Adam Barson, Jose Chan, Ramy Koudsi, CJ Pecor

Deliverable 2

Requirements for Blokus System

Essential:

-Create Gameboard object - medium => Adam
-Create square objects to populate gameboard -medium =>Adam
-Creating pieces' objects - Hard => CJ
-Checking location - medium => CJ:

- Checking piece that piece is in bounds of board
- Checking that piece is not on top of other piece

-Rotating pieces – hard =>Ramy
-Moving pieces - medium => Jose
-Placing pieces - hard:

- Check piece adjacent method - medium => Ramy
- Check piece diagonals method - medium = >Adam
- Checking bounds of board - medium => Jose
- Checking whether piece already exists in this location - medium => Jose

-2 player classes - hard => All team members
-Let user switch between which piece they choose - medium => Jose
-Keep track of and calculate score method - easy => Ramy

Need to have:

-Have piece change color if allowed/not allowed to place there. - medium => Adam
-Running score calculator - easy => CJ

Nice to have:

-Menu that lets you pick theme - easy => Ramy
-Use mouse - hard => Jose

Specifications

Blokus: Iteration 1

Objectives:

- Create all unique 21 subclasses of BlokusPiece
- Get rotate and move methods of BlokusPiece to work
- Check boundaries of board

BlokusPiece
#pieces: BlokusSquare[] #relativeX: int[] #relativeY: int[] #board: BlokusBoard #rotatePiece: boolean
+<constructor>BlokusPiece(board: BlokusBoard) +setRelatives(): void +moveToLocation(x: int, y: int): void +isRotatePiece(): Boolean +rotateLeft(): void +rotateRight(): void +moveUp(): void +moveDown(): void +moveLeft(): void +moveRight(): void +hasAdjacent(): Boolean +hasDiagonal(): Boolean +placePiece(); void

BlokusPiece is an abstract class, which holds an array of 1 or more BlokusSquares. One origin square is placed according to x and y coordinates on the board, and the other pieces are placed according to relative coordinates to the first piece. This allows rotation to take place very easily (implementation already in BlokusPiece). Checking the board coordinates should also be done in terms of the piece's relative values. Relative coordinates are set by each of BlokusPiece's subclasses. Each subclass also sets the value of rotatePiece, which dictates whether or not the piece is allowed to rotate (the Square, for example, shouldn't be allowed to rotate).

Once rotation and move methods work, the next step is to get boundaries of the board to be checked when moving or rotating the piece. So, the piece should not move or rotate if it would land the piece out of bounds. More methods might need to be added to BlokusPiece for this.

To test your code, you can create instances of BlokusPiece in BlokusGame.

Blokus: Iteration 2

Objectives:

- Get place piece logic to work
    - Should not place piece on other placed pieces.
    - Get hasAdjacent and hasDiagonal to work, so that pieces can only be placed if they are diagonal but not adjacent to another piece.
- Add an "inventory" area to the board for remaining pieces

Get the one piece to be added to the board by pressing enter, and spawn a new piece. Pieces should not be able to be added where there already exists a piece.

Have the ability to switch between currently unused pieces in the player's inventory. Piece's that have already been placed onto the board should not be available for use again.

hasAdjacent and hasDiagonal should be used to determine if a piece can actually be placed-- that is, if hasDiagonal is equal to true, and hasAdjacent is false. In the future, we'll have to check to make sure these methods support player colors, too.

Blockus: Iteration 3

Objectives:

- Design player class.
    - Players should have a list of shapes that can still be placed. The on screen inventory should reflect changes to this list.
    - Get player turns to work
        - One player places a piece down, then the next player places a piece down
- Get game to end when no pieces are remaining or board is full
- Design an opponent class once basic turn functionality works
    - First, get computer to do brute force turns

Blokus: Final Iteration

Objectives: (if time permits)

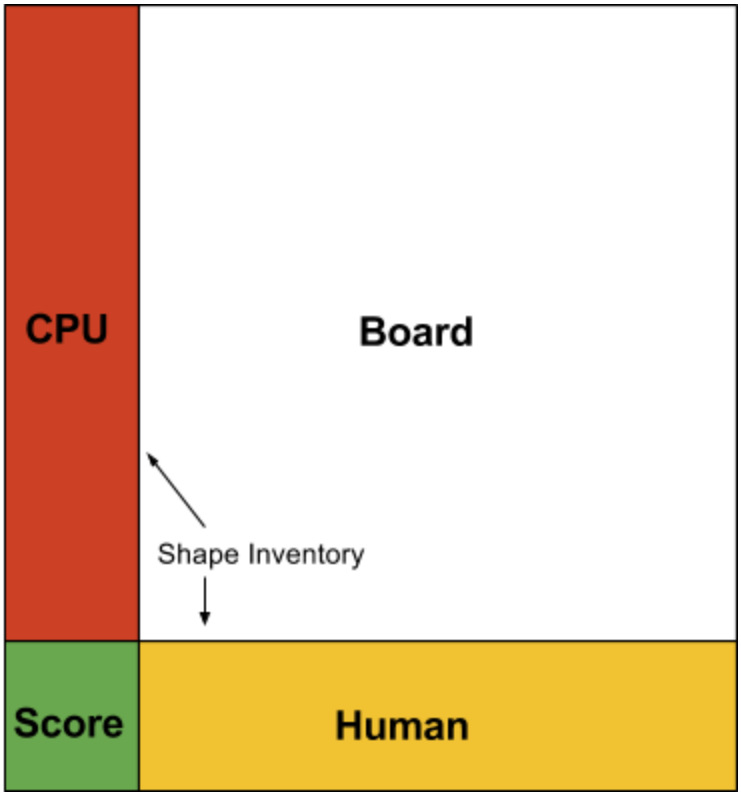
- Create main menu
- Get computer to play smart
    - Add difficulty options
- Add animations, extra flare

GUIs

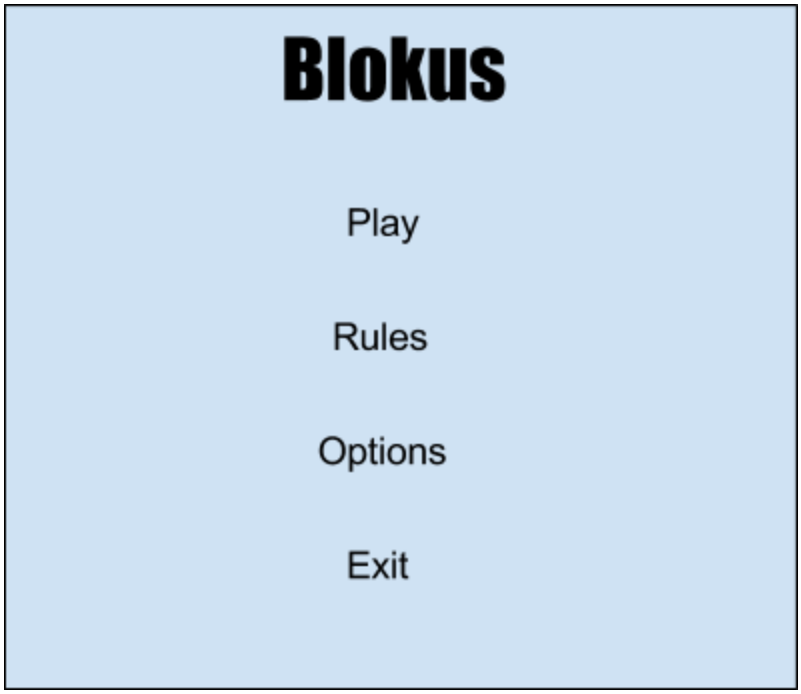
- Game Board:
    - Window with a 20x20 grid should be centered in the screen.
    - Pieces will appear on this board as they are selected.
- Inventory:
    - Coming off the left and bottom of the main board will be the inventory sections for each player
    - Each should be look like a rectangle extending the main board, just minus the grid.
    - Each inventory section will display the collection of remaining pieces for its respective player.
    - The square of empty space that will be formed through doing this (see first picture below) will be used to display a running score (optional).
    - The bottom inventory section will be labeled something like “Your Pieces.”
    - The left inventory section will be labeled something like “Opponent’s Pieces.”
    - The pieces listed in these inventory section will disappear as they are used and placed into the game board.
- Pieces:
    - Game pieces will be displayed in either the players’ inventories or the game board, depending on the situation.
    - Each piece will be one of four colors (red, blue, green, yellow) with each color having its own version of a piece (ex: the 2x2 square piece will exist a total of 4 times, for red, blue, green, and yellow).
- Main Menu (reach goal)
    - Display name of game in big letters at the top of the screen.
    - Beneath should be various clickable buttons with the following options:
        - Start: to send the player into the game.
        - Difficulty: to select the difficulty for the AI opponent
        - Exit: to end the application.
    - Have some sort of jaunty theme to decorate it.

Models

Main Board GUI



Main Menu GUI



Game Modules

BlokusSquare
-shape: Rectangle -board: BlokusBoard -x_comp: IntegerProperty -y_comp: IntegerProperty
+<constructor>BlokusSquare(board: BlokusBoard) +setLocation(x: int, y: int) +getX(): int +getY(): int +setColor(): void

BlokusSquare is the foundation of a BlokusPiece. The main component of the BlokusSquare is the Rectangle field (derived from the Javafx Shape class) which is physically added to the board. Through the x_comp and y_comp integer properties, the actual x and y coordinates of the shape can be binded to the size of a square, drastically simplifying the use of x and y coordinates.

BlokusPiece
#pieces: BlokusSquare[] #relativeX: int[] #relativeY: int[] #board: BlokusBoard #rotatePiece: boolean
+<constructor>BlokusPiece(board: BlokusBoard) +setRelatives(): void <abstract> +moveToLocation(x: int, y: int): void +isRotatePiece(): Boolean +rotateLeft(): void +rotateRight(): void +moveUp(): void +moveDown(): void +moveLeft(): void +moveRight(): void +hasAdjacent(): Boolean +hasDiagonal(): Boolean +placePiece(); void

BlokusPiece is an abstract class that defines the behavior of all pieces available to users. Despite having different shapes, pieces share the same common functionality, such as moving and rotating. The shapes of a piece are determined by the pieces' relative coordinate values. There are 21 subclasses that extend BlokusPieces, and must override setRelatives, which sets the relative coordinates of that piece. The relative coordinates move the squares of a piece relative to some origin square, and allow for very easy rotation.

BlokusBoard → Pane
+NUM_SQUARES_X = 20 <static int> +NUM_SQUARES_Y = 20 <static int> +SQUARE_SIZE = 30 <static int> -squares: BlokusSquare[][]
+<constructor>BlokusBoard() +checkBounds(x: int, y: int): boolean +checkLocation(x: int, y: int): boolean +addSquare(square: BlokusSquare): void

BlokusBoard is the main bulk of the screen. It extends Pane from Javafx, which allows the Application to create a screen with it. It will start off as an empty 20x20 grid, which will be populated by various different BlokusPieces. There is an underlying data structure, squares (see above), that does not directly change the GUI, but keeps track of where pieces have been placed, and thus allows for additional board logic to be implemented. It should stay consistent with the reality of what is on the pane.

BlokusPlayer
-pieces: ArrayList<BlokusPiece> -dock: Inventory -activePiece: BlokusPiece -score: int
+<constructor>Player() +getActivePiece(): BlokusPiece +setActivePiece(activePiece: BlokusPiece): void +getScore(): int +setScore(score: int): void +remainingPieces(): int

BlokusPlayer is an object representing a player who is playing Blokus. The data fields are rather intuitive, as each player will have collection of blokus pieces, a score, and an “active piece”, (the piece they are in the process of trying to place), just as a player in real life would.

BlokusGame
-players: BlokusPlayer[] -board: BlokusBoard
+<constructor>BlokusGame(board: BlokusBoard) +play(): void +left(): void +right(): void +up(): void +down(): void +enter(): void

BlokusGame is the controller of the game logic. Within BlokusGame, the game loop is run, and players will switch turns until the game is over. BlokusGame contains an array of type BlokusPlayer, so that in the future, it is possible that more than two players play the game. All keyboard commands are also handled in this object.

BlokusApplication → Application
-board: BlokusBoard -game: BlokusGame
+start(): void +createGrid(): void +setUpKeyPresses(): void +main(): void

BlokusApplication extends Application, and launches the program. It creates a screen using a BlokusBoard object, creates a grid, and sets up keyboard listeners, so that the player may actually interact with the screen. After constructing everything, it calls the play method of BlokusGame, which starts the game.

Inventory → Pane
-board: BlokusBoard -pieces: ArrayList<BlokusPiece>
+setX(): void +setY(): void +removePiece(): void +addPiece(piece: BlokusPiece): void

This class is simply a visual representation of the player's remaining pieces.