KINGDOM BUILDER

Prospectus

Prepared By Khoa Le, David Lu, Krish Shah, Vladimirs Sterbs

Table of Contents

Project Timeline:

Due Dates:

The submission of the prospectus is March 24, 2023.

The submission of the game is May 5, 2023.

Major Milestones:

1. Finish Prospectus
2. Develop UML
3. Finish Presentation
4. Finish Board Logic
5. Finish Player Logic
6. Finish final UI positioning
7. Finish Game Loop
8. Test cases and check for bugs
9. Finish Game

Meetings:

- Sunday 2 pm call on Discord.
- Use iMessage during school days as an alternative to Discord.
- Communicate on Discord for any updates on weekdays after school, because it has better compatibility with computers.

Each member of our group has their choice of which IDE they want to utilize during the project, but we will collectively store our project on GitHub.

Schedules Time Table

We will use the following time table to guide us during development. We will assign due dates on a weekly basis, and each person will have a weekly checklist of tasks to accomplish. We will also track the due dates for our milestones every week.

Prospectus Time Table

Member:	Week 1 (02/27-03/03)	Week 2 (03/06-03/10) + Spring Break (03/13-03/17)	Week 3 (03/20-03/24)
Khoa	☑ ~~Worked on Prospectus~~ ☑ ~~Decided which data types to use for back end parts~~ ☑ ~~Figure out the panel systems and switch ints~~	☑ ~~Start programming the window and panels classes~~ ☑ ~~Finish Prospectus~~ ☑ ~~Work on powerpoint presentation~~ ☑ ~~Develop UML for front end and back end~~	☑ ~~Final touches prospectus and preparation of presentation~~ ☑ ~~Rehearse for the presentation~~
David	☑ ~~Worked on Prospectus~~ ☑ ~~Developed the gamestate class and how different classes would work together~~	☑ ~~Start programming the gameState class~~ ☑ ~~Finish Prospectus~~ ☑ ~~Develop UML~~ ☑ ~~Work on~~	☑ ~~Final touches prospectus and preparation of presentation~~ ☑ ~~Rehearse for the presentation~~

	~~in that class~~	~~polishing the prospectus~~	
Vlad	☑ ~~Worked on Prospectus~~ ☑ ~~Compiled assets~~ ☑ ~~Think of map systems~~	☐ Start programming the tile ☑ ~~Finish Prospectus~~ ☑ ~~Work on powerpoint presentation~~ ☑ ~~Develop UML~~	☐ Final touches prospectus and preparation of presentation ☐ Print out and distribute the prospectus for the presentation
Krish	☑ ~~Worked on Prospectus~~ ☑ ~~Started working on algorithms~~ ☑ ~~Think of map systems~~	☑ ~~Start programming the Player class~~ ☑ ~~Finish Prospectus~~ ☑ ~~Develop UML for frontend and back-end~~	☐ Final touches prospectus and preparation of presentation ☐ Rehearse for the presentation
Milestones achieved:	**1**	**1, 2**	**1, 2, 3**

Programming Time Table

Member:	Week 1 (03/27-03/31)	Week 2 (04/03-04/07)	Week 3 (04/10-04/14)
Khoa	☐ Start Map System - instantiate graphics for background and maps ☐ Start and finish Player System ☐ Upload rudimentary UI images	☐ Continue working on Map System ☐ Start on MainMenuPanel and associated graphics functions ☐ Code case for when there is no available spot for settlement	☐ Finish Map System ☐ Begin scoring system in GameState ☐ Complete all Panel Classes and all graphics functions ☐ Show scoring ☐ Being working on paddock and barn
David	☐ Start Map System - work on back end map storage ☐ Finish Tile Parent class ☐ Upload rudimentary UI images	☐ Continue working on Map System ☐ Start on InstructionPanel and associated graphics functions ☐ Code case for when there is no adjacent tile of the type drawn	☐ Begin scoring system in GameState ☐ Complete all Panel Classes and all graphics functions ☐ Begin working on oasis and tower ☐ Integrate graphics with extra actions
Vlad	☐ Start Map System - finish graphics for boards ☐ Create basic UI graphics	☐ Continue working on Map System ☐ Start on GamePanel and associated graphics functions ☐ Begin progress on rest of the Board Logic	☐ Animate settlement placement ☐ Complete all Panel Classes and all graphics functions ☐ Begin oracle and farm
Krish	☐ Start Map System - work	☐ Continue working on Map System	☐ Finish Map System

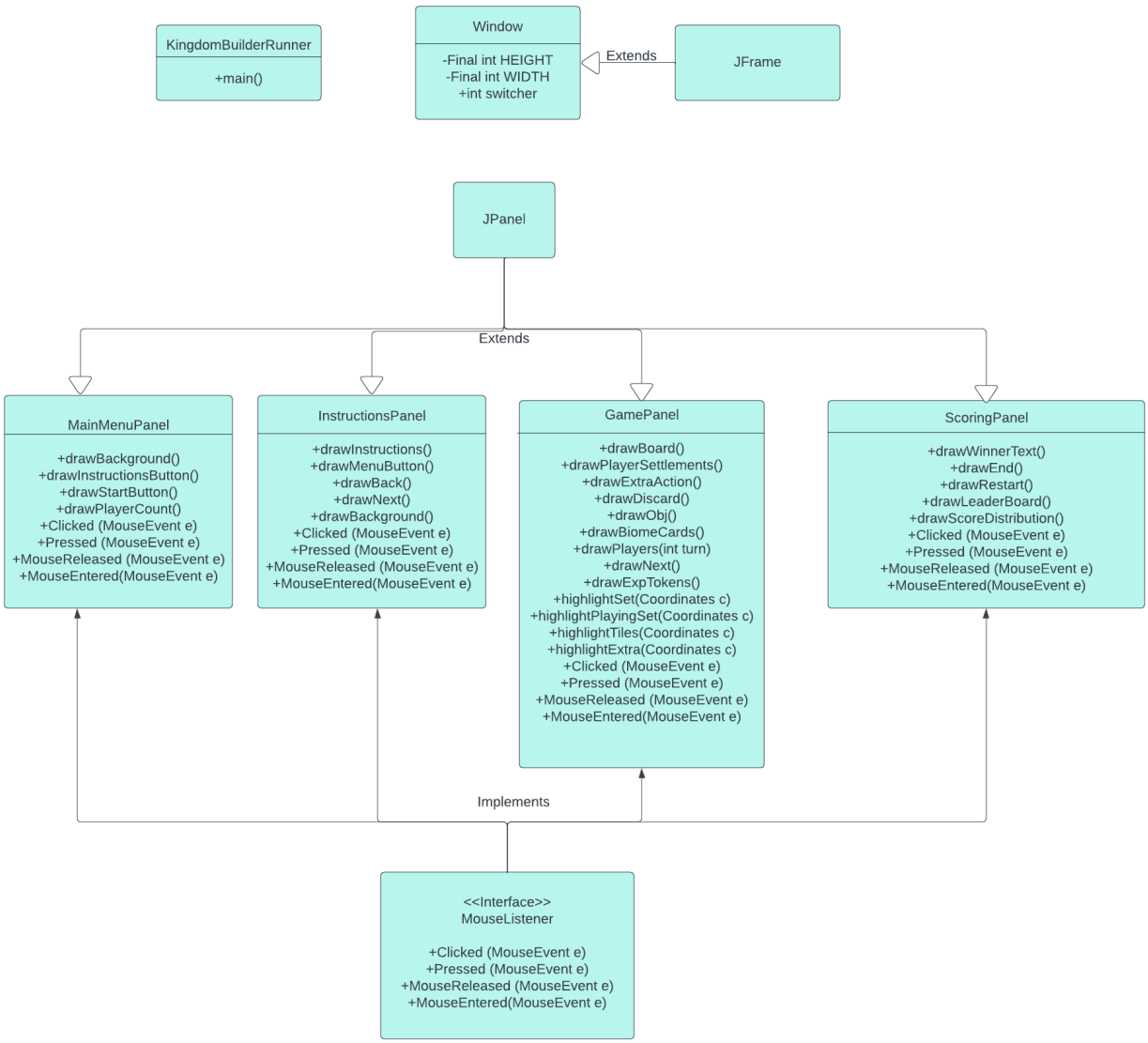
	on combining 4 boards together with randomness ☐ Upload rudimentary UI images	☐ Start on ScoringPanel and associated graphics functions ☐ Integrate mouse clicks with placing settlements	☐ Complete all Panel Classes and all graphics functions ☐ Begin working on the port and tavern
Milestones achieved:	**4, 5**	**5**	**5**

Member:	Week 4 (04/17-04/21)	Week 5 (04/24-04/28)	Week 6 (05/01-05/05)
Khoa	☐ Finalize scoring system ☐ Finalize GameState class ☐ Finalize Player logic ☐ Finalize the UI and user input graphics ☐ Begin testing for bugs	☐ Finish game states ☐ Begin testing the entire game ☐ Program/Complete any extraneous edge cases	☐ Test test cases and scoring via referring to the algorithm ☐ Look for any major and minor bugs ☐ Polish graphics, if necessary ☐ Prepare the final jar for submission
David	☐ Finalize special cases ☐ Finalize Object classes ☐ Finalize Player logic ☐ Finalize the UI and user input	☐ Finalize the loop for the game ☐ Begin testing the entire game ☐ Program/Complete any extraneous edge cases	☐ Test test cases and board creation via referring to the algorithm ☐ Look for any major and minor bugs ☐ Polish graphics, if necessary
Vlad	☐ Finalize extra actions ☐ Finalize animations and UI design ☐ Finalize Player logic	☐ Test special cases for each turn ☐ Begin testing the entire game ☐ Test extra actions and extra action token graphics	☐ Test test cases and turn order via referring to the algorithm ☐ Look for any major and minor bugs
Krish	☐ Finalize settlement placement ☐ Finalize Board design ☐ Finalize Player	☐ Finalize the loop for the game and finish game states ☐ Begin testing the entire game	☐ Test test cases and turns via referring to the algorithm ☐ Look for any major and minor

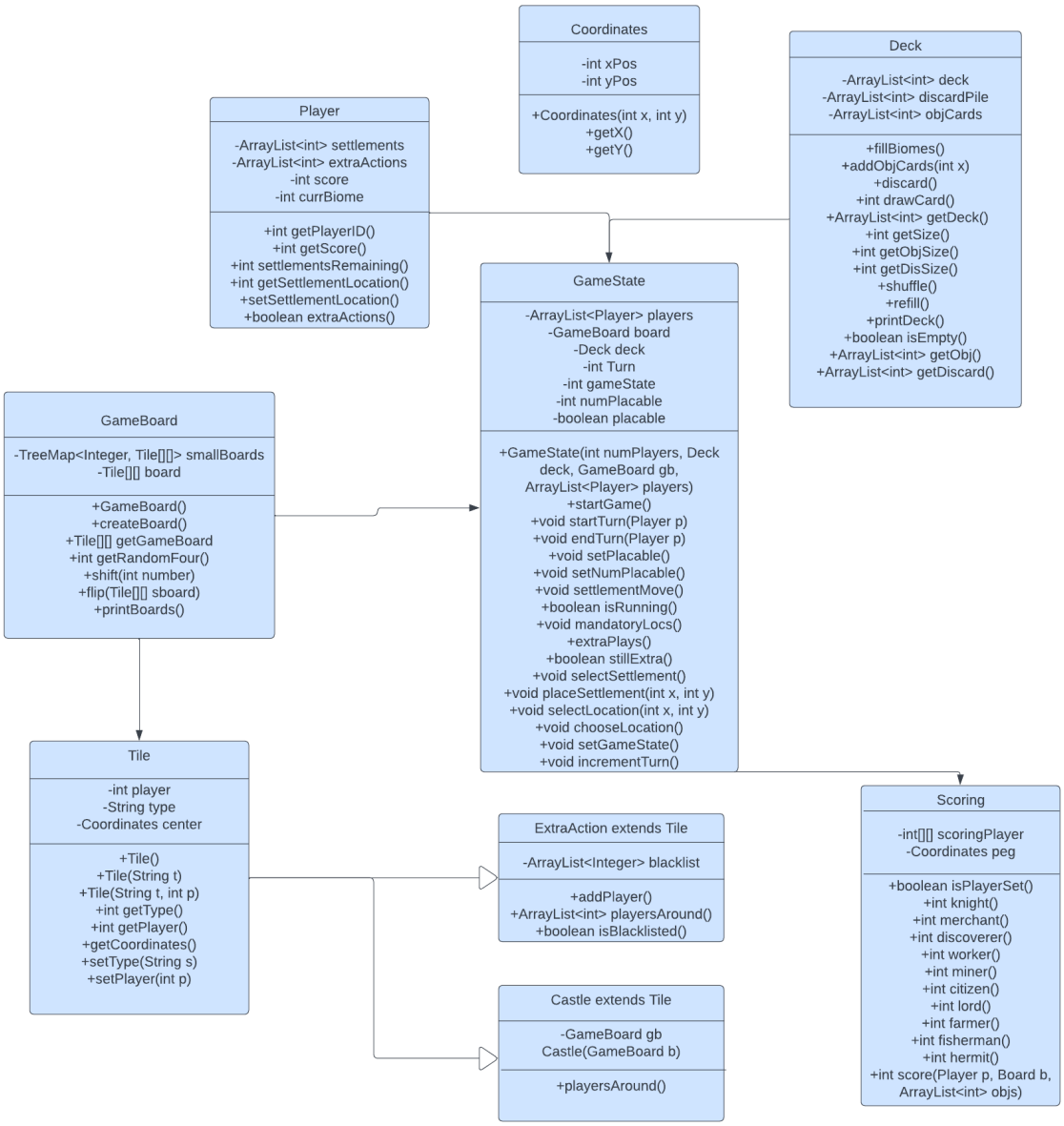
	logic ☐ Finalize the UI and user input	☐ Program/Complete any extraneous edge cases	bugs ☐ Polish graphics, if necessary
Milestones achieved:	**6, 7**	**6, 7**	**8, 9**

UML

Front end



Back end


Coordinates
-int xPos
-int yPos
+Coordinates(int x, int y)
+getX()
+getY()
Deck
-ArrayList<int> deck
-ArrayList<int> discardPile
-ArrayList<int> objCards
+fillBiomes()
+addObjCards(int x)
+discard()
+int drawCard()
+ArrayList<int> getDeck()
+int getSize()
+int getObjSize()
+int getDisSize()
+shuffle()
+refill()
+printDeck()
+boolean isEmpty()
+ArrayList<int> getObj()
+ArrayList<int> getDiscard()
Player
-ArrayList<int> settlements
-ArrayList<int> extraActions
-int score
-int currBiome
+int getPlayerID()
+int getScore()
+int settlementsRemaining()
+int getSettlementLocation()
+setSettlementLocation()
+boolean extraActions()
GameState
-ArrayList<Player> players
-GameBoard board
-Deck deck
-int Turn
-int gameState
-int numPlacable
-boolean placable
+GameState(int numPlayers, Deck deck, GameBoard gb, ArrayList<Player> players)
+startGame()
+void startTurn(Player p)
+void endTurn(Player p)
+void setPlacable()
+void setNumPlacable()
+void settlementMove()
+boolean isRunning()
+void mandatoryLocs()
+extraPlays()
+boolean stillExtra()
+void selectSettlement()
+void placeSettlement(int x, int y)
+void selectLocation(int x, int y)
+void chooseLocation()
+void setGameState()
+void incrementTurn()
GameBoard
-TreeMap<Integer, Tile[][]> smallBoards
-Tile[][] board
+GameBoard()
+createBoard()
+Tile[][] getGameBoard
+int getRandomFour()
+shift(int number)
+flip(Tile[][] sboard)
+printBoards()
Tile
-int player
-String type
-Coordinates center
+Tile()
+Tile(String t)
+Tile(String t, int p)
+int getType()
+int getPlayer()
+getCoordinates()
+setType(String s)
+setPlayer(int p)
ExtraAction extends Tile
-ArrayList<Integer> blacklist
+addPlayer()
+ArrayList<int> playersAround()
+boolean isBlacklisted()
Castle extends Tile
-GameBoard gb
Castle(GameBoard b)
+playersAround()
Scoring
-int[][] scoringPlayer
-Coordinates peg
+boolean isPlayerSet()
+int knight()
+int merchant()
+int discoverer()
+int worker()
+int miner()
+int citizen()
+int lord()
+int farmer()
+int fisherman()
+int hermit()
+int score(Player p, Board b, ArrayList<int> objs)

UML Discussion

Front end

Our UML was designed with simplicity in mind. We plan on utilizing a frame that has its panels switched when the gamestate changes. We can switch those panels through the gamestate integer and using switch. The different panels are planned to show different portions of the game. For instance, the instructions panel will only be utilized when the player requests to see the instructions. Each panel will extend JPanel and implement MouseListener (the methods required are not shown in the UML).

Back end

Our game logic will consist of 9 classes. The coordinates class will link the different elements and items of our back end and position them accordingly in our front end. In a way, it acts as the middleman for both classes. When the round starts, all of our objects will be available with the exception of the Scoring class, which would only run after the round has ended. Our tile class consists of all the different types, including the extra action tiles and the castle tiles. As for our cards, we decided to make it short and sweet by using ints for them.

Detailed Discussion

Player

`-ArrayList<int> settlements`
Holds all the 40 settlements of each player
`-ArrayList<int> extraActions`

Holds all the extra action tiles that each player possesses
`-int score`
Stores the score of the player
`+int getScore()`
Returns the score of the player
`+int settlementsRemaining()`
Returns how many settlements the player still has
`+Coordinates getSettlementLocation()`
Returns the position of the settlement location
`+boolean extraActions()`
Returns true if the player still have any extra actions left (doesnt count if the player just got the tile this round) and false if not

Deck

`-ArrayList<int> deck`
Holds the cards (ints) in order, needs to be shuffled
`-ArrayList<int> discardPile`
Holds the discarded cards that might be used to refill later
`+int drawcard`
Returns a card (int) and discards it from the deck
`+ArrayList<int> getDeck()`
Returns the arraylist for the deck
`+int getSize()`
Returns an int that's the size of the deck
`+shuffle()`
Shuffles the deck
`+refill()`
Refills the deck with cards, does not shuffle them

GameBoard

-TreeMap<Integer, Tile[][]> smallBoards

Holds all the small boards in matrix form that will be used to make the game board

-Tile[][] board

A matrix of tiles that is representative of what a small board would like, can be flipped to make the big board

+createBoard()

Creates the big board using 4 random small boards that could be flipped

+Tile[][] getGameBoard

Returns the big gameboard

+int getRandomFour()

Generates 4 random numbers from 1-8 to be used to select the small boards that will be merged to make the game board

+shift(int num)

Shifts the board by one tile when flipped

+reverse()

Takes in a Tile matrix as an input and flips it vertically, making the first row the last, etc.

+printBoards()

Prints out a numerical representation of the boards to be used for bug testing

Tile

-int player

Stores an integer representing the player that has a settlement on the tile (-1 if the tile is not occupied)

-String type

Stores the type of the tile, with numbers for the normal tiles and letters for the extra action tiles

-Coordinates center

Stores the x and y positions of the center of the hexagon for use in click detection

`+getType()`

Returns the type of the tile

`+getPlayer()`

Returns the player that occupies the tile

`+getCoordinates()`

Returns the coordinates object for the center of the tile

`+setType(String s)`

Sets the type of a tile to a given s

`+setPlayer(int p)`

Sets the int player to the new value p to change ownership of the tile

ExtraAction

`-ArrayList<int> blacklist`

Holds all the players that cannot pick up another extra action tile because they already have one of the same kind from previous turns

`+addPlayer()`

Adds a player to the blacklist for an extra action tile

`+ArrayList<int> playersAround()`

Returns an arraylist of all the players surrounding that extra action tile on the board

`+boolean isBlacklisted()`

Returns true if the player is in the blacklist arraylist and false if not found

GameState

-ArrayList<Player> players

Stores all the players in an arrayList

-GameBoard board

Creates the game board and stores it in a private variable

-Deck deck

Creates the deck and stores it in a private variable

-int turn

Holds an integer that shows the Player turn that the game is currently on

-int currentGameState

Stores an integer that represents the current screen being displayed by the program. For example, a gamestate of 1 would represent the mandatory action phase awaiting the player input

+gameState (int numPlayers, Deck deck, GameBoard gb, ArrayList<Player> players)

Creates a new gamestate object that is stored in the JFrame class. This allows the gamestate to remain the same even as the panels change.

+setGameState()

Sets the gameState int

+incrementTurn()

Adds 1 to the turn int

Scoring

+int knight()

Returns the int of the score that the player earned from the knight kingdom builder card

+int merchant()

Returns the int of the score that the player earned from the merchant kingdom builder card

+int discoverer()

Returns the int of the score that the player earned from the discoverer kingdom builder card

`+int worker()`

Returns the int of the score that the player earned from the worker kingdom builder card

`+int miner()`

Returns the int of the score that the player earned from the miner kingdom builder card

`+int citizen()`

Returns the int of the score that the player earned from the citizen kingdom builder card

`+int lord()`

Returns the int of the score that the player earned from the lord kingdom builder card

`+int farmer()`

Returns the int of the score that the player earned from the farmer kingdom builder card

`+int fisherman()`

Returns the int of the score that the player earned from the fisherman kingdom builder card

`+int hermit()`

Returns the int of the score that the player earned from the hermit kingdom builder card

Algorithms

Round of Play Algorithm:

1. Start turn
2. Draw one random card from the deck class

    a. If the new length of the deck is 0, transfer used cards to the deck and shuffle
3. Set an integer numPlacable to 0 which tracks the number of tiles that can be placed on
4. Traverse the all of the tiles of the board
    a. If the card and tile are the same type AND there is nothing on the tile:
        i. If there is a settlement from the current player adjacent to the tile:
            1. Set the tile to placeable
            2. Add 1 to numPlacable
5. Choose either 6 or 7 to run first, the run the other
6. Repeat 3 times:
    a. If numPlacable == 0:
        i. Traverse the board
            1. If the card and tile are of the same type AND there is nothing on the tile:
                a. Set the tile to placable
    b. Highlight all placable hexagons
    c. Wait for a user input of a mouse click on hexagon (the index of clicked hexagon can be found using the code below)
    d. Get the tile that was clicked
    e. Set the tile to be occupied with a settlement of the current player.
    f. If this is not the 3rd repeat:
        i. Subtract one from numPlacable
        ii. Check all tiles adjacent to the tile that was just updated:
            1. If the tile is of the same type as the card:
                a. Set the tile to placable
                b. Increase numPlaced by one

2. If the tile is an extra action tile AND this player is not on the blacklist of the tile:
    a. Give the player one of the extra actions from the tile
    b. Add the player to the blacklist
    c. Set that extra action to used

7. Check if the current player has any extra actions:
    a. If they do:
        i. Repeat n times, where n is the number of extra actions available to be used:
            1. Await a user input:
                a. If they click the “End” button
                    i. Jump to end turn
                b. If an extra action is clicked:
                    i. Determine whether the action moves a settlement or places a new one
                c. Else
                    i. Do nothing and wait for user input
            2. If the extra action moves a settlement:
                a. Traverse the board and find all tiles with a settlement
                b. Highlight all selectable tiles
                c. Await a user input on which settlement to move
                d. Highlight all tiles where the settlement can be moved to
                e. Await input of the location to move the settlement to
                f. Move the settlement
            3. Traverse the board and find all places that the extra actions can modify
            4. Highlight all of these tiles

5. Await a user input of (if applicable) which settlement to move and where to place/move the settlement

8. End turn
    a. Draw a new terrain card
    b. Set all tiles to not placeable
    c. Await user input to end the turn.

Kingdom Builder Card Algorithms:

Fisherman:

1. Repeat for all players:
    a. Let tempScore be the amount of gold the player has.
    b. Traverse all tiles of the board:
        i. If the tile has a settlement then check the tiles around the settlement:
            1. If there is a water tile around the settlement:
                a. Increment tempScore by one
    c. Add tempScore to the player's score

Discoverers:

1. Repeat for all players:
    a. Let tempScore be the amount of gold the player has
    b. Traverse all of the rows of the matrix of the board:
        i. If the row contains a settlement of the player:
            1. Increment tempScore by one
    c. Add tempScore to the player score

Hermits:

1. Create a new matrix of ints, tempBoard, of the same size as the board
2. Traverse the board:
    a. If there is a settlement at the current tile:
        i. Set the corresponding index of tempBoard to the index of the player
3. Let tempScore be 0
4. Repeat for all players:
    a. Traverse the tempBoard:
        i. If the tile has a settlement of the current player:
            1. Set the index to 0
            2. Check all adjacent tiles:
                a. If they are also occupied by a settlement of the player
                b. Recursively call this section again
            3. Increment tempScore by one
5. Add tempScore to the player's score

Citizens:

1. Create tempBoard (see Hermits)
2. Let tempScore be 0 and max be -1
3. Repeat for all players:
    a. Traverse the tempBoard:
        i. If the tile has a settlement of the current player:
            1. Set the index to 0
            2. Check all adjacent tiles:
                a. If they are also occupied by a settlement of the player
                b. Increment tempScore by one
                c. Recursively call this section again
        ii. If tempScore is larger than max
            1. Set max to tempScore

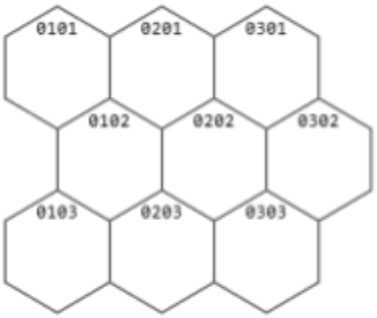
4. Add max/2 to the player's score

Merchants:

1. Create tempBoard (see Hermits)
2. Let tempScore be 0
3. For each player:
    a. Traverse the tempBoard:
        i. If the tile is a extra action tile:
            1. For each tile around the extra action tile:
                a. Set an integer t to the player that occupies the tile
                b. A: If the player occupying the tile is not t and the tile is not an extra action tile, return
                c. If the tile is an extra action tile, add 1 to tempScore
                d. Delete the tile at the index
                e. Check each tile around the tile:
                    i. Recursively run from A to here
    b. Add tempScore*4 to the player's score

Finding the Clicked Hexagon:

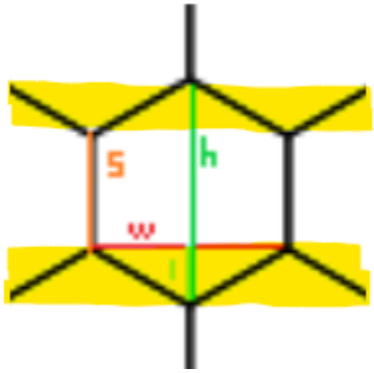
We can map the tessellated hexagons to a matrix in the following way:



0101	null	0201	null	0301	null
null	0102	null	0202	null	0302
0103	null	0203	null	0303	null

We can find the hexagons surrounding a specific hex with the following indices, where r and c are the indices of the hex:
Left - [r][c-2]
Top Left - [r-1][c-1]
Top Right - [r-1][c+1]
Right - [r][c+2]
Bottom Right - [r+1][c+1]
Bottom Left - [r+1][c-1]

Let *w* be the width of a hexagon and *h* be the height of the hexagon. Since we know that the tiles are regular, *w* is $\sqrt{3} * s$ and *h* is *2s*. Let x_0 be the offset from the edge of the screen on the left side and y_0 be the distance from the edge of the screen to the top of the first hexagon. Using this equation, we can find the x index (i_x)of the hexagon:



1. Let an integer yindex equal the floor of 2y/s-1
2. Let an integer xindex be -1
3. Subtract x_0 from x and y_0 from y to normalize the coordinates such that the top left of the board is (0, 0)
4. If the y index mod (%) 3 is 2 (the click was in the highlighted region):
    a. Set yindex to the floor of yindex/3
    b. Set yindex to ((yindex + 1) * 10 + yindex)*1000 //this indicates which two rows the click was between

5. Otherwise, the click was not in the highlighted region, so we can find the indices easily:
    a. Set yindex to the floor of yindex/3
    b. Let an integer f be the floor of (yindex % 2 * w / 2)
    c. Set xindex to the floor of (x-f) / w
    d. End the script for this branch
6. If the xindex is still below 0 (this means that the click was in the highlighted region):
    a. Traverse every tile in the rows specified by the first two digits of yindex
        i. Find the shortest distance from the center of the tile to the click
    b. The tile with the shortest distance is the tile clicked
    c. Set x index to the x index of the tile above
    d. Set y index to the y index of the tile above

Boards:

To create the back end boards, we wrote a text file with the sequences of each board. By reading these files into the board, we can store the sequences of tiles for the boards in the back end and merge the smaller boards into one. We use letters to represent extra action tiles because they are easy to detect with compareTo() since letters are greater than all numbers. An example of text used for one board is shown below:

2 2 2 2 7 7 0 7 3 3
2 7 2 2 1 0 7 7 7 3
1 1 2 1 1 1 0 0 6 7
4 1 1 1 6 D 0 6 7 7
4 4 4 4 1 6 0 6 3 3
4 3 4 4 4 6 6 3 0 3
4 4 3 4 4 6 1 5 0 3
3 3 D 4 6 1 1 1 0 0
4 3 6 6 6 2 2 1 0 0
4 3 3 6 2 2 2 0 0 0

Key:

0 is the grass
1 is the flower field
2 is the forest
3 is the canyon
4 is the desert
5 is the castle
6 is the water
7 is the mountain
8 is the oracle
9 is the farm
D is the tower

Testing Other Algorithms

Turn:

- Verify that the player cannot place on a tile with another player's settlement.
- Test if the player can only place on tiles that are of the same type as the card drawn.
- Ensure that the player must place adjacent to one of their settlements.
- Make sure that each player has only 40 settlements to place.
- Ensure that if the player runs out of settlements in the middle of a turn, their turn ends.
- Test if the player must start their turn by placing 3 settlements or using their extra actions.
- Ensure that players cannot use extra actions on the turn they receive them.
- Verify that additional settlements can be moved and placed using extra action tiles.
- Guarantee that settlements cannot be placed on water, extra actions, castles, or mountains, but can be moved on to water using ports.
- When settlements are moved, moved settlements must be placed next to existing tiles.

- If there is not a place for the new settlement, then the player may place anywhere that is of the type of the current card.
- If chosen, the player has the option to not use the extra actions, but must place the 3 settlements.
- When a player runs out of settlements, the players after them may have one more turn.
- Ensure that the hex that is clicked is the hex that is selected by the game logic.
- If the click is not within the highlighted region, the program does not waste time on checking the distance to each center.
- Verify that the tiles that can be placed on are highlighted.
- Ensure that the fill polyline function for highlighting uses a color with an alpha of 150, so the highlight is translucent.
- Guarantee that the turn shows the correct number of settlements that each player has left.
- Check that the graphics correctly show whose turn it is.
- Ensure that the hitbox of all tiles is a regular hexagon, so clicks in the edges also count towards the correct hex.

Cards:

- When the deck of terrain cards is empty, shuffle the discarded cards to make the new deck such that the new deck has exactly 22 cards before being drawn from.
- When the game starts, 3 cards are selected from the kingdom builder cards.
- Ensure that exactly 3 kingdom builder cards are drawn at the start of the game and that they remain in effect for the whole game.
- Verify that both decks are shuffled by random selection of cards.

- Ensure that discoverers earn only one point for only one settlement in each row.
- Make sure that fishermen earn one point maximum if a settlement is adjacent to a water tile.
- When merchants are scoring, extra actions tiles are counted only once if connected and cannot be connected by others.
- If merchants connect tiles, chains of settlements pass through extra action tiles.
- Ensure that knight counts the highest number of settlements in only one row and adds two gold for each settlement.
- Verify that the correct number of cards is displayed in the graphics.
- Ensure that all three Kingdom Builder Cards in play are visible at all times.
- Verify that four terrain cards are drawn at the start of the game, one for each player.
- Guarantee that the terrain cards

Board:

- Verify that 4 random boards are selected when the game starts.
- Guarantee that each board can be in all 4 positions (bottom left, bottom right, top left, top right) and either orientation (normal or flipped) with equal probabilities.
- Ensure that the boards are concatenated such that the final board is 20 hexes by 20.
- Verify that the board cannot be changed throughout the game other than the placement of settlements.
- Ensure that the graphics for the board are oriented the same way as the back end.
- Check that the terrain is correctly read from the text files.
- Ensure that there is one null between each matrix index.

- Verify that invalid indices are never called in code.

End of Game:

- At the end of the game, ties in gold result in all tied players winning.
- Guarantee that the final board can be seen at the end of the game.
- Verify that score contributions from Kingdom Builder Cards are displayed separately.
- Ensure that the graphics properly display the Kingdom Builder Cards.
- Ensure that graphics properly display the correct gold count.
- Check if gold is given for each castle hex, not settlement next to a castle.
- Test that all players get the same number of turns, with players after the player that runs out getting a turn after the end.

Other:

- Verify that the instructions are properly able to be traversed.
- Guarantee that all of the pages of the instructions are visible.
- Ensure that the title page displays the start button and instructions button.
- Verify that the title page changes panels when buttons are clicked.
- Make sure that the First Player token is displayed next to the player that went first.

GUI


KINGDOM
BUILDER
Start Game
Game Rules
Number Of Players
2
3
4


Discard Pile
Card Deck
Card In Play:
Player:1
Settlements Left: 28
Gold: 13
Player:3
Settlements Left: 28
Gold: 16
Player:4
Settlements Left: 28
Gold: 14
FARMERS
Build settlements in all sectors
3 gold for each of your own settlements in that sector with the fewest of your own settlements.
WORKERS
Build settlements next to location or castle hexes
1 gold for each of your own settlements built adjacent to a location or castle hex.
FISHERMEN
Build settlements on the waterfront
1 gold for each of your own settlements built adjacent to one or more water hexes.
Current Player:2
Settelments Left: 25
Gold: 12
Extra Tokens:
NEXT PLAYER


tra Tokens:
Go back to scoreboard


WINNER PLAYER 2
Gold Count
Player:1 23
Player:2 32
Player:3 16
Player:4 27
Return to main menu
Return to board

GUI Discussion

The general priority in creating the GUI for all of the possible screens was simplicity and accessibility. Throughout the entire game, the player should be able to make whatever possible decisions presented spontaneously without having to extensively search through a tedious mess of buttons and screens. For that reason, every major visual aspect of the game such as the scores, the player numbers, etc… all have an explicit, bolded label. Additionally, the most important actions a player could possibly make are made resoundingly clear, either through distinct coloration such as the ‘next player’ button or through a large indicator on the screen. With that in mind, let’s explain the specific screens every player will likely encounter in their first few playings of the game.

The first interface is the game’s beginning screen. Besides simply showing the button to begin the game, the first button below

provides a detailed explanation of the game’s rules. Additionally, the second button provides a simple way to toggle between two to four players, as that is the range of the number of players in which Kingdom Builder can be played. Toggling can be achieved either by pressing or by dragging.

The second critical interface detailed in this prospectus is the screen the player will likely see throughout the entirety of his gameplay. This screen is also the screen that has the most possible functions and actions. Because of this, it is formatted extremely neatly, without any visible overlap. The card in play is presented at the center of the top half. The current player will be able to see their own score and number of settlements, as well as the other players’ scores and settlements. The explanation tokens are neatly organized vertically on the right-hand side, and the extra tokens that have not been played yet are at the bottom. Other important aspects of the front end, such as the scoring cards and the discard pile are also displayed.

The third and fourth images display the ability for players to flip between the scoreboard and the board at the game’s conclusion. This will allow players to count their points manually and additionally view the final board. As shown, the red ‘next player’ will be automatically modified to ‘go back to scoreboard’ to reflect that the game is over. The respective buttons are also shown on the scoreboard.

The final interface shows the pages of the instructions that can be accessed on the home screen. To flip through the pages, simply press the respective arrows on the sides.

Final Discussion

As we dive into the programming process, there will undoubtedly be unforeseen issues that may not be immediately noticed or resolved given the complexity of Kingdom Builder. However, given the extensive and exhaustive nature of the algorithmic testing, this should not be a profoundly problematic issue. Exhaustive testing, especially towards the completion of the project, should be sufficient to add any necessary checkbacks or prevention measures to encompass all edge cases and potentially hidden bugs.

Another major issue that may arise is the various IDEs that the programmers are utilizing. To best mitigate the potential consequences of this, we've resolved to centralize all of our code on GitHub, which is especially beneficial, given GitHub's automatic ability to be tied to different IDEs. GitHub is also particularly essential in that it indicates when code was last edited, and for what purpose it was changed. Besides GitHub, we will work in a close and efficient manner to prevent any overlapping or overworking on specific aspects of the game. Overall, GitHub is an easy and efficient way to share and centralize code that doesn't generate an immense amount of confusion or inconvenience that other methods may otherwise cause.

Overall, this game will require extensive thought and adaptation while programming, but we've engaged in several measures to prevent or mitigate the risk or harms of this game.