COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

CSCI 370 Final Report

Hexagonal Chess

Hanuman Chu
Jacob Hulvey
Dawson Matthews
Nate Webster

Revised June 14, 2024

CSCI 370 Summer 2024

Advisor Tree Lindemann-Michael

Table 1: Revision history

Revision	Date	Comments
New	5/13/2024	Started Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done References Appendix A - Key Terms
Rev – 2	5/17/2024	Completed Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done References
Rev – 3	5/24/2024	Completed Sections: VI. System Architecture XIII. Team Profile Updated Sections: References Appendix A - Key Terms
Rev – 4	5/31/2024	Started Sections: VII. Software Test and Quality Completed Sections: VIII. Project Ethical Considerations Updated Sections: VI. System Architecture
Rev – 5	6/8/2024	Started Sections: IX. Project Completion Status X. Future Work XI. Lessons Learned Completed Sections: VII. Software Test and Quality Updated Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done VI. System Architecture VIII. Project Ethical Considerations Appendix A - Key Terms
Rev – 6	6/14/24	Completed Sections: XII. Acknowledgments Updated Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done VI. System Architecture VII. Software Test and Quality VIII. Project Ethical Considerations

		IX. Project Completion Status X. Future Work XI. Lessons Learned References Appendix A - Key Terms

Table of Contents

I. Introduction

Our overall goal for this project was to create a hexagonal chess Windows application that can be played against other players or against a trained artificial intelligence (AI). The game variation we will be using is Gliński's hexagonal chess [1], which uses a hexagonal board with hexagonal tiles. The board in this variation is set up differently than regular chess with pieces that are all altered to move slightly differently, but ultimately follow the same rules, just on a hexagonal board.

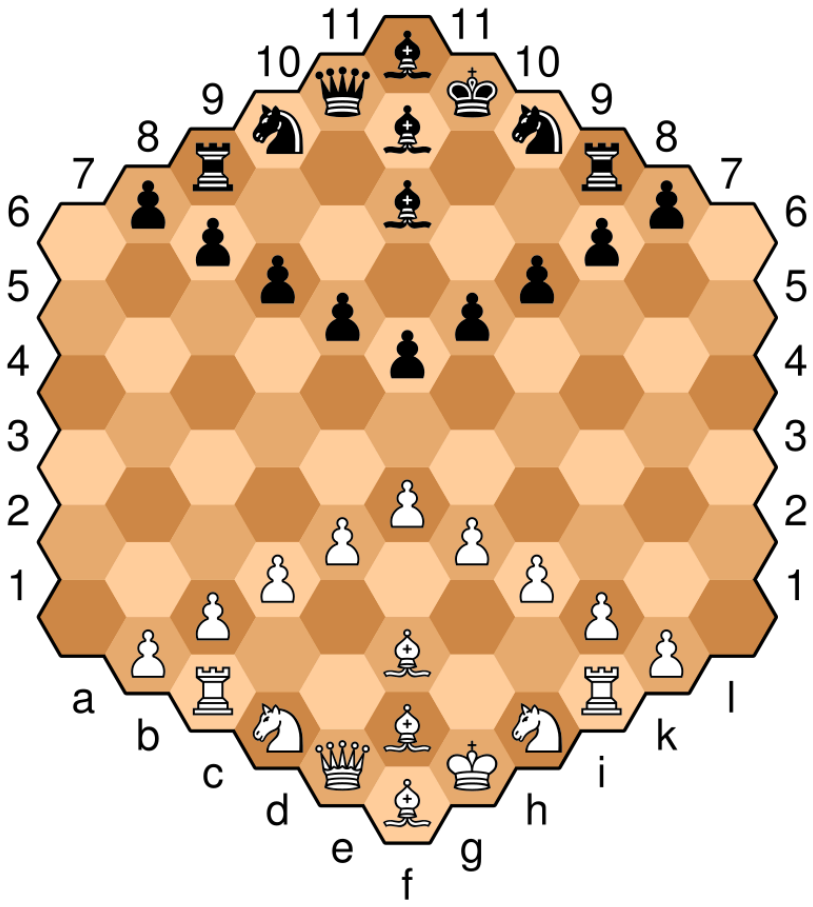

Figure 1: Gliński's Hexagonal Chess Board [1]

As our project is a student project, our client and project proposer is Hanuman Chu, a member of our group. The stakeholders and users of the software are any future players or developers of the game. These players should not need to know anything about hexagonal chess (they do need to know the basics of normal chess), and thus the user interface should be descriptive, intuitive, and consistent to teach players the rules and to not confuse them while they are playing.

II. Functional Requirements

The functional requirements necessary for this project are:

- Two game modes: solo play against an AI trained with reinforcement learning and pass-and-play multiplayer.
- A user interface that displays the game board and any valid moves a piece can make, alongside a rules encyclopedia to teach and remind new players of the game's rules.

Some stretch goals are:

- A main menu and pause menu.
- A turn timer that introduces a time element that is present in many chess games, which also restricts the amount of time the AI has to make decisions.
- Music in the main menu and while playing.
- Porting the game to Linux.
- Difficulty options for singleplayer mode.
- A settings menu that can alter the user interface to conform with each user's desired experience.
- The loading and saving of games so that a user can pick up or put down a game at any time.

Many of the features involving the user interface require a solid and working backend implementation of the game to properly function. If anything is communicated to the player incorrectly, it could result in an unplayable game or an unreliable experience.

III. Non-Functional Requirements

The non-functional requirements necessary for this project are:

- The AI should be able to beat a new player while taking less than one second per move.
- The AI should be comparable to online AIs while taking a reasonable amount of time per move.
- The Graphical User Interface (GUI) should have aesthetically pleasing graphics, and it should be able to convey the necessary information to learn and play the game.
- The game should not crash the average computer and the game window should be able to run for a "significant" amount of time without crashing.

IV. Risks

Table 2: Risks

Risk	Likelihood	Impact	Mitigation Plan
Slow AI	Very Likely	Moderate	Begin programming AI from sprint one and plan to be able to begin training AI's neural network from sprint two or three.
Loss Prone AI	Likely	Major	Begin programming AI from sprint one and plan to be able to begin training AI's neural network from sprint two or three. Train the AI using Mines computing resources on campus. Be ready to use an Alpha-Beta search algorithm in place of a neural network-based AI.
Not able to complete functionality	Moderate	Major	Create proper documentation so that any future programmers can easily pick up the project.

V. Definition of Done

Since we did not have a formal client, we defined our own minimum useful feature set to be a functioning singleplayer and local multiplayer mode for our game. The AI in the singleplayer mode must

be able to reasonably quickly (within 1 second) calculate its moves, and hold its own against a new player and against other online hexagonal chess implementations (specified in Section VII). We also tested if our singleplayer AI can win a majority of its games while playing other implementations.

When applicable, we used doctest unit tests to test our code. We tested our implementation by playing against it ourselves, since none of us are experts at hexagonal chess and by recruiting additional players to play. We also imputed its move decisions, determined after an allocated amount of time, into other implementations to test if it can beat other AIs. These tests consisted of multiple games (at least one time with both colors) against the same human or AI opponent.

The final product was delivered as well-commented C++ source code with a corresponding Windows executable that can run on the average modern Windows machine.

VI. System Architecture

The foundation of this system's architecture is a collection of code we call the "game" code which handles creating a game, calculating all potential moves from a game's state, and saving and loading the game (see Figure 2). The "game" code is wrapped in a wrapper that communicates with the GUI and a trained neural network to run the game application (see Figure 2). The AI is based on Deepmind's AlphaZero algorithm which uses a neural network and a modified version of Monte Carlo Tree Search (MCTS) made to work with an evaluation function (see Appendix A).

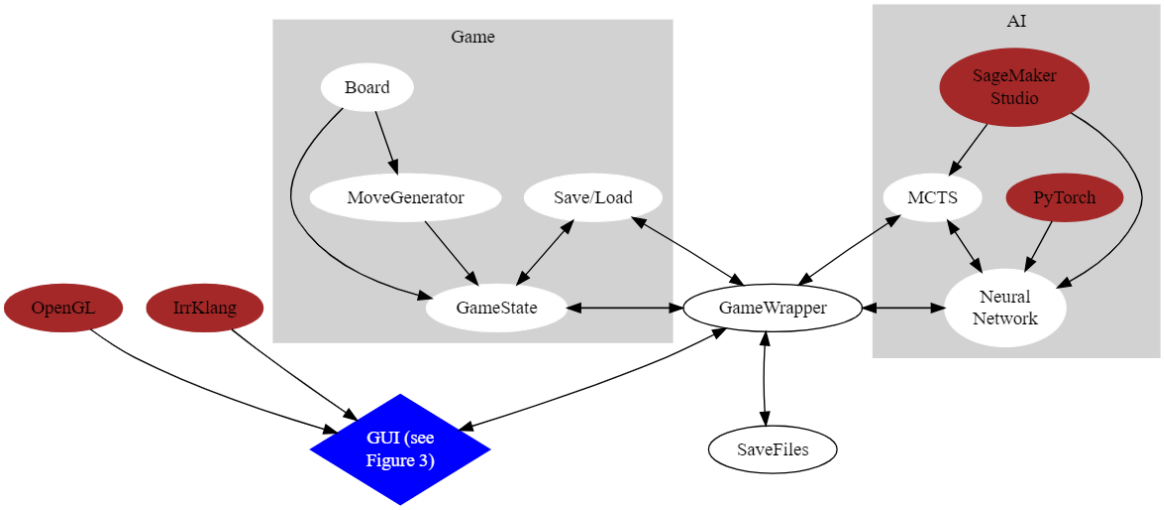

Figure 2: Abstracted Diagram of System Architecture not including Hex (LinkToFigure2)

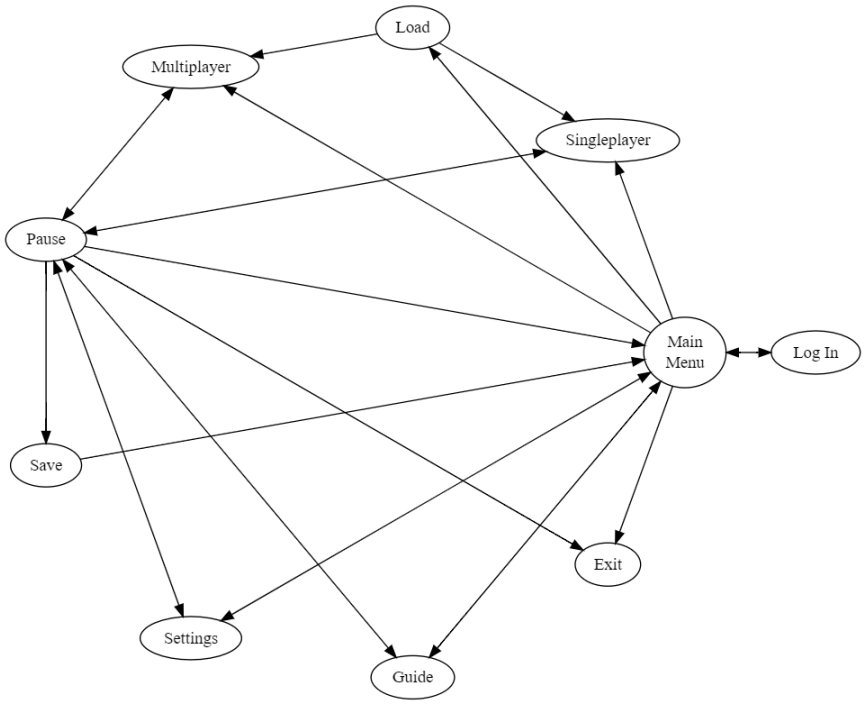

Figure 3: Complete GUI State Diagram (LinkToFigure3)

To create the system's architecture four 3rd party elements are needed: OpenGL, PyTorch, IrrKlang, and SageMaker Studio. These 3rd party elements are indicated in Figure 2 using the dark red color. We are using OpenGL as our graphics library for our GUI, PyTorch to model our neural network, IrrKlang to play music on our GUI, and SageMaker Studio to run MCTS to generate data for and train our neural network.

Hex is one of the index systems we are using to describe the board. The exact specification we are using is the QRS coordinate system which can be found in [2].

Board (see Figure 2) is a namespace that allows for efficient storing and changing of the board's state. It defines boards as unsigned integers with a length of 128 bits of which the first 91 bits are used to represent the 91 spaces on the board.

MoveGenerator (see Figure 2) generates the moves for all pieces on the board given the array of boards and the current "players" color.

GameState (see Figure 2) is a class that contains a snapshot of a game. For this purpose, it contains an array of eight boards; six of which hold the locations of pieces of a particular type and two which hold the locations of pieces of a particular color. GameState contains all the necessary functionality needed to play a game including determining whether the game is over, what moves can be made, the GameState that would result from a move, etc.

GameWrapper (see Figure 2) is a class that organizes and abstracts all other "game" files to make interfacing with the GUI easier. The GUI creates an instance of the GameWrapper on startup. The GameWrapper class implements functions like getMoves, playMove, getBoard, isGameOver, saveGame, loadGame, etc. These functions make it easier for the GUI to quickly render the board and communicate game changes with the "game" code.

The save files store all of the necessary information to be able to load games after closing the program. They hold the data for the state of the game in a Forsyth-Edwards-like notation (see Appendix A), modified for hexagonal chess. The stored data includes piece placement data by board rank using

individual characters representing piece type and color, the active color of the board, any en passant target squares, and the number of half-moves, which is used for the boring moves rule. Date-time information, user statistics, and game numbers/names are also included to help keep track of each game in progress.

Neural Network abstracts away interaction with a model including asking it to predict a GameState's move probabilities and value, training the model with new data and saving/loading the model from the disk.

MCTS uses a Neural Network to perform a modified version of Monte Carlo Tree Search and return a list of move probabilities for a given game state of which the highest is chosen to be the AI's move.

The GUI uses OpenGL to render our game to the screen. The GUI stores the model files for the hexagons, as well as the game pieces, alongside different textures for those models. The GUI can then take given hexagon locations from the Board namespace and translate each hexagon from its QRS coordinates to XYZ coordinates in the XZ plane. The GUI registers mouse clicks on the game pieces by using a color buffer, a multipass rendering technique. Before rendering the real scene, the GUI renders a version of the scene into a buffer in which each hexagon has a designated color ID that it is rendered in (see Figure 4). The program then checks which pixel of the buffer the mouse is hovering over, and reads the color of that pixel. The program maps that color to which hexagon it corresponds to know which hexagon tile was clicked. To play the game, the GUI essentially checks for clicks on game pieces, gets a list of all valid moves for that piece from the GameWrapper, displays those valid moves, waits for a valid move location to be clicked, and sends the move to the GameWrapper. The program utilizes the Phong Reflection lighting model with Gouraud shading, which applies ambient, diffuse, and specular lighting per vertex in the scene.

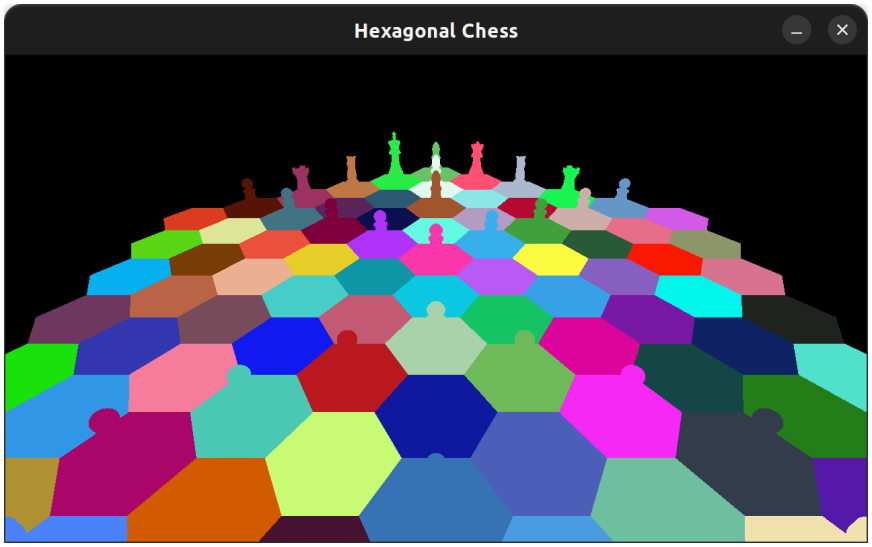

Figure 4: Hidden Color Buffer Visualization

VII. Software Test and Quality

We used six methods to assure the quality of our code: code review, refactoring code, commenting code, a readme file, unit tests, and manual testing of the GUI.

The primary way we ensured the quality of our code was by having a partner review our code. Each week two team members would form a pair and the pair was responsible for reviewing their partner's work to assure readability and reliability. Every week the partners would rotate. Partner-based

code review allowed errors, mistypes, and readability of the code to be caught because another set of eyes would look at said code.

Another method we used to ensure the quality of our code was by creating unit tests for our code. We used doctest unit tests to verify that our code accomplished its necessary functionality. Doing unit testing on our code allowed for edge cases to be found and corrected before the code was used by a user thus increasing the quality of our code.

After our code was functioning and passing all unit tests we returned to the code to add any helpful and descriptive comments and to refactor our code. Commenting and refactoring the code increased readability and helped find additional edge cases in the code that had not been thought of and tested. Commenting and refactoring thus increased the quality of our code.

A readme file was created to help future users of the code have an increased understanding of the code structure and what each code file does. The readme helped our team understand the code's structure and helped find untested edge cases and extra unnecessary code.

We did manual testing of the GUI to find errors. By having the team act as users trying to break the GUI and "game" code we found errors which led to fewer fixes needed later and by future programmers. This assured the intuitiveness of the GUI and led to a more satisfying user experience. By finding errors in the GUI and in the "game" code we improved the quality of our code.

Table 3: Penetration and Unit Testing Tests

Purpose of Test	Description	Tools Needed	Threshold of Acceptability	Edge Cases	Results
Check GameWrapper functionality	GameWrapper Unit Tests	doctest.h	All tests passing	Hex outside of range for playMove(); playPromotion() of non-pawn; Move piece with no moves; Called aiPlayMove() after game was over or on a multiplayer game	All Passed
Check GameState, Board, and MoveGenerator functionality	GameState Unit Tests	doctest.h	All tests passing	Piece movement/capture; Checkmate/stalemate conditions; Save/load string parsing	All Passed
Check Network Wrapper functionality	NetworkWrapper Unit Tests on different network parameters and via different computers	doctest.h, Mines network, multiple computers	All tests passing on a single computer on the Mines network, and on multiple computers on a less secure network	Network too slow; Network too secure	All Passed
Find GUI Errors	Manually try to break GUI	Time	No errors affecting functionality	Click buttons with auto-clicker; Try to crash the program by navigating fast; Small screen; Large screen	All Passed
Check AI works reasonably well and fast against a new player	Manually try to beat AI	New player	A new player does not get frustrated waiting and does not always win	N/A	All Passed: AI is fast enough for no frustration, AI has won against new players.

Check AI works reasonably well and fast against other hexagonal chess AIs	Play our AI against other hexagonal chess AIs	Steam Game [3], Online Game [4], GitHub Game [5]	Our AI wins 50% or more of the time.	N/A	Steam: won 1 out of 1 game (deterministic) Online: won 4 out of 4 games Github: won 2 out of 2 games (deterministic)

VIII. Project Ethical Considerations

The ACM and IEEE principles that were particularly important to the development of our project were ACM principle 2.1 and IEEE principles 2.06, 3.01, and 3.09.

ACM principle 2.1 and IEEE principle 3.01 regard the quality of our work and final product. We needed to take special consideration of these principles because a chess application is a relatively simple program which could have caused us to write low-quality code. The negative effect of these principles being violated is that future bug fixes and updates by ourselves or others will be harder than necessary.

IEEE principle 2.06 regards intellectual property laws. As our team created a GUI using models with three dimensions there was a potential that we use assets that are other's intellectual property. The negative effect of this principle being violated was that an asset creator might not receive respect and/or monetary compensation.

IEEE principle 3.09 regards realistic scheduling of obtaining certain quality and functionality in the final product. As our team was creating a neural network that required time to generate examples and train there was a possibility that the time would not be enough for an acceptable product. The negative effect of this principle being violated was that the product would not be finished in time even though we told the "client" and stakeholders that the product would be finished.

We applied Michael Davis's Reversibility Test to our decisions regarding which assets to use. If we were to use paid and/or non-commercially licensed assets (that we did not get legally), if we were in the shoes of the asset creator then we would feel wronged as we are not receiving proper monetary compensation or respect. Other options to using paid and/or non-commercially licensed assets, were to create our assets, find commercially licensed assets or we could have purchased the assets.

We additionally applied Michael Davis's Harm Test to our decision to use an AI instead of other less computation-heavy methods. We were potentially harming the environment by using an increased amount of energy due to our use of an inefficient method known as AI. Since this specification was in the requirements document provided by Hanuman Chu we were required to use an AI in the final product, but there were other options like just MCTS with a custom-built evaluation function or using the Alpha-Beta Search Algorithm.

IX. Project Completion Status

The minimum goal of this project was to create a Hexagonal Chess Windows application with a functioning multiplayer and singleplayer mode. The singleplayer mode's AI must also be reasonably fast and hold its own against a new player. We have completed the functioning singleplayer and multiplayer mode, however, the AlphaZero AI is not to our difficulty standards, though it can make semi-reasonable moves almost instantly. We began generating training data during the second sprint on multiple devices, but we did not generate an adequate amount of data. We thus had to fall back on our backup plan which is using Alpha Beta search. Regarding our functional requirements, we have implemented a working singleplayer against an AI, pass-and-play multiplayer on one computer, menu interfaces including a main

menu, a how-to-play page, a pawn promotions page, and game-over banners (see Figure 5). Regarding non-functional requirements, the AI can beat new players, and we believe the graphical user interface is pleasing and conveys all the necessary information to learn and play the game. Additionally, the game does not crash itself nor the computer it is running on for the average computer.

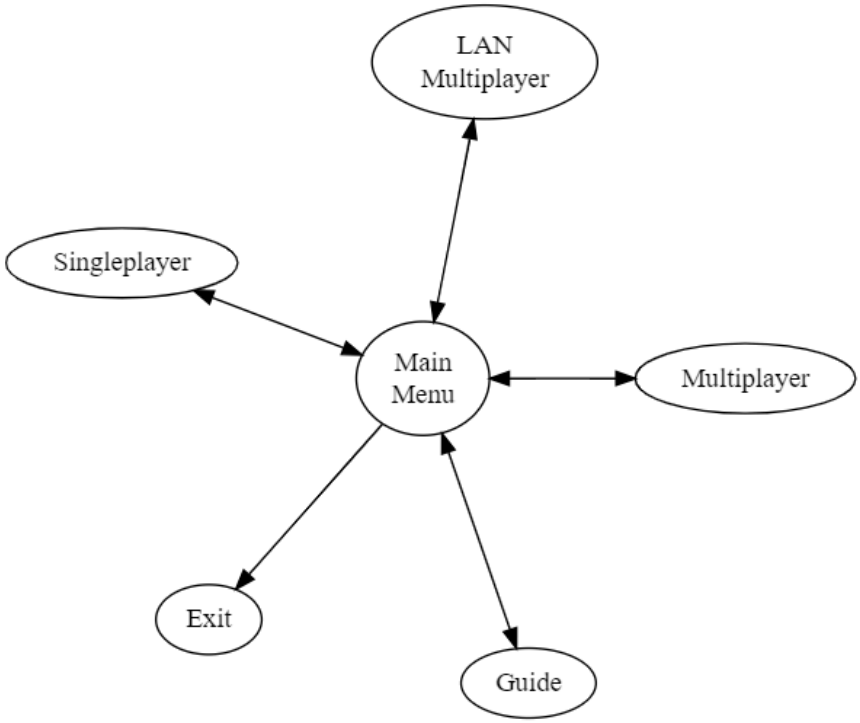

Figure 5: Actual Implementation of GUI (LinkToFigure5)

X. Future Work

Since we established a lot of stretch goals during the planning for our functional requirements, any future work can consist of completing stretch goals that were not started or fully realized. Some of the stretch goals that could be completed in the future include improving the AlphaZero AI so that it can defeat other online AIs, adding a pause menu, adding save and load functionality, adding a login page, porting LAN multiplayer to Linux, and adding a settings menu with various settings and user preferences. The save and load code has already been written, but it requires the users to log in with a yet-to-be-implemented GUI page.

We have left some artifacts in our code so that future developers have an easier time updating, improving, and debugging our code. We have heavily commented our code so that future developers can easily understand the code specifics and what each code file does. We have additionally created a readme file that will help future developers determine which files they need to modify to implement certain features and debugs. Lastly, there are our unit tests that have been written for the Board, MoveGenerator, GameState, and GameWrapper code which could allow for test-driven development for future developers.

XI. Lessons Learned

We learned that managing a GitHub with multiple people working on multiple different branches is difficult. Our first attempt at managing this was through a single development and review branch, but we quickly learned that it was not adequate for the vast and varied changes we were making daily, so we

ended up creating a branch for each new feature we wanted to add and then followed the following process:

- Write the code (including unit tests)
- Peer review the code
- Locally merge with main
- Check that main runs
- Push the changes to main

We learned about the differences between MinGW and Microsoft Visual C++ (MSVC) and how MinGW does not work with PyTorch on Windows. This difference required most of us to download Visual Studio and install MSVC for the first time and to spend several hours changing the auto usage of MinGW to MSVC.

We learned how to write unit tests in C++ using doctest. Most of this process follows the process taught in CSCI 306 (Software Engineering), although our team did not implement test-driven development due to our limited time to work on the project. Additionally, there were a few differences in the testing framework, but it was easy to pick up and learn since it follows a simple syntax.

We learned that training an AI to a usable level is difficult and time-consuming. We had to develop a mostly playable game before we could start the AI's training which left us with less time than we had hoped. Regardless, if we were able to start training on the first day of the project, we would have still lacked the computing resources needed to get the AI to a state that we would have liked.

We learned that creating a working program in five weeks with documentation, planning, a report, and multiple presentations is exceptionally difficult. A lot of the work was not focused on developing the actual project, especially near the end of the field session, but was instead focused on fulfilling these deliverables.

XII. Acknowledgments

Thank you to our advisor, Tree Lindemann-Michael, for his guidance and support throughout our project planning and development.

XIII. Team Profile

Hanuman Chu
Junior
Computer Science General Track
Hometown: New York, New York
Work Experience: Teaching Assistant for CSCI 200 in Summer/Fall 2023
Sports/Activities/Interests: Reading, Rationalizing, Reasoning, Relaxing, Reawakening

Jacob Hulvey
Senior
Electrical Engineering/Computer Science
Hometown: Lake Jackson, Texas
Work Experience: Asset Management Summer Intern 2022 for BASF in Freeport Texas; CM Electrical Engineering Summer Intern 2023 for BASF in Freeport Texas; Teaching Assistant for Embedded Systems in Fall 2023
Sports/Activities/Interests: Running, Reading, German Culture, Skiing

Dawson Matthews
Senior
Computer Science + Computer Engineering
Hometown: Malibu, California
Work Experience: Teaching Assistant for CSCI 101
Sports/Activities/Interests: Video Games, Skiing

Nate Webster
Senior
Computer Science + Data Science
Hometown: Fort Worth, Texas
Work Experience: Teaching Assistant for Introduction to the Linux Operating System
Sports/Activities/Interests: Hiking, Video Games, TTRPGs, Board Games

References

[1] Gliński's hexagonal chess setup, by László Németh, 23 Jul. 2013. Wikimedia Commons. [Online]. Available: https://commons.wikimedia.org/wiki/File:Hexagonal_chess.svg.
[2] A. Patel, "Hexagonal Grids," *Red Blob Games*, Mar. 2013. https://www.redblobgames.com/grids/hexagons/.
[3] KonsolKongen and JayJay, "Chex," Steam, published by Stuen, Jan. 21, 2023. [Online]. Available: https://store.steampowered.com/app/2241410/Chex/.
[4] P. Saár, "Hexagonal Chess," Hexagonal Chess, [Online]. Available: https://hexagonalchess.com/.
[5] esbudylin, "HexChess," GitHub, Dec. 23, 2023. [Online]. Available: https://github.com/esbudylin/HexChess.

Appendix A – Key Terms

Term	Definition
Association for Computing Machinery (ACM)	*An international, non-profit, US-based, scientific education and computation-oriented society.*
Artificial Intelligence (AI)	*Goal-driven or human-like intelligence exhibited by a machine implemented in a method allowing learning and complex logic.*
Alpha-Beta Search	*A depth-first search algorithm that uses alpha-beta pruning to avoid searching much of the tree.*
AlphaZero	*A game-playing algorithm, developed by Google's Deepmind, that uses neural network reinforcement, self-play, and a version of Monte Carlo Tree Search to determine the best move in a given game and game state.*
Deepmind	*A Google subsidiary that developed AlphaZero and other AI-based technologies.*
doctest	*An easy-to-use testing framework for C++ that allows for unit testing and test-driven development.*
Forsyth-Edwards-Notation (FEN)	*A standard notation that describes board positions of chess games. It can be used to provide all of the necessary information to restart a game from a particular position besides three-move repetition.*
Graphical User Interface (GUI)	*A graphics-based interface with buttons, icons, and menus allowing ease of use for various users.*
Institute of Electrical and Electronics Engineers (IEEE)	*An American-based professional organization/association for electrical engineers, electronics engineers, and other related STEM fields.*
Michael Davis's Harm Test	*An ethical test based on the question: "Does this option or situation do smaller amounts of harm to any other alternative?"*
Michael Davis's Reversibility Test	*An ethical test based on the question: "Would this choice still look good if I traded places?"*
Minimalist GNU Compiler for Windows (MinGW)	*N/A*
Monte Carlo Tree Search (MCTS)	*An algorithm that enables searching for the best moves to play in a game from a given game state. The algorithm uses statistics versus the traditional brute force depth-first search used in traditional game algorithms.*
Microsoft Visual C++ Compiler (MSVC)	*N/A*

OpenGL	*A free, multi-platform, graphics library specialized in rendering 2D and 3D vector-based graphics on graphics processing units (GPUs).*
PyTorch	*A neural network construction, design, and training library that works with various programming languages including C++.*
QRS Coordinates	*A coordinate system compiled by Amit Patel at Red Blob Games that uses three axes on a flat surface allowing for hexagonal coordinate indexing.*
SageMaker Studio	*A framework that allows developers to build, train, and distribute machine learning models using Amazon's cloud computing resources.*