

Project 4 Report

Atharva Pendse & Zhaosong Zhu | CS 314H

Assignment Goals

In the Tetris project, the objectives are to implement a Tetris game, consisting of *TetrisBoard* and *TetrisPiece*, and design a brain that scores high in the game based on its heuristics. Through our research on the SRS rotation system for tetris pieces, we are to develop a relatively comprehensive understanding of the rotations possible for a piece. Our personal goals include writing out efficient algorithms that occupy fewer resources and performs more efficient, figuring out good heuristics and algorithms that suits well with AI, and finishing bonus Karma activities to gain more knowledge about the entire project.

Solution Design

The design of the Tetris project involves three aspects: *TetrisPiece*, which inherits *Piece* abstract class, *TetrisBoard*, which implements *Board* interface, design a brain that automatically plays the game, and devises a test harness. The *TetrisPiece* class aims to produce a new piece and its rotations based on the x,y value sets passed in. The *TetrisBoard* receives different inputs from the user and keeps track on all the conditions when playing the game, including clear rows. The brain uses heuristics we designed to play the game and needs to score at least 100. To see if all these classes are working, we also design a test harness for each of these classes.

A. High-level details on implemented classes and methods

1. *pointEqual(Point [] p1, Point[] p2)*

The *pointEqual* method intends to help compare if the two point sets passed in contains the same points, regardless of the order of the points.

To perform the intended function, two for each loop and an integer are used. Under *for(Point x:p1)*, another for a loop - *for(Point y:p2)* is constructed so that if for all points in p2, if there is a point correspond to Point x, then the integer *trueCounter++*. Since the size of p1 and p2 is compared at the beginning, so we will see if *trueCounter* matches with either one of p1 and p2's length. If yes, then return true and vice versa.

This method is also called within the *equals()* method of *TetrisPiece* class, where the two *Piece* objects passed in compares each other.

The objective of this method is to ensure even when the order of points is different; the comparison will still be correct. Besides, by using this method, we can more easily

compare between two *Point* array, without needing to pack that as a *Piece* object, which hopefully is more efficient and memory-saving.

2. *int ox, int oy*

These two variables are essential in the TetrisBoard class because many computations related to Piece movement are based on these two variables.

As shown by their names, the *int ox* keeps tracks on the *x* value of the origin on the board and *int oy* for *y* value of the origin. Whenever a new piece is created, the *ox* and *oy* are defined as below:

```
oy = getHeight()-p.getHeight();  
ox = getWidth()/2;
```

For example, if a horizontal stick is created within a board of height 24 and width 20, then its origin for *y* will be 23 and its origin for *x* will be 10. As the game goes on, these value will change based on different operations - for example, *ox--* if move left and *ox++* if moves right.

The objectives of these values are to help us keep track on the location of the piece and to determine whether it is okay to move left, right, drop, down. In doing so, we won't need to call on *dropHeight* method each time to determine all actions above can be performed. Besides, merely by controlling these variables is apparently less computationally expensive, thus allowing the board to perform more smoothly.

3. *checkRightLegal(Piece p), checkLeftLegal(Piece p), checkDownLegal(Piece p)*

These three methods help check whether it is legal to perform a LEFT, RIGHT, or DOWN(also DROP) action. All three methods follow the similar reasoning. At first, it will obtain the skirt for left, right, and bottom.

With this information available, it will first check if the *x* or *y* value for each point within the piece will be out of bound if the move is performed.

For example, in the *checkLeftLegal* method, *if(ox+a[i]-1<0) return false;* statement will try to see the sum of *ox* and its skirt value -1 will be smaller than 0, which is out of bound. If so, then a false will be returned, and this action will not be performed.

Then, the method will check if the next block in the boolean array is filled. *if(board[oy+i][ox+a[i]-1]) return false* statement checks if the block at the value we checked in the first step is filled. If the condition statement is true, which means it is filled, then a false will return. If it passes both of the tests, we will return a true.

B. Abstractions

1. *placePiece(Piece p)*, *erasePiece(Piece p)*

As name suggests, the *placePiece* and *erasePiece* methods helps update as movements are applied to pieces. The *erasePiece* method will change the current blocks that the piece occupies from true to false, as specified by

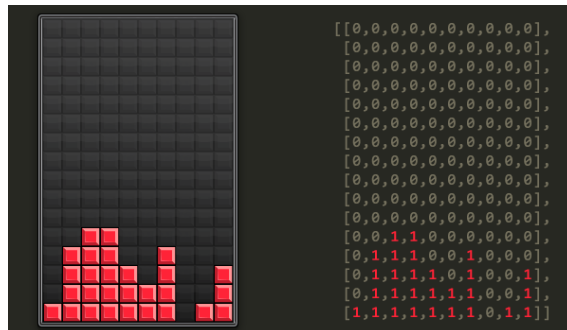
```
board[oy+(int)a[i].getY()][ox+(int)a[i].getX()]=false;
```

By using ox and oy value, which is the absolute location on the board, and x and y value relative to the origin, we can easily manipulate the corresponding points.

The *placePiece* does a very similar job, using the same expression above setting things true. Since these jobs are repetitive, we implemented these two methods so that whenever an action needed, we can call on these methods.

2. Abstracting the Board as an array of boolean values

Although the tetris board is shown as a board contains different pieces placed, or about to be placed, on the simulator, the *TetrisBoard* method uses a 2D boolean array to show whether a block is filled. If the block is filled, then the value equals to true. Otherwise, the value equals to false. This abstraction of the tetris board helps to show the state of the board clearly, without needing to use much memory. For example, if we have a board as shown below, then *board[0][0]=board[1][0]= true*, as well as all other areas marked 1.




https://cdn.tutsplus.com/gamedev/uploads/legacy/035_tetrisCollisionDetection/BasicGrid.png

Besides, the *placePiece(Piece p)*, *erasePiece(Piece p)* methods can be implemented easily because of this representation. With the ox and oy values, we can give indexes on *board[y][x]* to make changes easily.

3. Skirts for bottom, left, right of a piece

In the constructor of *TetrisPiece*, three arrays of integers, *sk*, *lsk*, *rsk*, are created after the array of the point is passed in. The skirt, *sk*, stores the lowest y value of each x, relative to the origin ox and oy. Similarly, the left skirt, *lsk*, and the right skirt, *rsk*, stores the lowest x value related to the origin.



For example, the skirt for  will be 1,0; the left skirt will be 1,0,0; the right skirt will be 1,1,0. With the knowledge of origin with an absolute index of the entire board, we can easily calculate whether the bottom, left, and right part will allow the piece to do any moves.

Project Discussions

A. Board and Piece Designs, Assumptions, Strength, and Weakness

As we write up codes for *TetrisBoard* and *TetrisPiece* class, one important consideration, as mentioned in the instructions, is to perform as many calculations before a getter is invoked, so that the simulator can run in constant time.

- *TetrisPiece*

To speed up the getter methods inside *TetrisPiece* method, we try to perform many of the computations inside its constructor, *TetrisPiece(Point[] array)*. Below is a list of variables initialized and their functions.

Point[] points	An array to store all points that a tetris piece contains
int[] sk	An array that stores the skirts for the bottom of this tetris piece
int[] lsk	An array that stores the skirts for the left side of this tetris piece
int[] rsk	An array that stores the skirts for the right side of this tetris piece
int w	Stores the max x value among all points as width by iterating through all points
int h	Stores the max y value among all points as width by iterating through all points

By initializing these variables properly inside the constructor, we try to ensure that the getter methods can perform in constant time.

However, one possible weakness lies in *getPiece(String pieceString)* method. In this method, a *String* that contains x and y values of points is passed in. Then, to return a *Piece* object, it has to call on the *TetrisPiece* constructor. Then, a chain of the *piece.Next* instances are created using the *rotate(Piece piece)* method to obtain correct point array to construct new objects again. Although these actions may slow down the *getPiece* method, it is necessary to make sure that all *piece.Next* instances can be created, in case it's necessary in the *TetrisBoard* method for rotations.

- *TetrisBoard*

1. *Designs for Board accessor methods*

What kinds of information do you need to store for the Board accessor methods (listed above) to run in constant time?

Of the information listed in response to the previous question, how often does each one change?

Based on the Board interface, there are in total nine accessor methods that return information on current piece, board layout, last action and result due to a certain action. To satisfy the constant time requirement, we try to let each accessor method perform no or very slight calculations. Thus, most initialization and change of variables happens in the constructor and certain methods. Below is a list of how each accessor method variable is updated.

getCurrentPiece()	A <i>Piece cp</i> variable is initialized inside the constructor, and whenever a <i>nextPiece</i> method is invoked, the <i>cp</i> is set to the Piece passed in.
getLastResult()	Inside <i>move()</i> method, before a result is returned, we use a <i>Result result</i> variable to store the last result returned.
getLastAction()	Inside <i>move()</i> method, we use an <i>Action action</i> variable at the first line to store the action passed in as the last action performed.
getWidth()	return <i>board[0].length</i> ; <i>board</i> is the 2D boolean array initialized in the constructor to store the state of the entire board
getHeight()	return <i>board.length</i> ;
getRowsCleared()	Merely return an integer; <i>int rows cleared</i> was initialized to 0 for a newly created <i>TetrisBoard</i> object; in the <i>clearrow()</i> method, whenever a row is detected to be true in the full row, we clear the row and <i>rowscleared++</i> at the same time
getGrid(int x, int y)	Directly access the board through the x,y value passed in; return <i>board[y][x]</i>
getColumnHeight(int x)	An array is initialized in the constructor to store the height of each column x; this method returns <i>columnheights[x]</i> and <i>columnheights[x]</i> is updated through <i>updateColumnHeight()</i> method, which changes this array whenever a piece makes any movement.
getRowWidth(int y)	A for loop is used in this method to iterate through a given row and return the # of <i>true</i> found.

On the implementation introduced above, we find that many of the variables update pretty often. The piece updates whenever a new piece is created; the last action and result is updated with the move method; the number of rows cleared and the height of each column is changed when *clearrow()* and *updateColumnHeight()* methods are called on, which means they are updated when the clear row and update column height actions are

complete. All other methods are pretty much related to the *board* array itself, so they can be easily obtained by calling *.length* or count the number of *true*.

2. *dropHeight()*

The implementation of *dropHeight()* method mainly depends on three variables - the x value to drop from, the y *origin*, *oy*, of current piece, and the skirt.

In our design, we define a variable *dropheight* to see the change of *oy* value if the drop is going to happen. Then, a for loop is used to iterate between *int i = 0* and *i < piece.getWidth()*. For each *i* value iterated through, we will try to find the minimum value of the difference between the sum of *oy* and *the skirt at specific i*, which gives the height of the bottom-most piece at each x value relative to 0. Then, we use this value to minus the column height at that specific column to see the max drop possible. Each time when we find a value, we keep the minimum between *dropheight* and the result of calculation above.

In doing so, we find the minimum value that we can drop for each x correspondent. Thus, the value returned will be *oy - dropheight*, which is the difference between y origin and the max height that we can drop.

3. *Comparisons between Boards objects and replications through testMove()*

The default *equals* method is rewritten for the *TetrisBoard* class to make a meaningful comparison between two boards. In the overridden method, we check each of the following items.

- Width and Height of the two boards

We check the width and height of each of the two boards at the beginning to see if they matches. This is accomplished through the *getWidth()* and *getHeight()* method. If anything doesn't match, we will return a *false*. We think checking these two variables can make the algorithm less expensive because if their height of width is not equal, there's no point in checking other states.

- Is the array matching with one another?

This part is done through the *getGrid(x,y)* method. Having checked that both boards have the same height and width, we use the height and width of either one of them to iterate through each element inside the two boolean array. If there's any square that's not equal, we return a *false* because the piece on the board is not equal.

- Last action and result

Then, we will see if the last action and result can be a match between the two boards. These two factors indicate that an action ranging from *DROP*, *LEFT*, *RIGHT*, *DOWN* has just executed, and the same result was returned for the corresponding action.

- Rows cleared

By checking the rows cleared, we can eliminate the case where the two board has the same configuration inside their array, and same last action and result, but have been played for a different length of time. Although this case is very rare, we try to consider it.

In the *testMove()* method, we try to make a copy of the current board and perform the passed in action on this board to see the results that come with a certain move. Thus, the implementation here is twofold. For the one thing, we need to copy the current board state by attributing some variables from the current board to the new board. For the other thing, we return the board once after it makes a move.

To accomplish the 1st part, we will first call the *TetrisBoard* constructor, which requires the width and length of the board. These two variables can be easily obtained through *getWidth()*, *getHeight()*. Then, we designed another method called *setBoard()*, which takes in the board array, ox, oy, and a number of rows cleared. And also the last action and result. Once these variables are set, *updateColumnHeight()* method and the current piece will be added to the new board. This design is in line with the *equals* method and all the getters of the board so that this will be a board the same. With the array, the *getWidth()*, *getHeight()*, *getGrid(int x, int y)*, and *getColumnHeight(int x)* are matched; besides, the current piece, last result, last action, number of rows cleared are dealt with so that the same result will show in *getCurrentPiece()*, *getLastResult()*, *getLastAction()*, *getRowsCleared()*.

The second part is straightforward: we invoke the *move(Action action)* method on this new board, and one of the action is performed in accompany with a result. Then, this new board can be returned so that the client can judge on the result.

4. *Do some Actions always return the same result*

Based on our implementation, *Action.NOTHING* and *Action.HOLD* should return the same result.

For *Action.NOTHING*, since no action is performed, it is expected that when nothing happens, doing nothing should be the success result. This will be useful especially when a Piece is held for AI. Once a piece is held, if no move is applied to it, we can let it do NOTHING so that it can keep the state all the time.

For *Action.HOLD*, since the HOLD action is pretty much equivalent to a pause, it should always return success as long as the game is ongoing. When we do HOLD once again, this means we want to resume the game, which should as well be a SUCCESS.

5. *Rows clearing and efficiency*

We design a method *clearRow()* to accomplish row clearing. This method is composed of three parts with three for loops.

The first loop starts from 0 and iterates to *getWidth() - 1*, serving as the y index on the Board. In doing so, this outmost loop makes sure that each row is checked. Inside this loop are two other loops. The first one checks from left to right, based on the y provided

if all the block in the same row is filled. If any of the element `board[i][j]==false`, then a break statement is used with the label to start from the first line of the outmost loop.

If it passes the first inner loop, then it enters the second loop, which will change the entire line from *true* to *false*. Right after the loop, we will add one to the number of rows cleared and minus one from the maxheight variable - `rowscleared++`; `maxheight--`;

Having finished all these, the last for loop will scan any line above the current y value, if there's any piece above, this loop moves them down and set the original block from true to false - `board[k-1][l]=true`; `board[k][l]=false`;

To make this method more efficient, we add in an *int* variable. This variable is used in the first inner for loop, such that if there appears to be a row that's not fully filled, this variable will be changed from 0 to 1. Then, the other two for loops, which erases the row and move pieces down from above y, will not be executed. In doing so, we can make it more efficient. Besides, this method is only called when DOWN cannot be done or DROP action is performed.

B. Edge cases and Potential Solutions

1. A null Piece used for actions

When a new *TetrisBoard* is created, the *Piece* variable is initialized to null because the piece is unknown for now. However, this may cause a problem that if `move(Action action)` is called on this board, a `NullPointerException` will happen because of the piece. Thus, at the beginning of the `move()` method, we will check if currently, the piece is null. If so, then result `NO_PIECE` will be returned instead of causing an error.

2. Null object used for `equals()`

Similar to the idea above, we also add the same mechanism to the `equals()` method for the *TetrisBoard* and *TetrisPiece*. It is necessary to do so because both `equals()` method takes in any *Object* and then cast the object to types necessary. If the Object is null, then an error will arise when trying to do type casting.

3. Rotation when piece just created

When a piece is just created, it is likely that its rotation may go out of bound. For example, when a ■■■■ horizontal stick is rotated counterclockwise, it may go out of the boundary of the boolean, assuming the origin of it doesn't change.

To deal with this problem, we referred to the SRS rotations rules on the tetris website: <https://tetris.wiki/SRS>. It provides the shift of origin. According to its rules, we implemented `wall kick clockwise()` and `wallkickcounterclockwise()` method. Therefore, the origin will change accordingly.

C. Brain Designs and Results

- 1) Brain Design Decisions

For our Tetris Brain, we used a strategy similar to the one used by lamebrain. Our brain considers all possible moves for a particular board and adds each of them to the `ArrayList options`, and the

corresponding first moves to the ArrayList *first moves*. However, our brain differs from the lamebrain in 2 key areas:

1. The features of the board that we use to calculate the score are better than the ones used by lameBrain.
2. We consider the rotations of pieces in addition to translation.

Instead of having the score be based on a single characteristic (the maximum height of the board), we came up with a set of features that we thought would be useful in calculating the score. These features are:

1. **The sum of heights of all columns in the board:** Obviously the sum of heights of all the columns of the board should be as low as possible. We therefore gave this feature a negative weight in our score.
2. **The number of 'holes' in the board:** We defined the number of holes as the number of squares that were empty themselves but had a filled block directly above them. Holes are undesirable, because to clear holes we have to clear all the rows above them first. Hence, we gave them a negative weight in our score.
3. **The 'bumpiness' of the top of the board:** Highly bumpy surfaces tend to create more holes, so our goal should be to minimize the bumpiness. Bumpiness is calculated by taking the sum of the absolute differences of the column heights for each pair of adjacent columns in the board.

Once we had the features, we simply gave each of them equal negative weights in the score. The formula is:

$$\text{Score} = 100000 - 51 * (\text{sum of heights} + 10 * \text{newBoard.getRowsCleared}()) - 18 * \text{bumps} - 35 * \text{holes}(\text{newBoard}) + 76 * \text{newBoard.getRowsCleared}();$$

Instead of merely relying on the max height of each board, we instead use the three factors above to calculate the score. And we give each one of them a specific weight so that (hopefully) the best board will have the least total height and holes, and the smallest division between heights of the column. We had originally intended to implement a genetic algorithm that calculated the ideal weights for the features, but couldn't implement it due to time constraints. Therefore, we decided the weights in referral to the following online article:

<https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/>

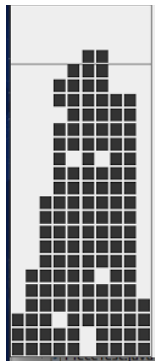
2) Results

Full speed	728
Full speed	1094
Full speed	795
Full speed	1877
Full speed	899
Full speed	2300
Full speed	1150

Full speed	2424
Full speed	1109
Full speed	2775

The highest score recorded by our Brain is 3894, which only happened once and not in the above ten trials.

With the assumption that the AI will perform as well or even better under lower speed, we run all the ten trials under full speed. Based on the results above, we find that there are always some pretty big variations in the scores. Whenever the board scores a relatively low score, it always stacks up pretty high in the center while leaving blank spaces on the two sides. Also, the last piece will land in the middle even when such situation happens, as shown in the figure below.



We believe the result is largely due to that since only the couple columns in the middle are high, the sum of all column heights are probably not that good a factor for scoring.

D. Karma & Personal Reflections

1. What we learned from this project and future improvements

For the one thing, this project greatly trained our logical thinking process. By implementing the SRS rotations and the AI brain, we have to consider a lot of corner cases and efficiency of the algorithms. For example, when designing our AI brain, we started off by listing all results after rotations and drop and rate those possible moves based on the original scoring method. Then, we started to consider how even the column heights are and how many holes present. These thinking process helps improve the AI greatly.

Besides, this is the second project that we work with someone else. This is a very refreshing experience for us. Throughout the two weeks, we have been getting more adaptive to each other and developed better understandings. More details are under pair programming experience.

However, one thing that we definitely can improve is to devise efficient code design. We started off by writing a lot of codes, which are hard to change after completion because it is hard to debug a large truck of codes. Next time, we will try to incorporate more ideas from XP, such as developing test cases as we write codes, finishing codes little by little.

2. Pair programming experience

A. Date and working time logs

Date	# of minutes	Items Accomplished
10/1/2017	Together: Zhaosong - 30 Atharva - 30	Read through assignment descriptions together; Implement the <i>TetrisPiece</i> class
10/4/2017	Together: Zhaosong - 40 Atharva - 45	Implement the <i>TetrisBoard</i> class; Spend most of the time working on SRS rotations
10/7/2017	Together: Zhaosong - 45 Atharva - 75	Finish up SRS rotations; Play the game through the simulator and fix clear rows
10/8/2017	Together: Zhaosong - 25 Atharva - 40 Individual: Zhaosong - 20 Atharva - 30	Continue to work on <i>TetrisBoard</i> class; Discuss JUnit testing
10/11/2017	Together: Zhaosong - 60 Atharva - 70	JUnit Testing class write out; Brain design; Write parts of report together through Google Docs
10/12/2017	Together: Zhaosong - 180 Atharva - 180 Individual: Zhaosong - 70 Atharva - 40	Write up parts of the report; Brain design; JUnit testing running and debugging;

B. Experiences

Having done pair programming for two projects, we are getting more comfortable to cooperate with one another.

First, we find that it is easier for both of us to come up solutions or insights. For example, when we are trying to link all four *pieces* together, we didn't realize to use a constructor and construct the object first, to avoid null pointer. Through discussion, we found out that merely doing `piece.next.next.next` was not feasible. Then, we found out we should use a constructor for each `piece.next` object. Besides, we also write up the rotate method that, instead of sending back

objects, sends back `Point[]`, which saves some memory and make it more efficient to create the new *TetrisPiece*, which wants a `Point[]` as a parameter.

Besides, pair programming gives each of us more incentives to work on the project. If working alone, we are likely to postpone the programming homework because of personal schedule. However, when working together, since we will periodically send a message to one another to ask if the one wants to work collaboratively. With the experience of project 3, we have had a good idea about each one's schedule, based on which we try to work fluidly both in a pair and individually.

Regarding difficulty, since we are not too familiar with command lines for GIT, we are still keeping up versioning through Google Drive, which sometimes causes unsynced changes on our PC. However, we will try to go over some tutorials from GitHub, so that we can set up the repository and enable versioning. Besides, as we work longer, we begin to understand each other much better, especially when transmitting ideas.

Testing Methodology

The JUnit testing for this project includes two classes: *PieceTest*, *BoardTest*. Detailed explanations are shown below:

PieceTest: This test intends to test the methods and internal calculations within the *TetrisPiece* class.

- *getHeight()*, *getWidth()*

To start off, we test if the *getHeight* and *getWidth* method for a certain piece will return the correct height and width. This method is intended to ensure that no calculation errors occur when we translate a String into a Piece object.

- *equals()*

Then, we created two *TetrisPiece* objects for comparison, one of which contains the same point sets and the others different. The expected result is that the pieces with the same set of the point should equal and vice versa. If it is true, the *equals()* method returns true. Else, it returns false.

- *getBody()*, *pointEquals()*

In the next section, we test these two methods together. Since *getBody()* returns the Point array for a given piece and *pointEquals()* compares between two point arrays, we use *parsePoint* to create a Point array and compares the array with the result from the *getBody* method.

- *getSkirt*

In this section, we manually type out the expected skirt for a given piece, which in this case is a square, and use a for loop to test each item. In doing so, we can not only check if the values are equal but also if the skirt array has the same size as we expected.

- *nextRotations()*, *nextClockwiseRotation()*

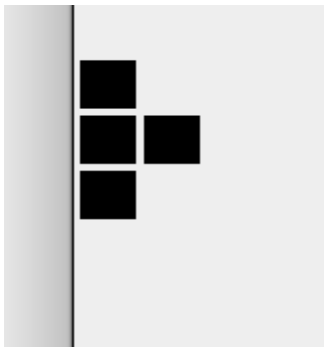
In our original implementation, we attribute the clockwise rotation to each *next* variable. Thus, a minor change is made such that we created *nextClockwiseRotation()*, which returns next, and *nextClockwiseRotation()*, which returns next.next.next. So, in the testing section, we try to utilize these two methods and see if the result will match after several calls on these methods.

First, we tested if using `.next` on a vertical stick will return a horizontal stick. Then, we randomly chose a piece from the array of 7 pieces. We called on `nextRotations()`, `nextClockwiseRotation()` to see whether the results matches. Finally, we implemented these two methods below to see if, after multiple calls, the pieces are still chained well and returns the correct result.

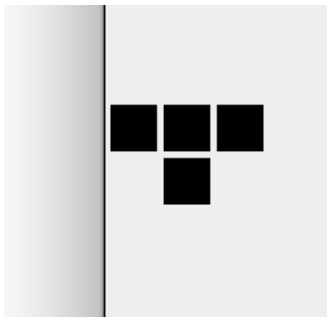
```
assertArrayEquals(testPiece.nextRotation().nextRotation().getBody(),
testPiece.nextClockwiseRotation().nextClockwiseRotation().getBody());
```

```
assertArrayEquals(testPiece.getBody(), testPiece.nextRotation().nextClockwiseRotation().getBody());
```

BoardTest: This class tests most of the methods in the *TetrisBoard* class. We tried to come up with cases that could potentially cause errors (For example cases in which after rotation, part of the piece went outside the boundaries of the board.) First, we created an instance of Board, which we called to *board*. We then initialized a T piece in the *board*, rotated it so that it was represented by “0 0 0 1 0 2 1 1” and moved it to the left border.



Then we tried to rotate it clockwise. After the rotation, part of the piece would have been outside if not for the wall kicks. Due to our correct implementation of wall kicks (at least for this case), the piece after rotation looked like:



Since our rotation implementation extensively uses the origin coordinates *ox* and *oy*, we checked that they were equal to the expected values after the rotation; that is we checked the correctness of the rotation function in *TetrisBoard*.

After rotating the piece, we moved it one space to the right (to check that moving right worked properly) and then dropped it to the bottom of the board. Since the board was previously empty, it fell straight to the bottom. Then, to check if the *updateColumnHeights* and *getColumnheight* methods were working correctly, we updated the column heights and checked if the height was equal to the expected height (For example, if *columheights[2] = 2*).

We then repeated the above process for the second L piece.

Next, we tested the *testMove* method. A new Board *board2* was created that stored the hypothetical state of *board* if Action *act* was executed on it. Then the same Action *act* was then actually executed on *board*, and the 2 Boards were compared to check if they were equal, which they were.

Implementation:

We used JUnit 4 for running all our unit tests. For the most part, we used the *assertEquals* and *assertArrayEquals* methods to check if the actual values matched with the expected values. We tested all the *TetrisBoard* variables in *BoardTest* using their access functions. Following is a list of the methods we tested and their line numbers:

Method in TetrisBoard	Code line in BoardTest
<i>equals()</i>	27, 28, 29
<i>move()</i>	34,72
<i>Ox</i> and <i>oy</i>	50, 51
<i>getColumnHeight()</i>	57, 58
<i>getMaxHeight()</i>	62
<i>nextPiece()</i>	144
<i>Wallkick()</i>	99, 105
<i>testmove</i>	139
<i>getRowsCleared()</i>	148
<i>getLastAction()</i>	152
<i>getLastResult()</i>	154
<i>getGrid()</i>	157

Negative tests:

We used a couple of asserts(false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn't just passing all the cases.

Black Box Testing:

While we were coding the assignment, we played Tetris over and over to make sure that the particular feature we had just completed worked. Black-box testing was also very helpful in reminding us what was still left to complete. For example, before we implemented the wall kicks, our program would give an `arrayIndexOutOfBoundsException` exception whenever we tried to rotate a piece that was aligned to the edge of the board. After we implemented wall-kicks this error was resolved.

In addition to actually playing the game, we used `System.Out.Print` statements in several places for debugging the code.

We also used black-box testing to check several wall kicks that we didn't code into our `BoardTest` class.

White box testing:

As explained above, we created two JUnit test classes `BoardTest` and `PieceTest` specifically for testing our code.

Testing Exceptions:

The exception may happen when parsing the string to an array. It is handled in the `Piece` class.

Strengths and Weaknesses in our Testing Methodology

Strengths:

Testing the code increased our confidence in its correctness. Since we used JUnit, we were able to automate the testing process. Instead of having the program print diagnostic statements and examining those statements to make inferences about the correctness of our program, we were able to have the JUnit test method check for correctness automatically. Having an automatic checking mechanism shortened the time required for testing individual cases considerably.

Weaknesses:

Tetris has so many possible moves and edge cases that it can become very difficult to think of and test all of them. We came up with a large amount of test cases and tried to test as many cases as we could think of, but couldn't code all of them due to time constraints. However, we were able to check most of them by actually playing Tetris countless times.