Programming Assignment #4: Tetris

Tejas Mehta and Ryan O'Leary
CS314H

Assignment Overview

This assignment asks the programmer to implement a working version of the game Tetris, specifically through the pieces and game board. Tetris is essentially a puzzle game that requires players to race against time in efficiently stacking different shaped blocks, arranging them in such a way that rows are completely filled and then cleared. Players are able to move the current Tetris piece left, right, and down while also being able to rotate them clockwise and counterclockwise. Players can also drop each piece, placing it where it would fall immediately below its current position. If a Tetris piece encounters the floor of the game board or another piece under it, it is then placed and a new piece is spawned. The graphics interface, scoring, and iteration of game states was handled by the JTetris class provided as part of the project, so the job of the programmers was primarily to implement the movement for each Tetris piece, check that each piece was only being moved to valid positions, and clear rows once they were filled.

Goals

Our goals when beginning this assignment were to create a comprehensive version of Tetris so that the numerous possible edge cases arising from wall kicks and the general randomness of the game were accounted for and handled. We hoped to continue building on the testing skills learned during Critters to systematically test each part of our code as well as the multiple different situations we should be accounting for. Additionally, we wanted to learn more about writing efficient AI through implementing the JBrain class. Machine learning and AI are viewed by many as the future of the Computer Science industry and thus we were excited to improve our abilities in this area.

Solution Design

Overview:

To implement a working version of Tetris, it was necessary to design two classes - TetrisPiece and TetrisBoard. TetrisPiece implemented the Piece interface, providing functionality for the methods defined in Piece to represent each piece created in the game board. TetrisBoard implemented the Board interface, containing most of the game logic and defining the behavior of each piece when subjected to user input. The job of TetrisBoard was to store a 2D array of TetrisPieces, manipulating the array based on the actions passed from the player to the move() method which is called every tick by JTetris.

TetrisPiece:

- TetrisPiece((PieceType type):
    - This constructor is designed for creating brand new pieces. It initializes the rotation index to 0. It sets the variable pType using the given pieceType parameter and uses the pieceType to get and store the body. Using this body it calls the getSkirt method to initialize the values of the skirt array. It later gets the min and max xvalues and yvalues in the body to find the piece's width and height which it stores as variables.
- TetrisPiece(PieceType type, int rIndex, Point[] body):
    - This constructor is the same as the above except it uses a parameter to set the rotation index and the body. This constructor is used for the rotation methods to store rotated bodies.
- getType():
    - Returns the pieceType
- getRotationIndex:
    - Returns the rotation index
- clockwisePiece():
    - Increments the rotation index or sets it to be 0 if it is 3. Then calls the rotatePiece method. Returns a new tetrisPiece that uses the second constructor with rotationIndex and body.
- counterclockwisePiece():
    - Decrements the rotation index or sets it to be 3 if it is 0. Then calls the rotatePiece method. Returns a new tetrisPiece that uses the second constructor with rotationIndex and body.
- getWidth():
    - Returns the bodyWidth calculated in the constructor.
- getHeight():
    - Returns the bodyHeight calculated in the constructor.
- getBody():
    - Returns the body array.
- getSkirt():
    - Creates a new array with the width of the bounding box. Fills the array with Integer.MAX_VALUE. Loops through the body array and if the value of skirt at the x value of the current body position is greater than the yValue of the current body position, sets the value of the skirt at the x value of the current body position to be the yValue of the current body position. Basically finds the lower value for each column of the piece. Returns the skirt array.
- equals():
    - Checks if a piece has the same type and rotation index and returns true if it does and false if it doesn't.

Helper Methods:

- rotatePiece():
    - Depending on the pieceType, access the array of rotations for the correct piecetype and sets the body to the value at the given index parameter for each array.
- getBottom():
    - Returns the bottom computed in the constructor.
- getTop():
    - Returns the top computed in the constructor
- getLeft():
    - Returns the left computed in the constructor
- getRight():
    - Returns the right computed in the constructor

TetrisBoard:

- TetrisBoard(int width, int height):
    - Constructor that sets the width and the height using the parameters. Creates a new 2 Piece array, PieceBoard, (Grid) with the width and height dimensions. Also initializes an int array, column heights to store the column heights and row widths to store the row widths.
- move(Action act):
    - Sets lastAction to the given parameter. Uses getBody and getSkirt to store the body and skirt in arrays. Loops through the skirt to get the right and left offset values for each piece/rotation.
    - Switch statement checks for each action.
    - Down:
        - Finds the minimum yPosition in the skirt. If the yPosition of the current piece plus the minimum yPosition of the skirt is greater than 0, then check if any value below the skirt is not null. If any of them are not null, return place(), otherwise set lastResult to success, decrement the position, and return success. If the current piece plus the minimum yPosition of the skirt is less than or equal to 0, return place().
    - Drop:
        - Continuously the same thing as Down until the piece is placed
    - Left
        - Set last result to success. Check for every point in the body if its beyond the left wall or if there is a piece to the left of the current piece. If either is true, set the last result ot out_bounds and return out_bounds. Otherwise if the xPosition plus the left offset is greater than 0, decrement the xposition and return success. Otherwise return out_bounds.
    - Right
        - Same as left except check for right wall, pieces on the right, and increment x instead.

    - Clockwise
        - Check if the piece should be placed and return place() if it should be placed. Create a new tetrisPiece using the same attributes as the current piece. Call clockwisePiece on this new piece. Call the canRotate method to see if the piece can be rotated. If it can, set the current piece to be the new rotated piece. If this piece should be placed, return place(body). Otherwise set the lastresult to be success and return success. If the piece cannot be rotated, create a wallKickArray and assign it the right values depending on whether it is a stick or not. For each wallKick, call testWallKick to see if the wall kick could be implemented. If it can, get the rotationIndex of the current piece and store it, set the current piece to be rotated piece, and call wallKick(). If the wallKicked piece should be placed, return place(), otherwise set last result to success and return success.
    - Counterclockwise
        - Same as Clockwise but use the counterclockwise wallkicks and rotate the object counterclockwise.
- testMove(Action act):
    - Creates a new board with the same width and height. Copies (deep) each and every piece from the original board to the new board. Calls nextPiece with the currentPiece and the current x and y value. Calls move with the parameter act and returns the board.
- getCurrentPiece():
    - Returns the current piece
- getCurrentPiecePosition():
    - Returns the current piece position
- nextPiece():
    - Sets the x and y positions based off of the given parameters. Sets current piece to the given piece parameter.
- equals():
    - Compares the grids of the current board and the given parameter and if they are equal return true otherwise return false.
- getLastResult():
    - Returns last result
- getLastAction():
    - Returns last action
- getRowsCleared():
    - Returns the number of rows cleared
- getWidth():
    - Returns the width of the grid
- getHeight():
    - Returns the height of the grid
- getMaxHeight():

        - Returns the max height based off of what was calculated in place.
    - dropHeight():
        - Loops through the skirt and finds which column and height would be the minimum for the piece. Returns the column height value for that column plus 1.
    - getColumnHeight():
        - Returns the value of the given index in the column height array.
    - getRowWidth():
        - Returns the value of the given index in the row width array.
    - getGrid():
        - Returns the value of the grid at a given x and y.
    - removeRows():
        - Creates a new array with the dimensions of the current grid. Copies each value to the new array from the current grid unless any of them is null. If all of the values were not null, increment the number of deleted rows and set the deleted row index to be the current row.
        - Afterwards remove each row until deletedRows is equal to 0.

Helper Methods:

- getPieceBoard():
    - Returns the reference to the grid
- place():
    - For each point in the body, place it in the appropriate place in the grid using the current xPos and yPos. Create an integer,max value, with the min value for integers. For each point in the body, if the yValue of the point is greater than the max value, update that to be the max value. If the value of column heights at the column is less than the new column with the point, update the column height. Increase the width of the row for that point. After looping through the points compare the max height to the current max height and if its higher, update it. Set the last result to place, call remove rows, and return Place.
- wallKick():
    - Increment the yPosition by the wallkick's y value and increment the x value by the wallkick's x value.
- testWallKick():
    - Perform a deep copy of the body into a new point array. Increment each point by the x and y values of the wall kick. Check if canRotate with the new body returns true, otherwise return false.
- shouldPlace():
    - Checks if any of the values right below the piece body in the grid is not null. If there is one, return true, otherwise return false.
- canRotate():
    - Checks every point in the given body to see if they are within both walls and there is no piece in the place of the locations of each point in the grid. If any of the above conditions are false, return false, otherwise return true.

JBrainTetris:

This class was used as a framework to connect any brain to the GUI. It extends JTetris and overrides a few of the methods but keeps most of the implementation constant. It uses one instance variable, brain, that stores the object of the brain.

Constructor

- In the Constructor, I call the super constructor from JTetris. I then store a new brain reference (depending on the brain) in the brain instance variable. I remove all of the keystrokes that were created in the super constructor by calling unregisterKeyboardAction for each element in registeredKeyStrokes. I created a timer that calls the tick function in the brain for brain's next move every second using the board variable created in JTetris.

Main

- In the main method I call createGUI and pass in a new reference to JBrainTetris.

TetrisBrain: (NotLameBrain)

Our brain took the basic logic of Lame Brain, but introduced enhancements to improve the performance. After calling enumerate options with the currentBoard, I created two new boards rotated clockwise and counterclockwise, and called enumerate options for each one so that the options and firstmoves arrays will have rotated pieces moved every position to the left and right, covering every possibility. I still used getMaxheight as the primary metric to measure how optimal each option was. If the item chosen was rotated, I returned Clockwise or Counterclockwise. I introduced a penalty for a move that would create more "holes" (empty spaces surrounded by blocks) on the board. I also introduced a bonus if the item is a stick and is rotated vertically. I also introduced a bonus to the piece if it was a vertical stick that was being dropped before the maxheight is within four of the top, since vertical sticks are usually much more useful than horizontal sticks. I also prioritized clearing lines by choosing the move that resulted in the highest number of cleared lines. I added 10*the number of lines cleared to the score for each piece. I played around with the weights and bonuses till I reached an optimal solution. The highest score we've reached is 218, with an average of around 85. In ten tests we usually go above 100 2 times and almost never get below 50.

Error Checking:

For each option in move, I conducted thorough error checking to always ensure the following conditions:

1. No part of the piece is beyond the left wall
2. No part of the piece is beyond the right wall
3. No part of the piece goes through another piece
4. A piece is always placed when it is above another piece

5. A piece is always placed when above the floor and never goes through

To prevent null pointer exceptions we always checked if an item is null before accessing values within the object using short circuiting. We also used short circuiting to ensure that we never access array values that are out of bounds.

Evaluation:

Overall, this project proved significantly challenging given the amount of code we were required to implement, as well as the numerous helper methods we chose to create in order to compartmentalize our code and improve the efficiency of our solution. As a result, we were able to extensively test our OOP skills and significantly improve our ability to identify edge cases and unit test with a test harness.

Scope:

Unlike previous projects, our implementation of Tetris relied on Actions passed in from JTetris rather than text file inputs we read in ourselves. This made error checking within each class significantly easier. However, we ended up needing to account for a significant amount of edge cases and thus most of our effort went into structuring TetrisBoard in such a way that pieces would move correctly according to user input while not intersecting other pieces or walls. I believe we achieved this very successfully by creating a switch statement that handled each action as a separate case, returning the correct result at any state and correctly updating the position of the piece in the piece board. As a result, our solution for Tetris successfully executes all piece wall kicks, checks to remove rows every time a piece has been placed, and functions without any discernible bugs. Our board also works for any size board and functions for any valid piece spawn position (within the board).

Quality of Solution:

We designed a very comprehensive solution to Tetris that is able to function for all possible cases and passes a significant amount of the functions at constant time, giving it high performance. We utilized several while and for loops throughout TetrisBoard to iterate through the pieceBoard and body of the piece, but by adding if and return statements throughout these loops we were able to optimize our design by allowing loops to break before iterating N times. We had to use some memory allocation in storing every possible rotation for each Tetris piece, but this tradeoff was beneficial as it enabled us to greatly speed up the clockwise and counterclockwise rotations of each piece. Overall, our total memory allocation for TetrisBoard was fairly low, with only two matrices, a Result, an Action, and several primitives being stored as class variables. We minimized our redundant code by creating helper methods such as removeRows() and testWallKick(), adding readability to our program and efficiently organizing our solution. If we were to try and improve our solution, we would remove and redundant if

statement that could exist within our move() method and try to reduce memory allocation in TetrisBoard by reusing certain variables rather than creating new ones.

Interesting Results:

Most of our results were not very interesting as our board worked as we expected to based off of our previous experiences with tetris. Our brain was quite interesting since as we changed each weight and bonus value, it was cool to see the board prioritize different actions over others.

Problems Encountered:

Initially we faced issues with Board.Action.PLACE not being called at the correct times, resulting in pieces hitting an object and then disappearing. This happened consistently when spamming either left, right, or a rotation, confusing us as we believed that PLACE was being returned at all necessary locations in our code. However, by refactoring our move() method to not alter pieceboard until after the switch statement had completed, we were able to resolve this issue and ensure that place was returned whenever a piece moved down into another piece or the floor. We also faced a persistent issue with wallkicks where certain pieces would phase through others if rotated at just the right time. This bug was difficult to isolate because of how rare it occured, with manually creating it taking extreme precision. However, by revising how rotations were handled within the move() method, adding checks before the logic executed to determine whether the piece should be placed, we were able to eliminate this issue and yield a robust rotation system. Overall, the majority of issues faced during this assignment were a result of the large amount of code we were required to implement, allowing for lots of opportunities to incorrectly structure statements in ways that created bugs like those described above. To help reduce this issue, we attempted to utilize Extreme Programming, writing tests as we wrote each method to reduce bugs and create higher quality code. I don't believe we were completely successful in utilizing this method of programming, as we leaned further towards writing large sections of code and then testing for TetrisBoard, but where we did use it we noticed a significant decrease in the number of bugs created, leading us to believe we should continue to practice this style of programming.

Assumptions:

The assumptions that we used while implementing TetrisBoard and Piece largely revolved around the fact that TetrisBoard and TetrisPiece will be run through JTetris, and thus the size of the board and spawn position of the pieces should be assumed to be valid. We also assumed that each piece was one of the seven given tetromino types, as passing a nonexistent/strangely shaped tetromino would be inconsistent with the game of Tetris as we are familiar with it and would pointlessly break our code.

Pair Programming Log:

10-1-19:
Ryan Drove 30 min
Tejas Drove 1 hr

10-3-19
Tejas Drove 45 min
Ryan Drove 1 hr

10-4-19
Tejas Drove 1 hr
Ryan Drove 1 hr

10-6-19
Tejas Drove 2 hrs
Ryan Drove 1 hr
Tejas Drove 30 min
Ryan Drove 1 hr

10-7-19
Ryan Drove 1hr
Tejas Drove 30 min

10-9-19
Tejas Drove 1.5 hrs
Ryan Drove 1.5 hrs

10-10-19
Tejas and Ryan Debugged 2 hrs

10-11-19
Tejas drove 1 hrs
Ryan drove 2 hrs

10-12-19
Tejas worked independently 5hrs

10-13-19
Tejas worked independently - 3hrs

10-14-19
Ryan worked independently 5hrs
Tejas Drove 2 hrs
Ryan Drove 2 hrs

Testing Methodology

To comprehensively test our Tetris implementation, it was necessary to create JUnit testing classes for TetrisPiece, TetrisBoard, and our Brain. The testing for TetrisPiece and Brain was fairly straightforward, with simple unit testing sufficiently testing our code to ensure that getting, setting, rotations, and score calculation were occurring correctly. TetrisBoard was by far the most difficult part of this assignment to test due to the near infinite number of possible random combinations of pieces that could exist on the board. As a result, we chose to test TetrisBoard as systematically as possible, checking whether each different piece type work behave correctly when passed through the move function in different scenarios.

White-box Testing:

For TetrisPiece, we utilized simple unit testing in our PieceTester class to determine whether each of its functions were executing correctly. Because TetrisPiece primarily consists of getters and setters, this involved passing certain parameters (Piece type, rotation index) to TetrisPiece and then asserting whether it returned the correct fields given our input. This was the case for TestGetPieceType(), TestGetRotationIndex(), TestGetWidthandHeight, TestGetBody, and TestGetSkirt. These tests were actually very informative because they allowed us to check that we were always passing and receiving the correct information to TetrisPiece. If there had been some unchecked error within these methods, our entire Tetris game would have functioned incorrectly and thus these tests were useful in ensuring that our solution design worked. We also tested the equals method within TetrisPiece, asserting that certain pieces should either be equal or not equal within TestEquals to validate that this function worked correctly. The most complicated test within TetrisPiece was that of TestClockwiseandCounterClockwisePiece(), a method that tested a full circle of clockwise and counterclockwise rotations by applying clockwisePiece() and counterclockwisePiece() to a given piece and checking whether it returned the correct rotation index. This test is arguably the most important within TetrisPiece because of how vital the functionality of rotations are to the game of tetris. As seen from LameBrain, it's not really Tetris if you can't correctly rotate the pieces and thus ensuring that these two methods worked correctly was extremely important to ensuring a robust solution design. Although all of these tests primarily look at the functionality of TetrisPiece at a more abstract level, I believe they still fall under whitebox testing given the knowledge of the code required to write these tests and gain useful information from them.

For TetrisBoard, we implemented a testing harness class BoardTester that utilized multiple different board states to test whether our piece transformation logic within move() was

working correctly. We were able to abstract this testing down into expected Result states given certain scenarios. For example, when putting a new piece at the top of an empty board we expected it to return Board.Result.SUCCESS whereas if the board already housed a piece directly below our current piece we would expect Board.Result.PLACE. Thus, we created a series of white box tests (TestMoveDownUnobstructed, TestMoveDownObstructed, TestMoveLeftUnObstructed, TestMoveLeftObstructedWall, TestMoveLeftObstructedPiece, TestMoveRightUnObstructed, TestMoveRightObstructedWall, TestMoveRightObstructedPiece, TestMoveClockwiseUnObstructed, TestMoveClockwiseWallKick, TestMoveCounterClockwiseUnObstructed, TestMoveCounterClockwiseWallKick, TestMoveDrop, and TestMoveNothing) that exhaustively tested each piece for every possible instruction and game state TetrisBoard could encounter. By abstracting our testing to the simple interactions between pieces whose outcome we would always know, we were able to comprehensively test move() so that it should perform correctly under any situation. Tests that included “Unobstructed” checked the behavior of a piece given an empty board, typically resulting in a Board.Result.SUCCESS. Conversely, Tests with “Obstructed” in the name tested the behavior of a piece when trying to move or rotate into either a piece or a wall. These tests were very important because it was critical to ensure that pieces would return Board.Result.PLACE and remain on the board when they could no longer move down. Additionally, it was very important to systematically check the wall kicks for each piece because this aspect of our code was more complicated and ensuring that each piece could successfully wallkick allowed us to know that our Tetris piece would function correctly even when the xPos or yPos were technically outside of the pieceBoard.

Black-box Testing

The blackbox testing we implemented for Tetris primarily consisted of just playing our implementation and trying different ways to break the game. This included spamming left and right when a piece was about to be placed, spamming clockwise and counterclockwise, and stacking blocks in such a way that removeRows() would have to remove multiple rows in one turn (using a vertical stick piece). Overall blackbox testing was a very inefficient method of testing to use for this project, given how difficult it is to replicate and identify the specific cause of errors by simply playing the game. However, it did have its uses in limited amounts, as the methods we described above actually helped us find out that certain bugs existed, allowing us to then narrow in on the issue with whitebox testing and ultimately fix our code. Thus, the only blackbox testing that we found necessary for this project was through playing our implementation of the game, while whitebox testing through JUnit were more than sufficient in locating the bugs we observed.

Conclusion:

Overall, I believe we implemented an extremely comprehensive and efficient solution for Tetris, in large part because of the extensive whitebox testing we utilized to test TetrisBoard and TetrisPiece. By first observing bugs through blackbox testing, we were able to tailor our

whitebox tests to check for certain scenarios and ultimately correct erroneous code that we had implemented. Through our testing, we were able to handle all discernable edge cases (wallkicks, stuck between two pieces, turning as it is placed, etc.) that would cause TetrisBoard to behave incorrectly, resulting in an implementation of Tetris that very closely models any version you would find online. Additionally, our code executed successfully for 7 PieceTester tests and 30 BoardTester tests, achieving 76% and 92% method coverage and 90% and 89% line coverage for TetrisPiece and TetrisBoard respectively. The 20% of methods within TetrisPiece and 8% within Tetris Board not covered consist of methods that we planned to use for our implementation of Brain but did not end up using.