An Isometric Game of "Chess"

System Reference Manual

Joseph Cavanaugh

Dr. Benjamin Bishop

2/21/20

Submitted in partial fulfillment

Of the requirements of

CMPS 490 – Computer Projects

Abstract

1. Table of Contents
2. Abstract Final
    a. Introduction
        i. Project is a computer-game based game, based on games like chess and Final Fantasy Tactics.
        ii. Advisor is Dr. Bishop
        iii. Goal is to create a game with visually demonstrates a full game as well as enforce the rules of the game.
    b. Justification
        i. Game is based on time-proven classics that most know how to play or are aware of.
        ii. Commercial value is based on nostalgia
    c. Purpose
        i. To create a game for experts and beginners alike.
    d. User Interface
        i. Graphic and text based. Uses keyboard controls.
    e. Language, Feasibility and Implementation Plan
        i. Language is C and graphics is C based OpenGl
        ii. 5-Small Development Phases
            1. Develop Board
            2. Develop Movement
            3. Develop Attacking
            4. Draw Graphics
            5. Link Graphics to Previous stages
3. Feasibility
    a. Chose C because I am used to working in it
    b. All parts are feasible in the time frame
    c. Extra time will go towards working on any extra design, although unlikely
4. Requirement Specification
    a. System Model

b. Functional Requirements
    i. Board is randomized every time
    ii. Offline game vs AI or Hotseat
    iii. Keyboard controls
c. Non-Functional
    i. System must be able to run C. Is not resource intensive so should be able to run on about any system.
d. System Evolution
    i. Code should be well-written and commented enough so that it can be easily modified or added onto in the case of additional features being added.

5. System Design
    a. Introduction
        i. Update loop that will continually update the graphics, look for keyboard input, and get game state.
        ii. Board object that stores all information relevant to the playing field
        iii. Player object that has piece and team information stored inside it
    b. UI
        i. User will input commands using their keyboard
        ii. If an invalid move is taken the system will alert the user
    c. Turn Flow
        i. Game takes place over a series of turns with two win conditions
        ii. Turns have been implemented in a flow like fashion. IE. After the player goes the enemy goes regardless of what happens. A→B→A……
    d. Help System
        i. Help system is implemented in every game which informs the user of incorrect moves and the current turn and the turn number.
6. Testing
    a. Introduction
        i. Unit Testing – Testing the main functions that make up the core of the project

ii. Module Testing – Testing attacking, movement, and board drawing
iii. Subsystem Testing – Testing interaction between the above three
iv. System Testing – Requirements to be met
v. Acceptance Testing – User satisfaction and quality assurance

b. Testing
    i. Phase 1: Offline Game
        1. Test "Initialize Methods" and functions included in the "Main functions" section of program.
        2. Test support functions
    ii. Phase 2: Logic
        1. Test individual functions like "AttackRules()" and "MoveRules()"
    iii. Phase 3: Subsystem and System
        1. All systems are local and subsystems are completed in Phase ½
    iv. Phase 4: Quality Assurance
        1. Allow others to play the game for feedback
    v. Schedule:
        1. Period from 10/21 – 11/21 – Unit testing began
        2. Period from 11/22 – 1/1 – Unit Testing ended
        3. Period from 1/2 - 3/14 – Module Testing began and ended
        4. Period from 3/15 – 4/30 – Acceptance Testing began and ended

7. User Manual
    a. Introduction
        i. Project is an effort to introduce an interesting twist on old classics
        ii. Simple to pick up and play
    b. Introductory manual
        i. Run the file from the folder where it is downloaded to.
        ii. Game is handled by the GUI
    c. Offline Game
        i. Displayed through graphics and some text-prompts
    d. Help System
        i. Text will display an illegal move in console.

ii. Rules manual is included

e. How to Play

i. Overview

1. Chess-like tactics game for 2 players or singleplayer
2. Two-win conditions
3. Random board creation

ii. Objective

1. Take all your pieces or have more pieces by the end of the max turns

iii. Board

1. Pseudo-randomly created
2. Has same amount of pieces for each player.
3. Each tile has a different height
4. Turn-timer. 15 moves each.

iv. Turns

1. Moving along different height terrain
2. Attacking the enemy

8. Glossary
9. Index

Table of Contents

Abstract Final

Isometric Strategy Game/ “Chess with Height”

In this project I will deliver a game that has the following features. The game will be isometric, meaning the board the game takes place upon will have height and will also be pseudo-randomly generated. To prevent the total domination of a player or computer, each Z level will have a minimum of four tile spaces and every tile on the board must be accessible to the players’ pieces. To prevent a dead lock between the player and computer a turn count will be implemented. If a set number of turns pass the player with the most pieces left will automatically win.

Pieces will be able to hit other pieces on the same X-plane and Y-plane. You cannot hit diagonally upwards on the Z-plane, but you can hit diagonally downwards. Additionally, the players will only be able to attack in a cross pattern. The players can move and then attack in one turn but will not be able to move after they attack. Each piece will be able to move a minimum of one tile and a maximum of four tiles. Moving “uphill” or up the Z-axis will cost 1.5 movement and moving “downhill” has a penalty of 1.5. You cannot move to a tile that is two or more Z-planes above the piece. You can move to a tile exactly three below your piece, but the turn ends for that piece upon movement. In the uphill example if my piece is at Z-level 0, I can move to Z-level 1, but not 2. In the downhill example, if my piece is at Z-level 2 and moves to Z-level 0 that piece ends it turn right upon dropping down.

The player’s game pieces will resemble checker pieces with a black and a white team, with the white team moving first. The game will be single player with a computer playing against the player or a hotseat option. The computer’s AI will be relatively simple and will move towards the player’s pieces to attack.

The game will be written using the C language and will use OpenGl for graphics. Visual Studios IDE will assist in compiling, running, and testing code.

Basic feature set: Pseudo-randomly generated isometric game board, computer AI, turn timer, checker piece models, rotatable camera, basic movement and attacking rule set.

If time allows extra features I would like to include would be as follows and in order of priority: Models for individual pieces, animations for pieces, terrain types, the ability to save a game, different difficulties to play against, class system, ranged and melee units with different skills, stats, animations for skills, BGM and sound effects.

Justification and Defense

Chess is a game that we all know and have most likely played at one point in our lives. It is probably one of the most well-known and popular board games in the world, even being reported by the World Chess Federation that 605 million people play chess. I would like to expand upon the already loved game and introduce "height" to the board opening more avenues of play and strategy. Many games have already been done in this style and have a cult following. A few of these games like *Disagea* and *Final Fantasy Tactics (FFT)* are two of the most popular and have multiple sequels and have inspired many other developers to create similar games. *FFT* received extremely positive reviews and has sold over 2.4 million copies worldwide. I would like to follow suit and create a game like *FFT* and *Disagea*. "Chess" is the perfect game to expand upon because it already plays out well on a square board and if you add height to the chess board you can add an extra layer of strategy of the game. I say "Chess" because the pieces of the game will not borrow the conventions of movement from chess. It is easier to visually image "Chess" with height than to try and explain an isometric turn-based strategy game. Graphically, the board the game takes place on will be no different than your standard chess board, the exception being made for height. The pieces will, at first, be more visually akin to checker pieces and will eventually develop into sprites.

Other games of this type having sold extraordinarily well, despite their relative simplicity in design, is indicative of how popular a game of this type can be. In this day and age where remakes and revivals of games have become alarmingly popular due to the effect nostalgia has on the average consumer, a game that capitalizes on this, and also advertises it well, has a chance to profit in the right market.

Description of Game

The game in its simplest form is a turn-based 2.5D game between a human player and an enemy computer. The game rules are not extremely complicated and will come naturally to any player who has played chess or checkers before. The added component of height adds rules to the game that might not be known to the player. In the game's completed bare bones form it will follow these rules and have these key features.

The rules and features of the game as follows:

- A randomly generated map 5x5 or bigger. The terrain will go from a height of 0 (base or sea level) to a height of 4. This might make games unbalanced in favor of one side or another, but its ultimately up to the player to capitalize on the advantage or overcome the disadvantage.
- Unintelligent "AI" that will randomly move towards the player and try to take the closest piece.
- The game will be turn based. The current player will move one piece and will then end their turn. There will also be a turn deadline; after 30 turns the game will end in victory for the player with the greatest number of pieces. This is to avoid a stalemate and an endless continuation of the game.
- The pieces of each player (of which they will initially be given five) will be grouped near each other, but not directly next to each other.
- The most important part of the game itself is of course moving and attacking, but there are complexities to consider when traversing a 3D space. Most simply, all pieces per-turn

can move a maximum of 5 spaces and hitting an enemy piece from one space away removes it from play. (* Coordinates take form of x,y,z)

- **Attacking**
    - Attacking an enemy piece ends the current players turn.
    - Pieces cannot attack on a flat diagonal (I.E. A piece at 0,0,0 can not attack a piece at 1,1,0)
    - Pieces cannot attack on an upwards diagonal (I.E. A piece at 0,0,0 can not attack a piece at 0,1,1)
    - Pieces can attack diagonally downwards.
    - Pieces can attack during a fall or climb maneuver.
- **Movement**
    - A player cannot exceed the movement cap
    - Pieces cannot move to a space 2 or more higher than their current position (I.E. A piece at 0,0,0 cannot move to 0,1,2) except in the case of a climb maneuver.
    - Pieces that fall a height of 3 or more immediately end their turn.
    - Moving upwards and downwards (in z axis) costs 1.5 movement points.

Feasibility

The timeline of the project, along with what needs to be completed is extremely feasible. Each piece individually is neither extremely difficult nor very time consuming, however my knowledge of each portion of the project is lacking. The most difficult aspect of this project will be if I get to the additional features that I would like to add onto the project towards the end. The hardest part of the project for myself will be the AI for the computer. Creating an AI for a chess bot is already difficult, with the millions of possible configurations for a standard game of chess. An AI for a game of chess with an added dimension will probably be even more difficult. The project will be written in C in VisualStudios and the graphics will be done in OpenGL. The timeline of about 3.5 months is possible for the base version of the game.

Schedule

A schedule that I will try to keep will be as follows.

First Stage: Static Board and Movement/Attacking – Feb 23rd

- Have a text version of the game with at least two pieces. One player piece, one uncontrolled enemy piece that can be attacked.
- A text version of the board to test the height components.
- Have all game rules and logic implemented
- Piece placement working

Second Stage: Player and Enemy classes and Randomly Generated Board – Mar 15th

- Board should be able to randomly generate itself at this point with the proper logic behind it.
- Player and Enemy should now have all pieces available to them. Turns and the turn timer should be implemented and the enemy should have a basic AI to attack the player.

Third Stage: UI and Graphics – April 5th

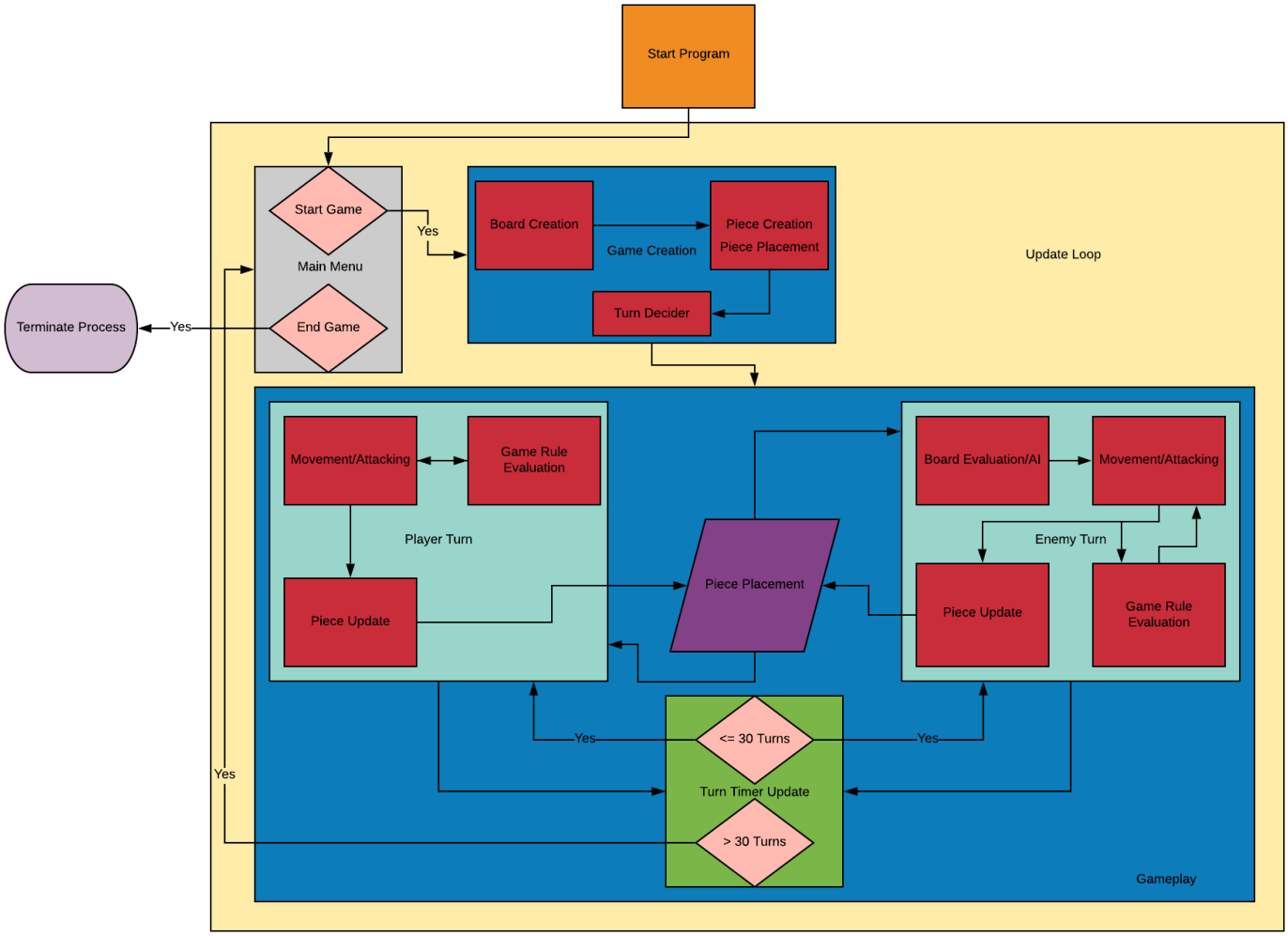
- The player should have a minimum UI to keep track of basic stats.
- Converting game into 3D model that is playable. Pieces and board most importantly.

Final Stage: Additional Features in order of importance – By due date

- Textures for pieces and board – More human like models for pieces
- Classes, stats, and skills for pieces
- Main menu
- Animations
- Sound effects and BGM

Requirements Specification

System Model



Functional Requirements

When the user starts the program the first thing that will happen even before the main menu opens is the establishment of the update loop. The update loop the program will be encompassed in will keep the game moving along visually.

The game creation step will generate the board and the pieces.. The pieces will be randomly placed on the board while also initializing the database that contains the pieces' locations. After the pieces are placed the game will decide whether player or computer moves first. After initializing the game, the enemy AI will be created with knowledge from the game creation initialization.

During either the player's or enemy's turn movement/attacking will be done and will simultaneously check the game rules to see if a move or attack is legal or not. Once a movement or attack has been confirmed the piece location will be updated. The computer will read data from the piece placement database to use during their turn. The AI will evaluate the board's state and will try to move accordingly. The player will be able to control their piece movement with the keyboard. Arrow keys will move the camera, 'X' will select, 'Q' will deselect and 'WASD' will hover over pieces with the cursor.

Each turn taken will update the turn timer until the timer reaches thirty. If the turn timer is less than thirty then the game will continue, otherwise the winner is decided upon the number of pieces each player has left. An even number of pieces on both sides will result in a stalemate. After a game over screen the game will terminate.

User Interface

The user interface will be minimalistic and display only important information. Who's turn it currently is, number of pieces left, number of movement points left, cost of the movement, current height of the selected piece and the height of the hovered over tile. The graphics will be included with the game and will be relatively simple. The pieces will be easy to distinguish based on teams and the board heights will be visually clear to the player. A rotatable camera will be included so that the player will be able to view the board from different angles.

Non-Functional Requirements

- Performance
    - The game is run locally and does not depend on a server to run so performance after the initial loading process should be quick. Resource allocation should initially go to game creation and afterwards be focused on keeping the game continually updated. The performance should only dip if the user has programs running in the background.
- Portability and compatibility
    - The game should be able to run on any operating system that can run C. If the game was ported to iOS or Android, the OS should still be able to support the game because of the lack of resource intensive processes needed to run the game.

- Reliability
    - The game should run without bugs or failures for a full playthrough and should try to reach above 90 percent reliability.
- Security
    - The game will be run locally and therefore should not have any security concerns
- Usability
    - A readMe will be included with the program that teaches the rules of the game. The simplicity of the game itself should help users to learn the game reasonably fast. Easy to learn, hard to master is the goal. The minimalistic UI should not interfere with the user in any way and cause unnecessary distractions.

System Evolution

The game will be written entirely in the C programming language and revisions to the code would require an entire refactoring of the code. The code for this project will be made with future features in mind. This game can take on many forms and therefore it is important that functions can be written easily and quickly. Adding on functionality should also be made as pain free as possible. Consistent commenting on the code will be necessary to make sure future programmers know how everything is used and how it works.

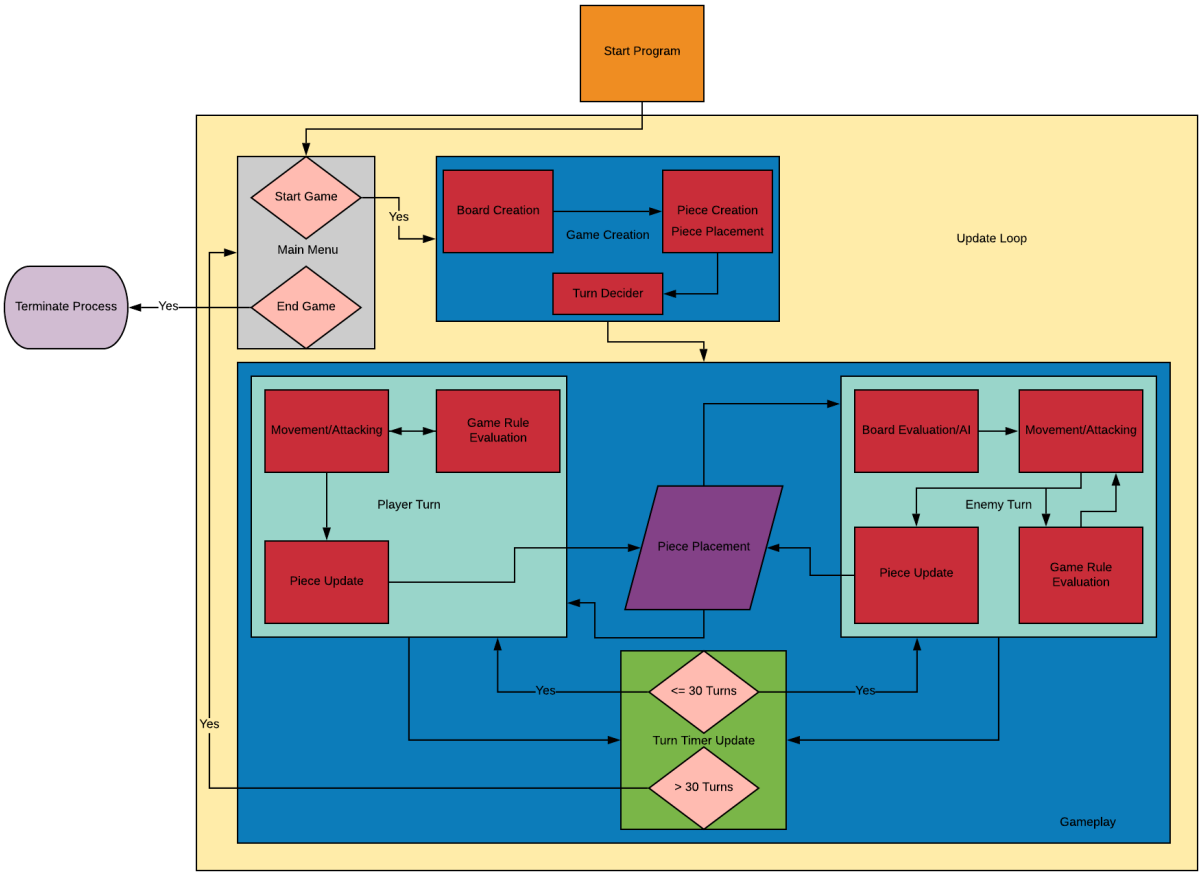
System Model

Introduction

When executed the game is opened and managed by a controlling program. This program will create an update loop in which the game will be looking for certain conditionals. These conditionals will decide the current game state. The update loop will keep the graphics, board, and turn state stored in their respective data bases. There are two main objects in the game at any one moment which are comprised of the Player and the Board object. The Player object keeps track of player team and what the number of pieces they have left. The Board object keeps track of all pieces on the board as well as the generation of the board which is used in the graphical generation of the board. The information is stored in the board object.

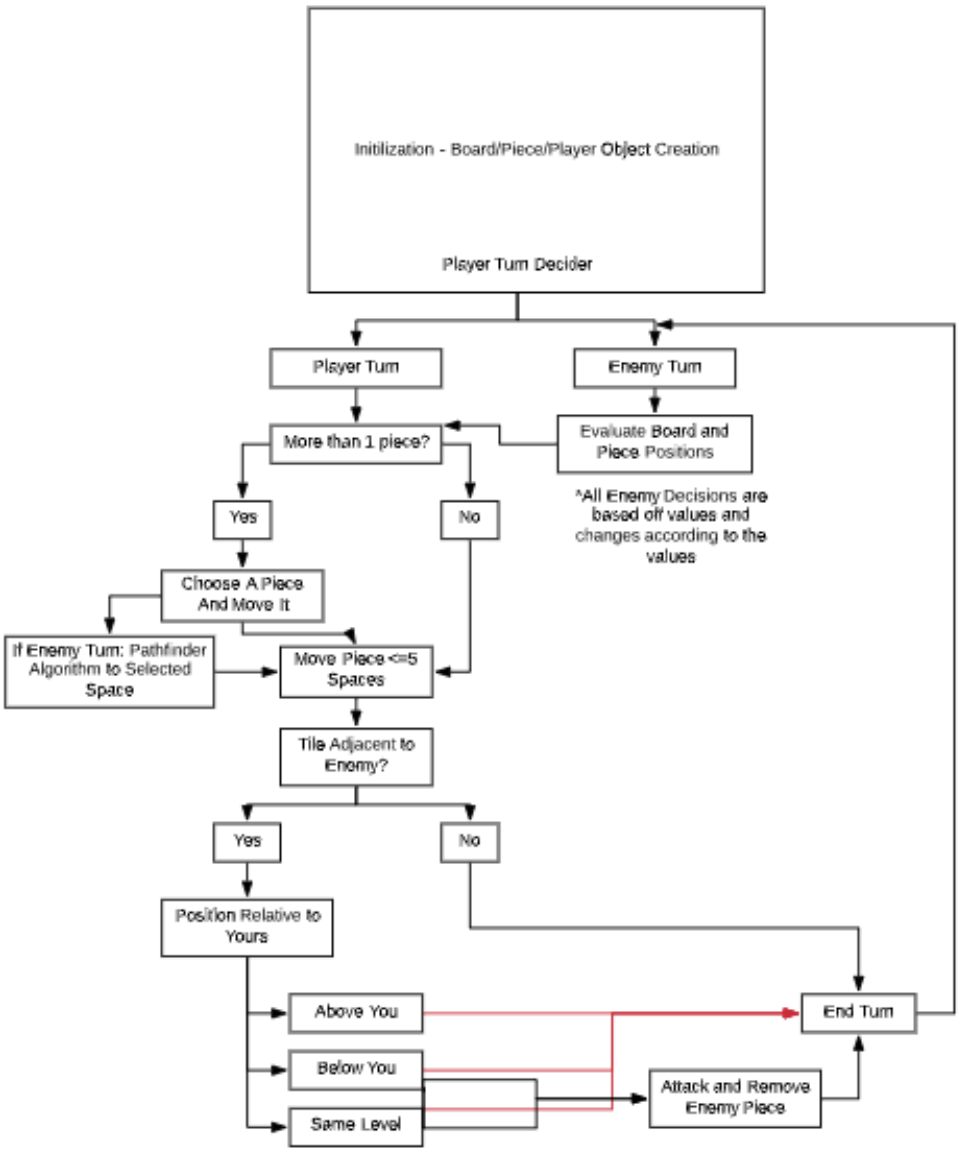
The game has been designed so that later versions can easily add different types of pieces and tile types to the game. The other parts of the game need no additions after they are created the first time. The use of less complicated code for each part of the game will allow for easy additions to the final product in case the need ever arises.

Top-Level System Model



At the top-level of the system design is the overall system model. The game when ran will bring up the start menu where the user will be asked to start or exit the game or to access the settings. When selecting to start the game the update loop will begin which will first and foremost generate the board and populate it with pieces for the player and the enemy. The player and enemy class's will be created, and the turn decider will choose who moves first. All these objects and the information relating to them are encapsulated in the update loop so that there is ease of access when referencing information. The graphical UI will also be kept in the update loop so that as the pieces move their location is updated visually so that the player can see.

Flow Chart of Turns



This diagram shows a finite state machine interpretation to the turn process. This flow chart only corresponds to one turn in the game and does not depict a whole game. In words, a turn follows this pattern.

1. The player or enemy is decided to go. If the player's turn is first, then the player must choose one of their pieces to move. The move must follow the movement rules set in place. The piece will then move to a valid space.

2. If the piece is adjacent to an enemy piece the evaluation of where the enemy piece is will take place. If the enemy is below you or on the same level as you the option to attack is given to the player. If the enemy is above you the player is not allowed to attack that piece. Lastly if the piece is attacked it is removed from play and that board space is freed up. The turn ends after this.

3. The enemy's turn begins after this. The enemy follows the two rules above but an additional step of evaluating the board and choosing the right path to move along is added in. This step would come before choosing a piece to move.

   These three steps alternate up until one of the two players win or the turn timer reaches its end. In the case of the turn timer running out the player with more pieces will be declared the winner.

UI

The user will input commands using the keyboard. The keyboard will be used to move a cursor throughout the screen, manually selecting pieces and the place they will move. The system will alert the user to when an invalid move is selected and display this in an informative manner. The graphical display will be able to visually represent data instead of having to tell the user where a piece has moved via text. A manual on the rules and how to play will be included in a README file later in development. As there are no command line prompts a list of commands will not be needed, but as different settings plan to be developed a detailed list of what each setting does will be included in the documentation. The data that will be displayed for the user will be the current turn timer and the number of movement points they have left.

Help System

The help system for the game is not very extensive but there are in game tips that the user can turn on and off at the start of every game. If the user makes an incorrect move or attack there is output designed to tell the user this. An in-game guide is a later development goal that can help the user with learning how to play for the first time.

Testing Design

Introduction

The testing for this game depends on unit testing, module testing, subsystem testing, system testing and acceptance testing. Functional testing only focuses on the output to check if meets the requirement or not. Incremental testing is so we can check if our previous working version can work with additional code that we've added in. Lastly, unit testing so we can see if individual code units work as intended. The most important aspect of this project is the board itself so we must test to be sure that everything works as intended with the Board object.

For the unit testing, everything is written in one place "main.c" so it is important that the order of method calls, and the functionality of the method calls are correct. The methods must be tested while incrementally adding functionality. If you add more than one feature at once you may unknowingly end up with bugs, or even worse, a method that doesn't work, which is a huge time waster. The testing of main and all its sub-methods will be done through state testing. By probing the program to see where we are and what the current logic is doing, we can see where the applied logic is incorrect. After perfecting one method with no reproducible bugs we can then start working on the next method functionality, refactoring previous code as we add in the new method.

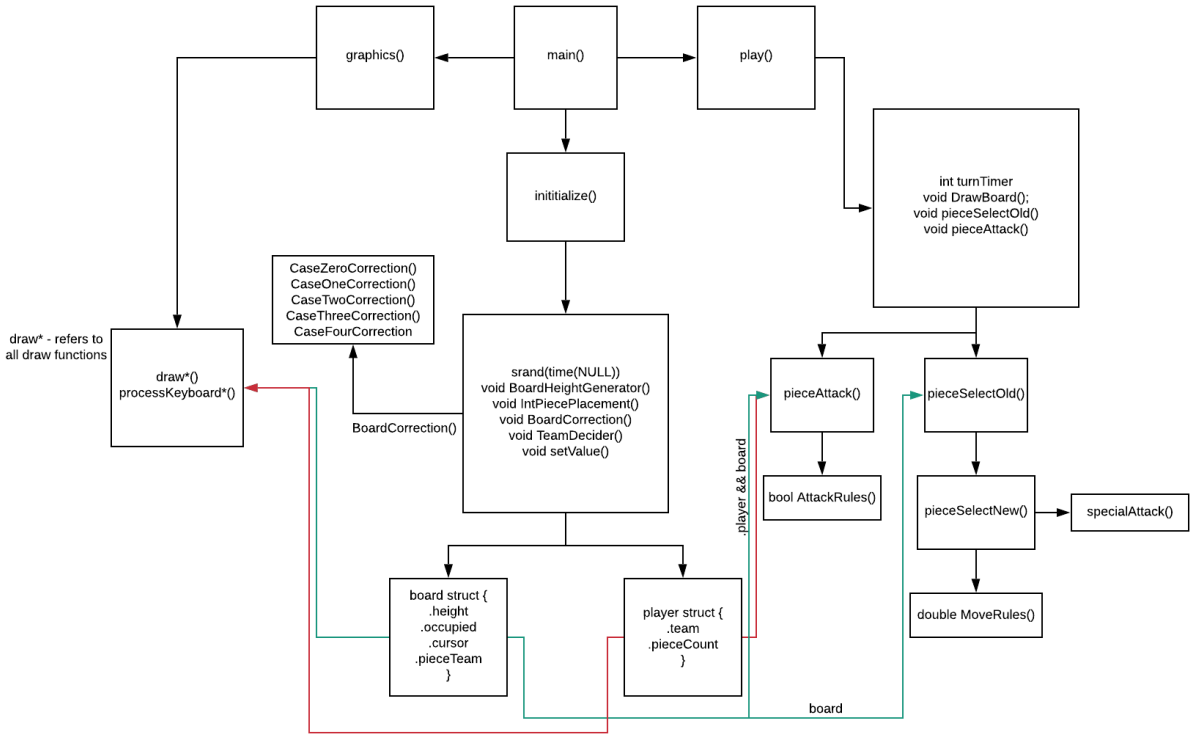
Module testing is testing code that is compile-able. For this project every piece of game logic must be tested so as not to break the game. The movement, attacking, turn order, win conditions, and graphics must be tested in their playable environment. All these conditions are met by the previous unit testing so we can confirm that all logic has been individually tested.

Subsystem testing involves testing a series of programs which work together to accomplish a greater goal. The subsystems that are being tested are how the “BoardGenerator” method works together with movement and attacking. Along with this we must be able to have the graphics interact with board object itself, updating itself every turn. Several test games have been run to test the interactivity of these methods and how well it meets the demands of an actual game.

System testing verifies the requirements specification. As the requirements for this project are the ability to run an offline version of the game, as well as visually display it in openGL it is rather simple to complete these specifications. Since these goals must be met for Subsystem testing to be completed, system testing is not required.

Acceptance testing ensures that the users of the end product find the it satisfactory. A game requires that the acceptance testing be used by multiple people to test multiple games. As of this writing we are currently in quarantine, so acceptance testing sample size was low. Generally, however, the game should perform as expected and users should encounter no troubles or raise any concerns. If all users are satisfied, then Acceptance testing will be successful.

Phase 1: Offline Game (Unit Testing)



The play() and initialize() methods must be tested and proved to work on their own, before implementing any graphics. There is a visual test method to show that all values work as intended however in the form of DrawBoard(). DrawBoard() lets us visualize how the cursor moves and how pieces move and attack. As these three methods form the core for the game to work the entire system depends on them to work. Minor tasks such as getting input from the user are side programs that are imbedded in most other methods.

Phase 2: Logic (Module Testing)

Although most of the overall system was completed in the unit testing, module testing is useful to us by letting us test the logic of functions like AttackRules() and MoveRules(). These functions are the underlying logic for pieceAttack() and pieceSelectOld/New(). These two functions were refactored to streamline the code from their parent methods. As the game logic is imperative to making the game a game these functions must be tested independently of each at first. After both worked, testing in tandem began. A special function was added to handle the special cases of climbing and falling called specialAttack(). This function was tested separately of the other logic functions and proved to work. Each of these three logic function needed to be tested under separate and unique conditions and could not be trusted to work without extensive testing.

Phase 3: Logic (Subsystem and System)

All of the system takes place locally so there was no need to run any system test. Subsystem tests are already completed while in phase 1 and 2 so there was no need to explicitly perform any subsystem tests.

Phase 4: Quality Assurance (Acceptance Testing)

Once phase 3 ended and the product works in its base state, without graphics, acceptance testing began. As mentioned previously, testing size was not big enough, but the few who did test the game were satisfied with the product in its base form (knowing it was not yet completed and what it was supposed to be). One observation I noticed in this phase of testing was that gameplay was too short, so considerations are being made to reduce movement points or to increase the size of the map.

Testing Schedule

Week 10/21 – 11/21: Unit Testing began. Was able to flawlessly move a piece from its old location to its new location. The introduction of a cursor and keyboard movement was also implemented. BoardGeneration and BoardCorrection were also tested and completed in this time.

Week 11/22-1/1: The logic to attack other pieces as well as the player struct was added in. New logic was added to remove bugs that were found after adding new methods. From these bugs sub-functions like AttackRules() and MoveRules() were created. Testing to complete these functions were finished by this timeline.

Week 1/2 – 3/14: Module testing began. After testing each method individually, the functions were added to methods to perform sequentially. This testing showed flaws of the functions when working together. For example, when skipping a turn the cursor could attack as if it were a piece. New logic was added to remove these bugs from the game.

Week 3/15 – Current: Acceptance testing. Testing of the completed base product. After the acceptance testing was completed work on graphics began. The game has dropped the functionality to be a single-player game based off AI and has instead made it local multiplayer. The AI was not performing well in acceptance testing and logic was added to have players play in a “hotseat”.

User Manual
Introduction

This user manual will be you guide to the game I have affectionately dubbed Tall Territories (working title). Tall Territories, or TT for short, is a local-multiplayer game played on your computer. The game is inspired by games I have enjoyed, and I hope that you enjoy my homage. TT is beginner friendly, but I imagine that people well versed in games like chess or checkers to find enjoyment in it as well. There aren't as many rules to the game as chess, but is slightly more complex than checkers is. The game does require two people to play at once in a hotseat fashion. This means that players must alternate turns sharing one computer.

Introductory Manual

The game is incredibly simple to play as there are not many rules or user controls. As of writing there are only 8 possible buttons that you would have to hit in order to play the game. When downloading the game, the user must go through their files to find the file titled “main.c”. This should be in the first level of the folder. Accessing or modifying any other file or folder may cause the game to crash or be unable to operate correctly.

The user will encounter a main menu screen where they will be able to choose between starting a game, settings, or to quit the game. The game is entirely offline and intended to be played with a friend by taking turns, but the possibility to play by yourself is there too. At the start of every game there is an option to enable or disable tips. By enabling tips, a player will be able to learn the rules and buttons to the game. The rules are also included in this manual.

Offline Gameplay

Offline gameplay is displayed through graphics and text-prompts. There may be more settings to choose from in future updates, such as number of pieces to play with, if you want to play against a computer, and the difficulty of the computer player.

Help System

If the user makes a move that goes against the rules, such as making an illegal movement or attack, the system will kindly inform the user via text prompt that it is not possible to do that. Reading the rules before hand however will allow the user to prevent these messages from appearing.

How to Play Tall Territories

The following will be a detailed guide on how to play. The intended audience is for new players or for those who need to be reminded on the rules. A rulebook on how the game is played will be useful for all players however.

Overview

Tall Territories is a chess-like game that incorporates the strategy of checkers; moving your pieces to get the advantage over your opponent, with the added element of having the height advantage. As any decent strategist know, having the high ground is always preferred when battling your opponent.

Objective

The objective of TT is to either take all your opponents' pieces or to last longer with a larger number of pieces. Once a player loses all their pieces the game is over, and the winner is declared. Alternatively, if the turn timer reaches its max, the game will be over, and the winner will be the player with more pieces.

Board

The board of TT is pseudo-randomly generated with strict rules as to not make the board unplayable. The board has a few things to keep in mind.

- Pieces – All players start with the same number of pieces. Although the pieces themselves don't have any special properties, they're important! Protect them!
- Turn-timer – Each player gets 15 moves to make. Once this timer runs down a winner is declared so act fast!

- Height – Each tile on the board has a set height that will stay that way for the remainder of the game. Knowing how height affects your movement is important and is a key aspect to winning the game.

Turns

One turn in TT is compromised of the two phases; moving and attacking. After a player uses or skips both actions a turn is said to be over and the next player takes their turn.

- Moving – A player can only move one piece per turn. A piece has a maximum of five(5) movement points to use. The difference in height depends on how many points you use to move along the map. There are five movement cases.
    - Flat terrain – Movement to the same height value. The cheapest cost to movement points and is the fastest way to move around the map.
    - Uphill – A costly, but advantageous action. Having the higher ground is always better, but moving uphill costs 1.75 movement points.
    - Downhill – The middle ground between uphill and downhill. Running downhill is hard and clumsy! This action costs 1.5 movement points
    - Climbing – If your opponent is on higher ground than you are and there's no way to approach except for a sheer cliff face, give them a surprise attack. This movement action uses both an attack and move action. You'll be open to attack after climbing, but you may be able to surprise your opponent…. The maximum height you can climb is a difference of two height.

    - Falling – If your opponent made an escape and jumped off a cliff our situated themselves below you it might be best to make a falling attack. A falling attack can be made from a height that is 2 or 3 the tile you're moving too, but a fall will definitely leave you winded. After this action is made it will be impossible to move or attack, be careful to avoid getting trapped in a tall place.

- Attacking – After moving a player's piece will be able to attack, excluding after falling or climbing. Attacks can only be made in a few cases
    - Same level – Traditionally all attacks can be made on the same level as there is no height difference between you and the attacker.
    - Downhill attack – If you are above your opponent by one level you will be able to attack downwards at them with no fear of retaliation!
    - Uphill attack – These attacks are impossible. Your opponent has the high ground and stabbing at them from below is pretty useless.
    - Falling or Climbing attacks – These are special attacks that are made in the movement phase. These let you counter opponents that are very far below you or just above you.

Index

Glossary

AI – Artificial Intelligence that decides the enemy computer's movements and plays against the player

Board – The place where the game takes place. Stores information about a number of different variables such as height, occupied, cursor etc

C – A popular programming language that is easy to install on virtually any OS

Chess – Turn-based board game that is popular around the world.

FFT – Final Fantasy Tactics, a cult classic game that is a spin-off of the popular Final Fantasy games. Many themes are kept similar from the main games.

GUI – stands for graphical user interface

Hotseat – a game where two players play on one computer against one another

Isometric – The game is 2D, but the camera angled in such a way that a three-dimensional effect is achieved. 2.5D or pseudo-3D are also ways to refer to this.

Initialization – Creating the default values needed for a playable game.

Locally – runs on the user's computer and does not need internet connection to play

Method – a piece of code that is used in other methods. Many methods make up a program

openGL – a cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics.

OS – an Operating System; Windows, Mac, iOS, Android, Linux, etc

Pieces – player controlled, used to beat the enemy and used to win the game

Random – a variable that is randomly decided upon when the game is created

readMe – A file typically included with most software products. Gives the user a detailed overview of the product and functions of the system.

Turn Timer – counts to thirty, adding one to the count every player and enemy turn. Ends game when it reaches thirty and prevents stalemates.

Update Loop – The loop that makes the magic happen. Keeps the game running as long as the program is running

UI – User Interface; what the user see's and how they can interact with the program.

Visual Studios – an IDE that supports many different languages and tools.