Tetris Architecture Overview

Nick Parlante, nick.parlante@cs.stanford.edu

This is the architectural overview that describes the Tetris classes. The tetris project lives at the Stanford CS Ed library as document #112 (http://cslibrary.stanford.edu/112/). See the Readme for an introduction to the project and instructions for running the program.

These Tetris classes provide a framework for a few different Tetris applications. For a fun little assignment, students can write their own "brain" AI code (surprisingly easy), and then load it into the provided JTetris GUI to see how it plays. For 2nd year CS undergraduates, we have a fairly difficult assignment where they implement all the main tetris classes. This makes a nice exercise in OOP decomposition -- dividing the rather large Tetris problem into several non-trivial classes that cooperate to solve the whole thing.

The classes that make up Tetris are...

- Piece -- a single tetris piece
- Board -- the tetris board
- JPieceTest / JBoardTest -- tester classes for Piece and Board
- JTetris -- present the GUI for tetris in a window and do the animation
- LameBrain -- simple heuristic logic that knows how to play tetris
- JBrainTetris -- a subclass of JTetris that uses a brain to play the game without a human player

As an additional feature, the tetris classes implement the logic for tetris in a way that runs **quickly**. The speed is needed for the later "brain" parts of the project.

Piece

The Piece class defines a tetris piece in a particular rotation. Each piece is defined by a number of blocks known as its "body". The body is represented by the (x, y) coordinates of its blocks, with the origin in the lower-left corner.

So the body of this piece is defined by the (x, y) points : (0, 0), (1, 0), (1, 1), (2, 1).

The Piece class and the Board class (below) both measure things in this way -- block by block with the origin in the lower-left corner. As a design decision, the Piece and Board classes do not know about pixels -- they measure everything block by block. Or put another way, all the knowledge of pixels is isolated in the JTetris class.

Each piece responds to the messages like getWidth(), getHeight(), and getBody() that allow the client to see the state of the piece. The getSkirt() message returns an array that shows the lowest y value for each x value across the piece ({0, 0, 1} for the piece above). The skirt makes it easier to compute how a piece will come to land in the board. There are no messages that change a piece -- it is "immutable". To allow the client to see the various piece rotations that are available, the Piece class builds an array of the standard tetris pieces internally -- available through the Piece.getPieces(). This array contains the first rotation of each of the standard pieces. Starting with any piece, the nextRotation() message returns the "next" piece object that represents the next counter-clockwise rotation. Enough calls to nextRotation() gets the caller back to the original piece.

Board

The Board class stores the 2-d state of the tetris board. The client uses the place() message to add the blocks of a piece into the board. Once the blocks are in the board, they are not connected to each other as a piece any more; they are just 4 adjacent blocks that will eventually get separated by row-clearing.

- place(piece, x, y) -- add a piece into the board with its lower-left corner at the given (x, y)
- clearRows() -- compact the board downwards by clearing any filled rows

Board has many methods that allow the client to look at the Board state. These all run in constant time -- makes things easy and fast for the client, but harder for the board implementation...

- int getWidth() -- how many blocks wide is the board
- int getHeight() -- how many blocks high is the board
- int getRowWidth(y) -- the number of filled blocks in the given horizontal row
- int getColumnHeight(x) -- the height the board is filled in the given column. This is 1 + the y value of the highest filled block
- int getMaximumHeight() -- the max of the getColumnHeight() values
- int dropHeight(piece, x) -- the y value where the origin (lower left corner) of the given piece would come to rest if the piece dropped straight down at the given x

Board Undo

The Board also supports a 1-deep undo() facility that allows the client to undo the most recent placement and/or row clearing. The undo facility is rather limited -- the client can do a single place() and a single clearRows(), and then use undo() to return to the original state. Although simple, the undo facility is fast. It turns out that the Brain (below) needs exactly this facility of a simple but fast undo. Here's how undo works...

- We'll designate the state of the board as "committed" -- a state that can be returned to.
- From a committed state, the client can do a place() operation that modifies the board. An undo() will remove the change so the board is back at the committed state.
- Then the client can do a clearRows() operation which further modifies the board. An undo() will remove all the changes so the board is back at the committed state.
- Finally, the client can do a commit() operation which marks the current state as the new committed state. The previous committed state can no longer be reached via undo().

A client can use the undo facility to animate a piece moving in the board like this: place() piece with y=15, pause, undo(), place() with y=14, pause, undo(), place() with y=13, ...

JPieceTest, JBoardTest

These are simple test classes that exercise the Piece and Board respectively. Tetris is complex enough that it's important to build and test the components separately.

JTetris

The JTetris class presents the GUI for a playable tetris game in a window. It uses Piece and Board to do the real work. Usually, I have the students write Piece and Board, but I provide JTetris.

JBrainTetris

JBrainTetris is a subclass of JTetris that uses the LameBrain (below) or another loaded brain to play tetris without a human player. JBrainTetris can also implement an "adversary" feature. The adversary is a cruel yet hilarious feature where the game figures out what the worst possible next piece is (using the brain), and then gives that piece to the player.

LameBrain

LameBrain includes heuristic logic that knows how to play the game of tetris. The LameBrain algorithm is very simple -- it knows that height is bad and creating holes is bad. Students can write their own brains. Here is the code...

```
/*
 A simple brain function.
 Given a board, produce a number that rates
 that board position -- larger numbers for worse boards.
 This version just counts the height
 and the number of "holes" in the board.
 See Tetris-Overview.html for brain ideas.
*/
public double rateBoard(Board board) {
 final int width = board.getWidth();
 final int maxHeight = board.getMaxHeight();

 int sumHeight = 0;
 int holes = 0;

 // Count the holes, and sum up the heights
 for (int x=0; x<width; x++) {
  final int colHeight = board.getColumnHeight(x);
  sumHeight += colHeight;

  int y = colHeight - 2; // addr of first possible hole

  while (y>=0) {
   if  (!board.getGrid(x,y)) {
    holes++;
   }
   y--;
  }
 }

 double avgHeight = ((double)sumHeight)/width;

 // Add up the counts to make an overall score
 // The weights, 8, 40, etc., are just made up numbers that appear to work
 return (8*maxHeight + 40*avgHeight + 1.25*holes);
}
```

It's pretty easy to write better brain logic, and that alone can be the basis of an assignment. Here's some suggestions for building a better brain (or don't look at these if you want to puzzle it out yourself) ...

- Height is bad
- Holes are bad
- Stacking more blocks on top of holes is bad

- Holes that are horizontally adjacent to other holes are not quite as bad
- Holes that are vertically aligned with other holes are not quite as bad
- Tall 1-wide troughs are bad
- 1-wide troughs are not so bad if they are only 1 or 2 deep. Think about which pieces could fill a 2-deep trough -- 1, 2, or 3 out of the 7 pieces depending on the two sides of the trough.
- Concentrate on issues that are near the current top of the pile. Holes that 10 levels below the top edge are not as important as holes that are immediately below the top edge.
- At some point, my brain code always has some arbitrary constants like 1.54 and -0.76 in it that I can only lamely optimize by hand. To get the best possible brain, use a separate genetic algorithm to optimize the constants. This is another reason why the design here has such an emphasis on speed -- the genetic optimizer needs to be able to rip through millions of board positions.