey was designed to be simplistic yet intuitive. Linear algebra and physics are applied to create the Magnet Hockey and Air Hockey game modes. Spring force, collision detection, collision response, and vector math were a few of the topics explored. The Magnet Hockey app also integrates three different types of data persistence: UserDefaults, Keychain, and SQLite database. Each of these methods of handling data fit different use cases. Keychain is good for securing preferences, SQLite is good for handling large lists of related data, and UserDefaults is good for efficiently storing preferences that do not need to be secured. Keychain is securely helpful for persisting in-app purchases. In-app purchases and advertisements are two ways that Magnet Hockey monetizes its free-to-play model. To instill trust in the userbase of the Magnet Hockey app, emphasis was placed on designing the game and menu to be thematic and attractive. The menu and game modes always have a means of navigating from scene to scene to prevent a feeling of entrapment for the player.

By creating Magnet Hockey, I experienced many common mobile application development challenges. It is important to have a careful plan to support new devices more feasibly. Creating a responsive and intuitive user experience can be tricky and time-consuming. This is important for player retention. The Magnet Hockey app is still being developed and improved. Online game options are currently being added to connect players around the world. The Magnet Hockey app has been downloaded over 150 times in 44 different countries.

References

Apple. (n.d.). *UserDefaults*. Apple Developer Documentation. Retrieved September 9, 2021 from https://developer.apple.com/documentation/foundation/userdefaults Apple. (n.d.). Keychain Services. Retrieved December 14, 2020, from https://developer.apple.com/documentation/security/keychain_services

Jason. (2022, May 12). *A simple wrapper for the IOS keychain to allow you to use it in a similar fashion to user defaults. written in swift.* Swiftobc. Retrieved November 15, 2021, from https://swiftobc.com/repo/jrendel-SwiftKeychainWrapper-swift-security

Loco, K. (2019, April 29). *Requesting a review for your app*. YouTube. Retrieved October 5,\2021, from https://www.youtube.com/watch?v=kMK-8m2P4Cec

Panchal, K. (2019, February 13). *Everything you need to know about sqlite mobile database*. Our Code World. Retrieved November 28, 2021, from https://ourcodeworld.com/articles/read/737/everything-you-need-to-know-about-sqlite-mobile-database

Schilling, M. A. (2005). *Strategic management of technological innovation*. McGraw Hill Education.

Sipila, J. (2017, February 25). *Securing user data with Keychain for iOS*. Retrieved December 14, 2020, from https://medium.com/ios-os-x-development/securing-user-data-with-keychain-for-ios-e720e0f9a8e2

Thompson, D. (2018, May 8). *The four-letter code to selling anything*. YouTube. Retrieved June 25, 2021, from https://www.youtube.com/watch?v=-6pY7EjqD3QA

About the Author

Trevor D. Wysong is pursuing his Master of Science in Computer Science at Bridgewater State University. His research project began in 2020. This research project is still in progress during spring 2022 under the mentorship of Dr. Enping Li. Trevor's graduate studies and research are funded by the GAANN Fellowship. Trevor began work as a software engineer in June 2022.