1 Introduction

Tetris is a widely played game, but the mechanics behind the screen are hidden from the user. In this project, we once again get to test our skills in paired programming by creating our own Tetris game. Even more exciting is that we also create an AI for our Tetris game. Less of an excitement is the overwhelming amount of tests we need to conduct. Overall, this project is by far the most complex we have created.

In creating this program, we hoped to further refine our understanding of object-oriented code. Furthermore, we wanted to make use of different testing methodologies to ensure the correctness of our game (and thus its enjoyability) through a combination of automated unit tests and integration tests. As always, we aimed to practice good programming practices, such as consistent formatting style and informative commenting, and attempt to design useful and powerful abstractions.

2 Solution Design

2.1 Piece

2.1.1 Representation

The body of a TetrisPiece is represented with an array of Points, where each point stores the coordinates of a block on the board occupied by the Piece. The array is first sorted by the x component of the point, and then the y. The coordinates are represented as stated in the assignment guidelines, with (0, 0) being the bottom-left corner of the tightest possible bounding box on the Piece. Note that TetrisPiece does not store the location of the piece within the context of the game; that is left for TetrisBoard to store.

2.1.2 Get Methods

When a TetrisPiece is instantiated, all needed information of that piece is immediately computed and stored in its respective field variable. This way, methods such as getWidth(), getHeight(), getBody(), and getSkirt() can run in constant time, which is beneficial during gameplay.
2.1.3 Rotation

Here is the most complicated part of TetrisPiece. Originally, we coded rotations by looping through a circular linked list of TetrisPiece objects, each with a different body depending on the rotation. But as Section 3 with explain in detail, we realized that without adding methods to the Piece interface, we could not know how to shift the position of the piece within TetrisBoard after rotations. As a result, our TetrisPiece implements rotation by storing all of its rotate forms within an array, and calling the nextRotation() method simply increments its rotation number. More importantly, nextRotation() no longer returns the piece in its rotated form, but instead returns a shifter piece, which holds vital information regarding the TetrisPiece.

Specifically, it holds
• how much to shift the piece’s board position, given a clockwise or counterclockwise rotation (shift value), which is critical for automating TetrisBoard’s rotation function without hard coding values, and
• the piece’s rotation number from 0 - 4, which is important for removing most of the hard coding for wall kicks. Again, more will be explained in Section 3.

Calculating the 4 rotation forms, and their corresponding shift values, was one of the more challenging aspects of this project, largely because the official Tetris SRS (Super Rotation System) guidelines for doing so are not entirely consistent mathematically. As far as we can tell, SRS for every piece except the I piece, SRS first chooses a center of mass. Let the center of mass be (x, y). If either x or y is rational (but not an integer), and not exactly halfway between two integers, then $\frac{x}{y}$ is rounded. In other words, $\frac{x}{y}$ can be “fractional”, as long as it contains a 0.5 in it. SRS then rotates the piece around the center of mass. In our implementation, we do exactly this, and calculate the clockwise shift values by taking the difference between the coordinates (0, 0) of the original piece, and the coordinates of the bottom left of the tightest bounding box for the rotated piece. And this works for all the pieces, except for the I piece.

For almost every single piece, the resulting shift values will be integers. However, the shift values of rotating the I piece this way are all fractions. Moreover, mathematical convention would round the shift values, but the I piece doesn’t behave like that either. In fact, without representing the I piece using bounding boxes, which our TA’s have disallowed, the I piece shift values are rounded non-systematically. In other words, they must be hard coded, and the int[] adjust array holds these hard-coded rounding values that only apply to the I piece. All other pieces are computed with the above methodology.

2.2 Board

2.2.1 getMethods

Like the TetrisPiece get___ methods, all get methods in TetrisBoard that query the board state are computed whenever a piece is placed, and retrieved when necessary in constant time. The getDropHeight() method is computed in constant time whenever it is called.

2.2.2 Left/Right/Down

These translation methods are all abstracted into a translate() method, which takes in a translation point shifter value, and applies it to all points of the object. Of course, this only
occurs if the translation is valid. Whenever the DOWN action is called, a **place** method is also called, which checks if the current piece is ready to be placed. If so, the board is updated to include the piece, and the current piece is set to null.

### 2.2.3 Drop

This keeps translating the piece downward (with a point shifter value of \((0, -1)\)) until it is placed.

### 2.2.4 Rotations

There are two possible rotations: **clockwise** and **counterClockwise**. The mechanism that rotates an arbitrary piece (of the 7 possible) has been discussed in great detail in Section 2.1.3; however, here we discuss the act of rotating a piece on the board.

The piece is first rotated accordingly (i.e: either clockwise or counterclockwise) using the **nextRotation** method; however, in the case that the piece cannot be rotated, we do not want the issue visual hopefulness to the player by rotating and then instantly rotating back again — so, we first hide the piece from the board using **hidePiece**. Then, we obtain the rotation shifter translation point and its starting rotation index as specified by the official Tetris SRS (Super Rotation System) guide. In many cases, rotation cannot take place in the piece’s current location due to obstructions by other blocks on the board and/or the board’s wall; then, wall kicks (see Section 3.3 for a detailed explanation) are used. Since the first translation point tested by the wall kick mechanism is \((0, 0)\) for each piece, this mechanism will first attempt to rotate in place.

If a rotation is feasible, the **position** field corresponding to the **currentPiece** variable in **TetrisBoard** will be translated by the (rotation shifter + wall kick testing translation point), the piece shown, and the **TetrisBoard.rotationNumber** incremented (and taken modulo 4), and the method returns true. If the rotation is impossible (i.e: a standard rotation cannot work, and neither can each of the wall kick test translations), then the piece is rotated back (thrice for clockwise, and once for counterclockwise), and the method returns false.

### 2.2.5 testMove

The testMove method first creates a new TetrisBoard, and copies all data from the current board to the new board. Note that any data structures are passed by value, so modifications in the new board will not affect the original board. Then, the move function is applied to the new board, and the new board is returned.

### 2.2.6 nextPiece

The method sets the current piece variable to the given piece, and places it as high as possible, in the middle of the field.

### 2.2.7 showPiece and hidePiece

These methods adds/removes the piece from the board by clearing and setting their respective point locations.
2.2.8 Test Methods

Various methods not in the Board interface are written for testing purposes, and they allow access to otherwise restricted data within TetrisBoard.

2.2.9 TestMode

TestMode is used to black-box test the testMove method, by relegating all move actions to testMove instead, and copying testMove’s generated board back into the original board after every move. We use the internal boolean flag TEST_MODE to switch this functionality on and off.

2.3 Brain

2.3.1 getNextMove

First generates all possible end states that involve rotations, translations, and drop, in that order. If a particular list of actions result in a failure, then the action is reduced to a single drop. After generating the possible action results, it evaluates all the resulting boards to choose the best one, based on a score function. Then it takes the action list that leads to the highest scoring board, and if there is a tie, picks the action list that has the least number of actions to carry out.

2.3.2 Scoreboard

The bread and butter of the brain. In our brain, we check for 5 parameters: rows cleared, covered empty spaces, horizontal blobs of white space (also known as isolated landings), maxHeight, and overHeight. Covered empty spaces are spaces with block on top of it, horizontal blobs are horizontally consecutive blocks of empty spaces, and overHeight is how much the maxHeight is over a pre-designated threshold, and is equal to 0 if it is less than the threshold.

For each of these parameters, we simulate an if condition structure by assigning each parameter a weight coefficient of different orders of magnitude. maxHeight has a coefficient of -1, horizontal blobs has a -100, rows cleared has a 10000, covered empty spaces has a -1000000, and overHeight has a \(-10^{10}\) coefficient. And, if overHeight is greater than 0, we swap the coefficients of rows cleared and empty spaces, to put more emphasis on clearing rows if we are in danger of failing the game.

3 Reflections

3.1 Design Decisions

3.1.1 TetrisPiece Rotation

There are two major design decisions we made when implementing TetrisPiece rotation. First is to have nextRotation() rotate clockwise. Second is to change the current Piece whenever nextRotation() is called, and have nextRotation() return a “shifter piece” which holds information regarding how to properly rotate the TetrisPiece manually.

The clockwise decision was made because the official SRS guidelines follow clockwise rotations, and therefore implementing clockwise rotation was both more natural and easier.
to use during wall kicks. However, it was our mistake to not check the java docs of Piece, and so we realized Piece expects rotations to default to counterclockwise. The decision to make TetrisPiece mutable under rotations was, as we have experienced, but incredibly powerful and dangerously difficult. We actually originally implemented TetrisPiece rotation via linked lists. But when we realized the TetrisBoard rotation methods, especially the wall kick mechanics, required more information about the piece than the Piece interface provide, we changed nextRotation() to give us the needed information instead. Specifically, nextRotation() returns the .next Piece, which we wrote to hold the information needed to do rotations and wall kicks. The consequence of this was that our TetrisPiece now needed to change upon rotations. So we changed TetrisPiece 4 to hold all its rotation forms in one object, and have nextRotation() simply increment the rotation number of the object, while returning a shifter piece that contained both the rotation shift values, and the rotation number. Below are the advantages/disadvantages of doing this.

Advantages:
• Simplicity in representation: An array holding all the forms is more intuitive and easier to test than a circular linked list holding all the forms.
• Access to shift values: Every time a piece is rotated, its position also needs to shift to follow SRS guidelines. It is only natural for piece being rotated to hold the shift values needed for rotation.
• Access to rotation number. Similar to shift values, the rotation number of a piece is critical for smooth implementation of the wall kick mechanic. Combined with the fact that we number of rotations clockwise, matching the SRS guidelines, we are able to remove the majority of the “hard code” needed to implement wall kicks.

Disadvantages
• Testing rotations. Rotation testing is not as simple as comparing a bunch of calls to TetrisPiece.nextRotation(). Instead, a copy of TetrisPiece needs to be made. Furthermore, testing requires the Piece to rotate counterclockwise instead of clockwise. As a result, we created a “testing version” of TetrisPiece, where if you feed the constructor a True Boolean parameter before the Point array, the nextRotation() acts almost exactly like the Piece interface expects. Almost, because the rotated piece the interface returns is not liked via reference to the original piece, so all piece comparisons must be done through the TetrisPiece.equals() method, and not the == operator.
• Mutable. Every time we need to copy a TetrisPiece, we need to create a whole new TetrisPiece. Everytime we need to test a rotation, we need to rotate the piece back into place if it failed. This resulted in a lot of time consuming bugs.

Overall, we found that perhaps it was a mistake to make this change. The ideal solution, we think, would have been to keep our original linked list implementation, but change the Piece interface to allow more data access. Part of the fault lies in the interface, but in our attempt to preserve the interface, we changed the fundamental logic behind some of the methods, to our despair.

3.2 Translation Abstraction
We noticed Left/Right/Down all did the same thing to the piece, translate it, so we abstracted this to a translate() method that can translate the piece in any direction by any length.
3.3 Wall Kicks

When a piece cannot rotate in its current rotation, due to obstructions from the walls (i.e: the bounds of the board) or other pieces on the board, the logical solution is to simply not rotate the piece. However, this is unintuitive for the player and leads to confusion, especially in fast-paced games. Instead, the official Tetris game incorporates a set of rules known as wall kicks to determine how to best handle these situations; these rules are described in detail in the Tetris SRS (Super Rotation System) guide.

In the SRS guide, 5 alternative rotation locations are given for each rotation from one rotated state to another rotated state. Since these rotations can go either clockwise or counterclockwise, there are 8 of them; however, each piece does not have its own set of rules — rather, the \{J, L, S, T, Z\} pieces have their own together, the I piece has its own, and the O piece does not move. The rules for each rotation is an ordered set of translation shift points that we test for validity (i.e: if the rotated piece is translated according to these points, will there be obstructions, or can it actually move there?)

The wall kicks take place in the TetrisBoard rotate methods (clockwise and counterClockwise). We first obtain the current starting SRS rotation index for the rotation taking place. Then, using data from the SRS guide, we obtain an array of the translation shifter points to test. The position is shifted by the first shifter point to work for each point in the currentPiece; if it exhausts the entire array without finding a suitable shifter, the rotation is deemed impossible, and no action is taken (along with the method returning false, which prompts a Result.OUT_BOUNDS response). Finally, the current TetrisBoard.rotationNumber is incremented (and taken modulo 4), and the new piece is shown, the method returning true is the rotation took place.

3.4 Copying Objects

We took great care to make sure when Pieces or Board were copied, all data was copied by value and not by reference, to prevent cross object modifications.

3.5 AI

The parameters and coefficients were an attempt to generalize David’s play style. In particular, the AI seeks to minimize covered squares and to place pieces against the edges of other pieces. Furthermore, if the max height reaches a certain threshold, the AI puts greater emphasis on lowering the height of the board.

3.5.1 AI Coefficients

Following David’s play style, we chose coefficients in different orders of magnitude in order to simulate an if/else condition. Specifically, we value rows cleared above all else, and then minimizing empty spaces covered, and then minimizing horizontal blobs, and finally using maxHeight as a tie breaker is all else fails. We chose this because row clears lower all three of the other values, so it is worth the most. Covered spaces stop a row from being cleared, so we want to avoid these at all costs. Horizontal blobs are bad because small blobs require more specific pieces to cover. And finally, if everything else it the same, we obviously want a smaller maxHeight.
3.5.2 AI Results

In a series of 15 runs, the media score was 273. The minimum score was 149, and the maximum score was over 1000.

3.5.3 AI Ideas

One idea we had, though we did not have time to implement, was to create a neural network that would take in each grid in the board as input nodes, and have every possible action be the output nodes, and run the neural network to calculate the best network to play the game.

3.6 Assumptions/Limitations

3.6.1 Immediate Placement

We assume a block should immediately be placed whenever a drop/down method is called and it is on top of another block. This is important because some implementations of Tetris allow the user to “slide” their blocks on the surface until a certain duration expires. The implication of this is that 1) some wall kicks will never be called upon because the block will be placed already, and 2) our AI score is significantly lower than a similar AI designed and tested on sliding Tetris.

3.6.2 Drop

Our drop method is implemented by calling DOWN until the piece lands.

3.6.3 Default Action and Result

Default lastAction is null, default lastResult is Result.NO PIECE.

3.6.4 Action.NOTHING

Calling Action.NOTHING should always be successful.

3.6.5 Pieces

We assume we will only receive standard Tetris pieces as inputs, and that the pieces will be in the Assignment Description format (i.e. not in a bounding box, in SRS L rotation).

3.6.6 Reasonable Board

We also assume the board size of the game is reasonable (ie less than 1 million by 1 million), because larger board sizes will cause the game to run slowly.

3.7 Bugs

3.7.1 Indecisive AI

Our AI originally did not break ties in decision making by least number of actions required to reach the end result. Therefore, the AI would often alternate between two opposite
actions, since both of then lead to the same score. This was fixed when we made the tie breaker, and by adding a hard limit of 20 moves before the AI must call down.

3.7.2 Wall Kick Typo
An unfortunate bug that took an hour to find just 5 numbers miss-typed. Shows the power of automated testing.

3.8 Karma: Holding Pieces
Sometimes the player would rather have a different piece than the one they currently have. A natural solution is allowing them to pick another piece, while holding their current piece for later. One caveat that maintains the game’s balance is that once a player has swapped their current piece for a new piece, they cannot swap back in the same turn. In other words, they must place the new piece before being able to get their original piece back. Additionally, the player cannot see their potential swap piece beforehand.

We implemented this feature by creating two field variable in TetrisBoard: the currently “held” piece, and a boolean variable indicating whether or not the piece had already been swapped in this “turn.” In our own JTetris runner, we created a new keyboard action to activate the HOLD action whenever the z key is pressed.

3.9 Personal Reflections
Overall the program was 30% fun and 70% tedious. While creating (and playing, while debugging) the game was enjoyable, and making the AI was incredibly interesting, the sheer effort required in testing made the overall process painful. However, the project did teach us better coding and (especially) better debugging habits, since both were aplenty in this rather larger project.

3.10 Pair Programming
We found it was optimal to have a mixture of individual work and paired work.
- 10/6/17: 3 hours Paired. Nimit drive, David observe. Go over basic interface, design solution, implement basic methods within TetrisPiece.
- 10/8/17: 7 hours David: Implemented TetrisPiece rotation. 3 hours Nimit: Implemented basic TetrisBoard methods.
- 10/9/17: 6 hours Paired: Reviewed code, implemented TetrisBoard methods, not including wall kicks and testMove.
- 10/11/17: 6 hours David: Tested TetrisPiece, implemented TetrisBoard.testMove.
- 10/12/17: 5 hours David: implemented brain, wrote report sections. 8 hours Nimit: implemented wall kicks.
- 10/13/17: 12 hours Paired: wrote TetrisBoard unit tests, wrote report.
- 10/14/17: 6 hours Nimit: wrote integration tests, finished writing report.

4 Testing
Testing in this project was quite different than previous projects. In this project, we found there were actual necessities for both black-box and white-box testing methodologies.
4.1 TetrisPiece

Since the input size for pieces was quite small (7 possible pieces), testing of operations on pieces was done with manual black-box testing. We manually tested all 7 pieces, and manually check all 28 rotation forms and 56 shift values to make the constructor and the `nextRotation()` method worked. Since these were all the possible values they could hold, the class, and its functionality, is proven to be correct.

Get methods were tested manually, because if we automated the process, our “solution” would be created using the same algorithm that TetrisPiece uses. A simple, manual, sanity checked sufficed.

4.2 TetrisBoard

All translations and standard rotations could be perfectly tested via manual black-box testing using JTetris. Since each piece is displayed in isolation, JTetris essentially tested the rotation/translation methods independent of the rest of the class. With only 7 different pieces, a manual test sufficed.

4.2.1 Wall Kicks

Conversely, wall kicks, a special case of rotations, necessitated automated unit testing. Using the SRS guideline from TetrisWiki, we iterated through each test point of a certain piece’s pre-wall-kick-rotation point checks in order, checked that manually rotating clockwise (and counterclockwise) and re-positioning a piece on a board was equivalent to move-ing the current piece on a board CLOCKWISE (and COUNTERCLOCKWISE). After each check, we set the point just checked to being filled on the grid, thereby forcing the wall-kick mechanism to check, and subsequently place, the piece at the next test check position. All of our unit tests passed, thereby confirming the validity of our wall kick mechanism.

4.2.2 Row Clears

To test row clears, we generated a starting board with no complete rows. We then randomly interspersed 5 complete rows with rows from the starting board to a new board. We prompted the board to clear the rows, and compared the result to the original board.

4.2.3 Integration Testing

To confirm the validity of everything working together properly, we went through a series of steps (covering every Action and Result) by hand, storing the expected board. Then we wrote code to execute the same steps, and compared the results.

4.2.4 Idea: Row Clears

Additionally, it would be useful to confirm that special and interesting/complicated cases of row clears work as expected; however, we did not have enough time to write these tests.

4.3 Edge Cases

- Multi-row clear: The board should clear all rows simultaneously, and drop the pieces accordingly.
• Action upon no piece: The board should return `Action.NO_PIECE`.
• We considered every single wall kick to be an edge case as defined by its entry in the SRS guide.

4.4 Software Engineering

We originally believed the project was easy enough to create using the waterfall method. As a result, we did the following:

1. Design Solutions
2. Implement `TetrisPiece`
3. Test `TetrisPiece`
4. Implement `TetrisBoard`
5. Test `TetrisBoard`

The problems began in steps 4 and 5. When we created `TetrisPiece`, we chose the linked list implementation. But when we started creating `TetrisBoard`, we realized we needed for data from `TetrisPiece`, so changed `TetrisPiece` to what is now, a mutable object. But then finally, when we started testing, we realized there were all sorts of complications involving a mutable `TetrisPiece`. As a result, we wasted a lot of time using a suboptimal implementation of `TetrisPiece`, because we used the waterfall method. In the future, we hope to use Test-Driven practices to ensure this would never happen again.