OCR A level Computer Science NEA Project

Borys Swider

April 29, 2026

Contents

1 Inital ideas

1.1 Idea 1 - goreplicate

Explanation: A program that replicates ("syncs") files between computers in a distributed manner e.g. you change a file on one computer, that change is "replicated" to all other computers.

Programming language: Go, due to its speed as a compiled language, as well as heavy support for concurrency.

Complexity: The program will need to be able to detect changes to files in the filesystem, feature a networking stack for discovering, authenticating and communicating with other devices, as well as needing a program interface for control. The program will also need to handle file conflicts, storing file metadata in a database, as well as ensuring file integrity during transfer and storage.

Computational methods: During the design process, I will use heavy abstraction to ensure the program is modular and not overly extensive, making maintainance, debugging and testing easier. I will also use decomposition to tackle one problem at a time, such as networking then file detection in the design process.

1.2 Idea 2 - godungeon

Explanation: A game that procedurally generates dungeon levels, with different rooms, levels and loot rewards, through an ever increasing difficulty curve. The game would also feature pathfinding for bosses and enemies, as well as a combat system and other common game attributes such as armour, weapons, health and more.

Programming language: C# or C++, due to their wide support and use for game development, allowing me to use either a game engine, or render the game using rendering API's. It is also highly performant, which would be important for my game, as the logic used for pathfinding and procedural generation can be computationally expensive, and I would like to ensure as many people across different hardware can run the game without issue.

Complexity: The game would have complexity from procedural generation, which would include large amounts of logic to ensure the levels are varied and playable. Pathfinding for bosses and enemies would also be complex, as well as implementing the combat system in an effective way to ensure players do not "ghost hit" enemies. Multiplayer support for cooperative gameplay.

Computational methods: The game would be designed with abstraction and modularity in mind, allowing it to be expanded easily and to aid the procedural generation algorithm design process. Decomposition would also be used to further aid modularity and to allow me to focus on one aspect of the game at a time, improving the overall quality of the game.

1.3 Idea 3 - Chess Game

Explanation: A chess game with a fully graphical user interface, integrating all the usual rules and UX of playing computer-based Chess, such as moving pieces, move history, recording moves and point score for both sides. The program will also incorporate a Chess Engine, allowing the user to play against the computer, or to provide the user with tips and hints of the best moves to make. Other features such as a percentage for which side is winning, as well as providing inference on which moves the user made were good, and which were bad (blunders, etc.).

Programming language: Python or Go, due to their ease of use, availability of libraries and portability.

Complexity: The program would need to have a user interface for interacting with the game, as well as a chess engine for calculating the best moves to make. A database would be needed in order to store move history, as well as previous games so that the user can return to them later. The program would also integrate networking for multiplayer support, both within the program and with chess servers.

Computational methods: The project would implement abstraction by separating the user interface/game and the chess engine backends. This will allow for easier debugging and testing, as well as making concurrency and non-blocking UI easier to implement. The project would also incorporate decomposition by using modular interfaces between game functions in order to break down the complex task of implementing chess rules easier.

2 Section 2 - Analysis

2.1 Problem definition

For my project, I wanted to investigate something relating to personal interests, but with suitable complexity. For this reason, I decided I wanted to create a high-performance, locally-hosted chess game, with an accurate and responsive graphical user interface (GUI), with data storage for past games and move history in databases, as well as a multiplayer system with networking for playing with friends.

In addition, I wanted to implement my own chess engine, allowing the user to play against a computer, as well as recieving feedback about their moves and how they affected their position in the game. There are currently available local chess clients, notablly PyChess, however they are usually made up of legacy styling and lack the performance I am looking for.

Creating my own chess game will also allow me to better integrate the custom chess engine into the GUI, as well as giving me opportunities to explore many different fields; such as previously stated networking (peer-to-peer for multiplayer chess games), databases (SQL), GUI design (pygame/OpenGL), as well as algorithms (Minmax Algorithms) and machine learning for the chess engine.

This project is suitable for a computational solution as playing chess using software provides a much more friendly, easy to use, and secure way to play the game. This will be broken into numerous reasons.

- Firstly, when playing chess with a physical board, playing with someone else over a long distance becomes a significant hassle. Both players would have to devise a system for telling each other what move was made, and needing to replicate this on both sides in order to keep their boards in sync. An example of this would be algebraic notation; (e.g. knf6 → knight, moved to f6) along side rank/file notation (f6 → square located at the 6th column, 6th row). Evidently, this can get confusing fast. Computationally, this can happen fully automatically over a network.
- Secondly, trusting the opponent to give valid moves opens up an opportunity for cheating to occur. This method of play relies on trust between parties, or the need of a middleman in order to accurately report moves to both parties. This makes playing chess competitively a huge hassle to do. Computationally, this can be solved by automatic checking of legal moves and rules.
- Lastly, this requires both parties to have access to a full, identical chess board to be able to play together. This may not be possible due to economic constraints, or time availability. Some people may not be able to sit down for potential hours to finish an entire chess game in one sitting. Computationally, the game state can be saved to be continued later.

2.1.1 Computational Methods