Sagar Doshi

UID: 604901376

Discussion-TA: 1A-N. Abani

May 17, 2018

Project 3 Report

1. Function/Design Descriptions:

The first approach I took to solve this problem was to treat each piece as a highly sophisticated object that handled collision depending on its bounding box, and I had planned to use inheritance to differentiate the collision properties of each piece along with dynamic memory allocation. **This was hard so I took a different, simpler approach**.

This simpler approach avoids the usage of inheritance, and even circumnavigates the necessity to implement destructors/assignment operators/copy constructors, or do extensive memory management. This approach also allows for a lot of code to be re-use with very slight modifications for boundary-checking/collision checking.

The game class uses the well and piece classes to run the Chetyris game, and thus it handles the user input as well as display. The only dynamic memory management of my entire project occurs in game’s playOneLevel() function where new instances of pieces are created, and they are deleted once the player wins one level, quits, or moves onto the next piece.

The Well class uses the piece class to edit certain properties of the game such as moving pieces, printing itself, handling collision, changing piece orientation, exploding special pieces, etc. the “meat” of the program is in this class. The well does all this based upon the type of Piece it is given, so it makes heavy usage of the piece class’s getters and setters. The well’s only member variable is a 2D array of characters 10 units wide and 18 units deep.

The piece class is SUPER SIMPLE. Each piece has an x and y coordinate in the well represented by int, and these coordinates represent the bounding box’s upper-left corner being at (x,y) in the well. Each piece also has an orientation (0,1,2,3) stored in a integer member variable, and each piece also has a PieceType member variable that represents the type of piece. The only member functions of the piece class are getters and setters for each of the member variables, however there is no setter for the member variable of the PieceType as that should NEVER be modified. As Piece does not utilize dynamic memory allocation, a custom destructor for this class doesn’t have to be written as the default destructor works perfectly.

NONE OF MY MEMBER FUNCTIONS ARE DECLARED VIRTUAL. The collision handling is done by the well, and that only depends on the enumerated type PieceType of the passed piece, so doing inheritance is a gross overcomplication as literally everything about a piece except its PieceType is the same when compared to another Piece object. The collision and out-of-bounds checking is done with the 2D array of the well based upon p->get\_type() and p->get\_orientation() as well as the pieces coordinates. Well does all the interpretation based upon the piece it is given.

**Member functions of Game:**

Game(int width, int height) – constructor provided, I didn’t modify this at all other than to initialize the new member variables I added to the appropriate values. (ex: m\_score)

Void Play(); -Contains the outer structure of the game with incrementing levels and displaying the game over screen/congratulations screen. Didn’t modify this at all

bool playOneLevel(); - contains the structure of one level. First the timer, rows left, and other initial conditions are initialized. Then a new Piece is created (it is deleted at the end of the function.) Another piece pointer is set to null, the two pointers are needed as the next piece needs to be displayed. The function checks to see if the user has reached the number of vaporized lines and returns true if so (true represents a completed level). The function then tries to add a piece using well’s member function addPiece(), and if this returns false, it means that the pieces have reached the top and user has lost. If the user hasn’t lost, the user can interact with the piece per time increment. After each time increment the piece is moved down until it comes to rest. Then if the piece is an exploding piece, it is blown up, or if it’s not special it is converted to ‘$’. Finally, the filled rows are vaporized and the score is updated. As the keyword ‘new’ is used, the function deletes any allocated memory depending on the exit condition (quit, next level, loss).

displayPrompt(std::string s) – didn’t edit this at all

displayStatus() – Uses the screen to display the score, lines left, and the words ‘Next Piece: ’

bool keyboardInput() – utilizes the UserInterface .h file to handle user keyboard input. Is just a giant switch statement that executes the proper command (move left, right, down, etc.) based upon what the user inputs. ALWAYS returns true unless the user hits q or Q, and then returns false which causes the program to terminate.

Void displayNextPiece(Piece \* p) – used to display the next piece w/bounding box at screen coordinate (16,4)

I added the member variables (int m\_score) and (int\_m\_rows\_left) to keep track of score and the number of rows the player had to vaporize.

**Member functions of Piece:**

Only getters and setters for the member variables m\_orientation, m\_x, m\_y, and m\_type. Note that there is no setter for m\_type, as a piece’s type should never be changed.

It is important to note that the constructor always sets the coordinates of the piece to (3,0) in the well, and its orientation by default is always initialized to 0. As a piece must have a type, the constructor must be passed a pieceType to assign to the piece.

There was no real need for inheritance because the only difference between different piece objects is their member enumerated PieceTypes, and using the member PieceTypes and orientations, my well class could differentiate between the passed pieces and perform the desired operations.

**Member functions of Well:**

void display(Screen& screen, int x, int y) – prints out the well and the surrounding ‘@’ boundary

bool addPiece(Piece\* p) – attempts to add a piece to well by modifying the characters of the 2D array depending on the type of piece passed. It checks for collision specific to the type of piece, and if the piece would collide with something if it is added, it returns false signaling the end of the game.

bool print\_to\_well(Piece \* p, char c) – modifies the well depending on the piece passed. If ‘#’ is passed, it means this function was used to move the piece somewhere, if ‘$’ was passed, it means the piece is coming to rest, and if ‘ ’ is passed, it means the piece is being ‘removed’ from the well

bool isOccupied(int x, int y) – simple function to check if the passed coordinate in the well is out of bounds or not ‘ ’ implying a potential collison. If there is something at the passed coordinate, returns true, otherwise return false.

bool isRowFull(int y)- iterates through a row at the passed y coordinate and checks if every piece is occupied, if so returns true. Otherwise returns false.

Void removeRow(int y)- removes the row at the passed y-coordinate (represents vaporation of the row by the game class)

bool move\_down(Piece\* p);

bool move\_left(Piece\* p);

bool move\_right(Piece\* p);

These functions are ENORMOUS switch statements with nested if-else statements to check for collisions/out-of-bonds on a piece-by-piece-by-orientation basis. They return true if the piece can be moved and thus moves the piece, and returns false and doesn’t move the piece if a collisions or out-of-bounds results. These are the ‘meat’ of the program.

bool move\_all\_the\_way\_down(Piece\* p) – Repeatedly calls move\_down until the piece can’t move down anymore

bool blowUpVaporBomb(Piece \* p) – delete a 2 x 5 portion of the well dependent on where the vapor bomb piece comes to rest. It has built-in checking for out of bounds errors (for example if the vapor bomb is at the bottom it doesn’t make sense to delete spaces below it.)

bool blowUpFoamBomb(Piece \* p, int x, int y) – Explodes the foam bomb by utilizing the recursive maze algorithm used previously. The only modification to the maze algorithm is that:

*based on the coordinate passed to the function, the recursive case terminates if either the next ‘move through the maze’ would be out of bounds or if the next move would be outside of the 5x5 box specified in the spec. The explosion can only move if the next piece is ‘ ’*

bool change\_orientation(Piece\* p) – Enormous switch statement with nested if-else statements that returns true if the orientation of a piece can be changed, and returns false if the orientation of a piece cannot be changed. Just like the move functions, A LOT of collision/out-of-bounds checking.

void cleanWell(); sets all the characters in the 2D character array representing the well used by the game to ‘ ’. This is called by the game when the user completes a level as in the spec it states that the well should be empty whenever the user starts a new level.

The only member variable of the well class was a 2D character array representing the well itself. All the collision checking, out-of-bounds checking, and modifications to the display of the well are done internally through this array.

2. There weren’t any functions/features I was unable to implement, and as far as I know I couldn’t find any more bugs though I’m sure whoever grades this will find some bugs :c

3. The only portion of the design document that was ambiguous was that it states at bullet point 20

*“When the player hits the space bar, the piece immediately descends to the position it would end up at if the user had instead hit the down arrow key repeatedly.* ***It has come to rest****.”*

The last portion is what seems odd to me. In traditional Tetris, when the user hits the button corresponding to “go-all-the-way-down”, they can still interact with the piece for a limited amount of time. When the user hits the down key, the design spec specifically states to reset the timer so that the user can interact with the piece.

This contrasts with the spacebar as in the project spec, it says that it has come to rest, so in my implementation, I made it so that if the user hits space bar, he/she CANNOT interact with the piece anymore. The user keyboard interaction is restored when the next key is added, and DiscardPendingKeys() is called in case the user was mashing the keyboard while the piece was coming to rest.