David Zhang
15-122 Term Project Design

Design

Throughout Middle School, I was an avid fan of Tetris. I played it for hours on end and when 15-112 brought Tetris back into our minds, I knew that I wanted to make my Term Project related to Tetris. Given that we already programmed a basic version of Tetris in class, I knew that in order to make a good Term Project out of Tetris, I would need to step up the complexity considerably. Thus, I decided that I would try to make a much more improved version of Tetris (from our class version), give players the option to play against other players or a computer (not just solo), and create an Artificial Intelligence that could play Tetris better than I could. I shall thus split this design document into these three parts.

Improving Tetris

While the Tetris that we worked on during Week 6 did resemble Tetris enough in order to be immediately recognizable, many essential features associated with Tetris were missing, namely a "queue", a "ghost piece", a "hard drop" feature, and a "hold" feature. To make my queue displaying the next 5 upcoming pieces, I first created a list of length 5 and whenever a new piece was called for, the queue would remove the first piece and make that the current piece and add a new piece to the end of the list (maintaining the list of 5). To make a "ghost piece", I made a copy of the tetris board named a "ghost board" (so that I don't interfere with the game board) and I moved the copy of the falling piece as far down as it could until one of its blocks touched another block on the bottom. Then I drew the ghost piece there, similar to how I drew the falling piece, but with a ghost color instead. For the hard drop feature, I looped through moving the piece down until it was no longer legal to do so, and then I placed the piece. For the hold feature, I simply made a “hold” variable that would keep track of what was in the hold and would switch the current piece with the hold piece (or simply add the current piece to the hold and spawn a new piece, if the hold was empty).

Multiplayer Tetris

Since I planned on making my Tetris multiplayer from the start, I made tetrisGame a class object. Thus, to make my Tetris multiplayer, I was able to easily create two distinct tetris Games that would run simultaneously. Then, I had to pass information between the two games. To do this, I added several functions to keep track of relevant information: addGarbage, hasGarbage, removeGarbage, sendGarbage, gameLost, and gameWon. As one can infer, sendGarbage was a function that returned how many lines of “garbage” should be sent to the opposing player. This was calculated by simply returning the number of lines cleared (after accounting for the player’s own amount of garbage). addGarbage was a function that modified the receiving player’s board by moving the entire board up an appropriate number of rows based on how much “garbage” was sent to them. I did this by making a new board which copied the old board except every piece was one row higher and the bottom row was full of garbage. This process looped until all the garbage was sent. removeGarbage and hasGarbage worked in conjunction. The purpose of the removeGarbage function was that if there was garbage on a player’s board, the line’s cleared would first clear the player’s garbage and would send the remaining lines to the other player. Thus, hasGarbage worked by checking if the bottom column was a garbage row or not and removeGarbage worked the same as addGarbage, except it moved every block down one row and added an empty row at the top. Finally, gameLost simply returned True if the player had lost the game, prompting a game over screen and also calling gameWon for the other player, which would prompt a win screen.

Tetris AI

Constructing the AI was by far the most difficult and time consuming part of my project. First, I made several functions that kept track of relevant board variables that would be necessary in determining what move to make: getColHeights, countHoles, and getGaps. To construct getColHeights, I searched through every column of the board and recorded the lowest row that was not filled by a block -- the column heights were returned as a list. To construct countHoles, I used getColHeights to start from the top of the column. I then counted the number of empty squares underneath this top of the column. This counted the number of holes, which was an integer. Finally, to construct getGaps, I used getColHeights and compared the heights of consecutive columns. Since one-column gaps are the relevant gaps that we would like to keep track of, I set each of the column's gaps to the minimum sized gap on either side of the column. If the gap was negative, I simply set it to 0. After making the functions that kept track of relevant board variables, I created five functions -- possibleRotations, calculateBoardScore, findBestPlacement, hardDropCandidate, and doAIMove -- that would test every possible move, calculate the score of each of the moves, and perform the move of the best score. possibleRotations simply returned a 3D list of the possible rotations of a piece, given the name of the piece as an input. possibleRotations was used in conjunction with findBestPlacement. findBestPlacement looped twice, once for the current piece and once for the potential hold piece, and goes through every rotation of the piece and every possible column the piece could be placed and returns the best piece to use, the ideal number of rotations, and the ideal column, and then calls doAIMove to place the piece in the desired location. doAIMove would first hold the current piece if necessary, then do the proper number of rotations, then move left or right based on what the current column of the piece was and the ideal column. Then, when the piece was in the proper position, doAIMove would hard drop the piece. To determine what the best move was the most algorithmically complex part of my project. With every candidate piece and rotation and column in findBestPlacement, the function hardDropCandidate was called. hardDropCandidate made a simulation of the board and then hard dropped the piece. Then, hardDropCandidate called calculateBoardScore, which would evaluate the board after the piece was hard dropped and give a score to the board. The highest score after the potential drop was then considered as the best move to make and return. To calculate the highest scoring move was the most essential part of my AI. In order to do this, I used a function calculateBoardScore which would calculate the board score based on the value of the relevant board variables (mentioned at the top of this section). However, because I did not know how important each variable should be in determining the move I had to use a genetic algorithm to calculate the optimal weights (or coefficients) applied to each variable. In order to execute the genetic algorithm, I made a new function called testAIs and a new class object called geneticAlgorithm. What the geneticAlgorithm class would do was simply use the Tetris interface as a sort of testing engine. Given an input of board score weights, it would run the AI until it lost or reached a lineCap (which varied from 1000 up to 8000 as I was optimizing the AI) and then return the final score (or lines cleared) at the end. The genetic algorithm came into play with the testAIs function. The testAIs function randomly generated the weights of the relevant board values (given a range to randomly choose from) and then ran the AI. If the AI reached the line cap, it was considered a good AI and that list of weights was stored. Otherwise, it would not be stored. This was how the AI evolved, by using the good weights to optimize itself while neglecting the bad weights. The function then looped dozens of times (as many times as I wanted) and printed out the 2D list of good weighs at the end. I generally stopped collecting data when there were around 50 successful AIs. Then, I took the 3D list of weights (because I ran the testAIs function several times per iteration) and input them into a function that I created called analyzeData (in my geneticIterationDict.py file). This file took in all the input data and output the average of all the successful weights. I then used these weights as the middle of another, smaller range of weights to randomly choose between and then iterated the testAIs function again until I collected sufficient data. I performed seven iterations optimizing the AI, with AIs eventually having to reach a line cap of 8000 to be considered good. Running the genetic algorithm (in two windows) took around 25 hours and provided me with the optimized AI that I use for my project. Finally, I made a high score feature that keeps track of the high score (most lines cleared in a game by a player) and a high score feature that keeps track of the most difficult AI defeated and the time taken to defeat it. To do this, I wrote relevant data like the final score, etc. to a text file every time a game was ended and then read through the text file to find the highest scores.

UI

In designing the UI of my game, I drew from my experience playing Tetris online and playing games in general. My tetris boards were loosely based off of Tetris boards of games that I have played in the past. I added a blue gradient background because blue was voted to be the most popular favorite color in our 112 lecture and a gradient looked much more pleasing than a simple background. I then chose a color scheme that was not too harsh on the eyes that fit the blue background that I chose. I wanted to make my Tetris pieces look more aesthetically appealing so I made them look somewhat 3-dimensional by drawing slightly darker blocks offset beneath the original blocks. I also made a help page to help users learn what Tetris is about and learn the controls of the game (which the user could access by pressing ‘h’ or clicking on the question mark). One could return from the help screen to the previous screen by clicking the return button or pressing ‘b’. I also made it very easy to navigate between the different screens/pages of my project. I gave the opportunity to return to the home screen to start a new game or try a different mode (which the user could access by pressing ‘q’ or clicking on the home button). In my home screen, I gave the opportunity to start a new game (‘n’) or continue a game if one was already started (‘c’). If one started a new game, they were brought to a selection screen where they could select what kind of game they wanted to play (solo or vs., with or without a puzzle board, playing against an AI or another player or having two AIs play each other) and made it easy to customize the game with clear buttons and sliding bars (for the AI difficulty/intelligence). To start the game, a user could click start or press ‘ENTER’. Within the game, I made gameplay simple and a user could pause by pressing ‘p’ or restart by pressing ‘r’ (after winning, losing, or pausing). Finally, several of my friends and I found it rather satisfying watching my fastest AI play as quickly and effectively as possible, so to demonstrate the capabilities of my AI, I gave the option to spectate a “God AI” play.