**Nikki Woo 204927670**

**CS32 Project 3 Report**

**Public Member Functions of Game:**

**Game::play()**

Initializes a 2d vector of space characters that will be used to keep track of the board screen throughout the game, displays the appropriate well, initial score, level, and rows left, and prompts the player to press enter to begin playing the game.

Begins an infinite loop that evaluates if a player passed one level (playOneLevel) function. If the player does pass, the screen prompts the player to press enter to advance to the next level. The appropriate level, score, and rows left are updated on the screen and the playOneLevel function is called again for the subsequent level. If the player does not pass the level, the infinite loop is broken and the screen informs the player the game is over.

This is inside the Game host class because it modifies private data members of game including m\_screen and m\_level. It is not virtual because there are not multiple “types” of games needed and thus there is not a need to “play” different ways.

**Game::displayStatus()**

Display the appropriate score, rows left, and level according to the private data members of the Game.

This is inside the Game host class because it modifies m\_screen- a private data member of Game. It is not virtual because there is only one Game that we are playing and no need for multiple ways to display the status.

**Game::playOneLevel(vector<vector<char>> b)**

Accepts the 2d vector of characters as a parameter to modify throughout the level. In the state it accepts the 2d vector, it is an empty well. This 2d vector is not pass by reference so that each new level begins with an empty well. Initializes a boolean q that will be used later to check when/if a player decides to quit the game by using the ‘q’ keyboard command and two pieces.

Begin a while loop that continues if rows left to be completed by player is not zero. Initializes an integer to 0 that tells the function the index of any filled rows, allowing that row to be deleted according to the rules of the game. Initializes an integer to 0 that counts how many rows are destroyed at once that allows the score to be updated accordingly. This function uses the while loop to continuously call pieces as long as the level is not yet passed or the piece can no longer display. In the case that this is the first time through the function, evaluate if the initial piece is not yet initialized (if its piecetype is the default value). If it is not yet initialized, a random piecetype is initialized. In the case that it is already initialized (in which it would have been initialized to match the nextpiece’s type that is displayed on the screen, it continues on. Initializes the next piece to a random piecetype. If a new piece is able to be called (in which initializing a piece on the screen will not coincide with any existing fallen pieces on the screen), the initialized next piece is called to display on the screen and the initialpiece is called via the callPiece function which drops the piece into the well and allows player interaction with the piece. If a new piece is not able to be initialized without interacting with/overriding any already placed pieces, the piece displays but the function returns false. After the running of this function, (meaning that the piece is placed somewhere in the well and is stationary) the function checks to see if the aforementioned boolean named q was set to false within the callPiece function which meant that the player prompted the game to quit. If the player wished to quit, the function exits and the previously mentioned infinite loop in Game’s play function is exited and the game over prompt is called. Otherwise, with the newly placed piece, the function must check the screen for any filled rows. This is done using a while loop calling the rowsfilled function that checks the screen for rows filled and modifies the integer variable declared earlier that returns the index of a filled row. While there are still filled rows (in which the rowsfilled function will return true), the function deletes the row corresponding to the index of the filled row that the rowsfilled function changes. Then, still within the while loop, the integer that counts how many rows are destroyed at once is updated, the rows left integer is decremented while it is greater than zero (which prevents the rows left integer from being negative in the case that multiple rows are cleared when there is only one row left to be cleared), and the screen is refreshed with the new number of rows left. Once all filled rows are deleted, the score is refreshed based on how many rows were destroyed at once and the initial piece is set to be equal to the next piece (to match the next piece already displayed on the screen). This process is repeated as long as the rows left to be completed by the player is greater than zero, the ‘q’ key is pressed (thus quitting the game), or the next piece is not able to be displayed on the screen (in which the game is over). At the end of this while loop, if the player successfully completes the level (in which the rows left integer is zero), the playOneLevel function returns true. Otherwise, it returns false and the game is over.

This function was placed in the Game host class because it utilizes and modifies many of Game’s private data members such as the number of rows left, the screen, and the score. This function is not virtual because there is no need for multiple ways to declare it as there is only one way to play the level for this specific game.

**Game::callPiece(vector<vector<char>>& b, BBPiece& p, int x, int y, int tt, bool& q)**

This function is used to call a piece meaning that the piece passed in is implemented to drop onto the board- letting the player control its movements as specified in the spec sheet. It accepts the 2d vector of characters a parameter which allows us to keep track of where/how the called piece can move on the screen. It also accepts the piece that we are moving on the board as well as the initial coordinate point of the piece (x, y), an integer tt that will be used to keep track of timing of the piece dropping, and a boolean q which is used to indicate if the player prompts the game to quit via the ‘q’ key command.

Check to see if it is valid for the called piece to move down on the screen. If it is not valid, it places the piece on the 2d vector to keep track of what is on the screen and turns the ‘#’ on the screen displaying the falling pieces to ‘$’. It then checks to see if the piece p is a vapor bomb or foam piece. If so, it calls the corresponding functions that perform the special actions of the vapor and foam pieces as specified in the project spec. However, if it is valid for the piece to move down, the piece is displayed on the screen and a timer is started. While the timer is at a time less than the amount of time specified in the project description, there is a switch statement that accepts keyboard input and translates it to moving the piece down, left, right, rotating the piece, bringing the piece all the way down to the bottom of the well, or quitting the game. For all commands except the quit and space command, we first check if it is allowable to move the piece in the way specified. If so, we erase the piece displayed and call the piece again using recursion for the callPiece function but pass in the time elapsed from the timer called earlier as the int tt in order to account for time passing between the user inputting commands. In the case that the piece passed through is the crazy piece, the code to move the piece right and move the piece left are swapped. For the case that the user presses space to bring the piece all the way down, the piece is erased from the display and the y coordinate of the bounding box is incremented as long as it is valid for the piece to move down and the piece is then displayed at that point. If the piece is a foam piece, the foam function is called to allow the foam piece to “burst” with foam. If the piece is a vapor bomb, the vapor function is called to “explode” the piece. If the user presses the quit command, the q boolean passed in is changed to false and the callPiece function returns false. After the timer elapses to the max allocated time, the piece is erased and moves down one and the callPiece function is called again for piece with an incrememted y coordinate for its bounding box.

This function was implemented in the Game class because it modifies the screen private data member of the Game class. This function is not virtual because, though there are different pieces and they are different in some ways, they inherently all move and interact with the board in the same basic manner.

**Game::rowsfilled(vector<vector<char>>& b, int& filledindex) const**

This function is used to evaluate if the board has filled rows. It does so by checking every individual row for a consistent non-space line of characters. If it does, it stores the index of the first filled row that it has into the filledindex integer and returns true. This function is constant because it does not modify any of Game’s private data members.

This was placed in the Game host class because it mostly fits in with the functionality of the game versus say, the functionality of a piece. It is not virtual because there is only one way to check rows in the extent of the spec given to us.

**Game::deleterows(Screen& s, vector<vector<char>>& b, int& index)**

This function deletes the filled row based on the index of the filled row, shifts all the subsequent rows down, and then initializes the top row to ‘ ‘. This is inside the Game class because it modified the private data member screen of the game. This is not virtual because there is only one way to delete rows in the extent of the spec given to us.

**Game::refreshScore(int rowdel)**

This function refreshes the score based on the number of rows deleted at once (given by the rowdel integer). This is in the Game class because it modified m\_score and is not virtual because there is only one way to refresh the score in the extent of the spec given to us.

**Game::clearScreen()**

This function clears the screen for the start of a new level so that there are no longer any pieces in the well. This is in the Game class because it modifies m\_screen and is not virtual because there is only one way to clear the screen in the extent of the spec given to us.

**Public Member Functions of BBPiece:**

**BBPiece::BBPiece()**

Initializes 2d vector bounding box of piece to all spaces (will change later based on what piecetype is passed into the initializepiece function. Sets piece type to initial NUM\_PIECE\_TYPES (used in Game to check if piece is initialized yet). Sets orientation to 0. I did not make this virtual because, the way I defined pieces, they each have a bounding box based on its 2d vector private data member. The inherent differences between pieces is defined in the way I fill out the 2d vector with ‘#’ components. All pieces are of the same class.

**BBPiece::ptype() const**

Returns piece type, in BBPiece class because it accesses the private data member of the piece. Not virtual because there is only one way to do this.

**BBPiece::operator=( const BBPiece& og)**

Assignment operator, changes private data members of this to match that of og, in BBPiece class because it accesses the private data members of piece. Not virtual because there is only one way to do this.

**BBPiece::initializepiece(PieceType rand)**

Initializes piece in that the 2d vector private data member is updated to have ‘#’ components where appropriate for the passed in piece type and sets the piece type the to the passed in value. This is in the BBPiece class because it modifies the 2d vector private data member. This is not virtual because it already accounts for all the different piece types.

**BBPiece::display(Screen& s, int x, int y) const**

Displays the piece onto the screen of the Game. This is in the piece class because it evaluates the 2d vector of the piece for the location of ‘#’ within the pieces. It is not virtual because there is only one way to display the pieces in the extent of the spec.

**BBPiece::printNextPiece(Screen& s, int x, int y) const**

Prints out the next piece under the Next Piece: slot on the screen of the game. This is in the piece class because accesses the 2d vector private data member of the piece. It is not virtual because there is only one way to print the next piece in the extent of the spec.

**BBPiece::erasepiece(Screen& s, int x, int y) const**

Erases the piece on the screen of the game according to the corresponding coordinates of the 2d vector of the piece itself. This is in the piece class because it accesses the 2d vector of the piece. It is not virtual because there is only one way to erase a piece in the extent of the spec.

**BBPiece::isValidDown(vector<vector<char>>& b, int x, int y) const**

Checks to see if it is valid for the piece to move down one row below its current state by checking the slots on the 2d vector representing the board that will be affected by displaying the piece one row down via the 2d vector representing the piece. If able to move down, return true- false if not. This used the host class BBPiece because it accesses the 2d vector representing the piece. It is not virtual because, in the extent of the spec, there is only one way to move a piece down one row.

**BBPiece::isValidRight(vector<vector<char>>& b, int x, int y) const**

Checks to see if it is valid for the piece to move right one row next to its current state by checking the slots on the 2d vector representing the board that will be affected by displaying the piece one row to the right via the 2d vector representing the piece. If able to move right, return true- false if not. This used the host class BBPiece because it accesses the 2d vector representing the piece. It is not virtual because, in the extent of the spec, there is only one way to move a piece to the right one row (besides the way of the crazy piece for which, in my opinion, it is easier to account for the crazy piece case in the function rather than creating a new class for the crazy piece).

**BBPiece::isValidLeft(vector<vector<char>>& b, int x, int y) const**

Checks to see if it is valid for the piece to move left one row next to its current state by checking the slots on the 2d vector representing the board that will be affected by displaying the piece one row to the left via the 2d vector representing the piece. If able to move left, return true- false if not. This used the host class BBPiece because it accesses the 2d vector representing the piece. It is not virtual because, in the extent of the spec, there is only one way to move a piece to the left one row (besides the way of the crazy piece for which, in my opinion, it is easier to account for the crazy piece case in the function rather than creating a new class for the crazy piece).

**BBPiece::isValidRotate(Screen& s, vector<vector<char>>& b, int x, int y)**

First rotates the piece using a temporary 2d vector and checks to see if it is valid for the piece to rotate clockwise from its current state by checking the slots on the 2d vector representing the board that will be affecting by displaying the piece one row down via the 2d vector representing the piece. If able to rotate, it rotates the piece on both the 2d vector board and the screen and increments the orientation integer of the piece used to account for the different ways the pieces rotate

Since different pieces rotate in different ways, the function checks the type of the piece before deciding if the piece even rotates and how it will rotate. This used the host class BBPiece because it accesses the 2d vector representing the piece as well as the piece type. It is not virtual because, though there are different ways to rotate different pieces, I found it easier to account for the differences in piece types in a single rotation function rather than implementing new class for each different type of rotation.

**BBPiece::isGameOver(vector<vector<char>>& b, Screen& s) const**

Used to check if dropping a new piece in the well will cause it to clash with currently existing pieces in the well. If so, the function returns false. Otherwise, return true. This is the BBPiece host class because it accesses the 2d vector representing the piece. It is not virtual because, to the extent of the spec, there is only one way to check if the piece clashes with already existing pieces.

**BBPiece::foamy(Screen& s, vector<vector<char>>& b, int x, int y, int rightbound, int downbound) const**

Used to implement the foam piece special function that explodes “foam” onto the board in the way described in the specification. Using recursion, the slots adjacent to the foam piece and the slots adjacent to those pieces are checked to see if the \* can be passed into the screen without overwriting existing pieces or the well wall. This recursion happens until a certain point- a bounding 5x5 box around the foam piece.

This is in the BBPiece host class because its function relates most to the foam piece, which is a piece. It is not virtual because, to the extent of the foam piece as described in the spec, there is only one way to explode the foam piece.

**BBPiece::vapor(Screen& s, vector<vector<char>>& b, int x, int y) const**

Used to implement the vapor bomb special function in the way described in the spec. Checks to see if the two rows above the vapor bomb bounding by the columns that the vapor bomb is in have any existing pieces. If so, deletes those fragments.

This is in the BBPiece host class because its function relates most to the vapor bomb, which is a piece. It is not virtual because, to the extent of the vapor bomb as described in the spec, there is only one way to explode the vapor bomb.

**BBPiece::cash(Screen& s, vector<vector<char>>& b, int x, int y) const**

Used to turn pieces that fall all the way down into the well into $, putting it permanently on the board (unless deleted by filling a row or the vapor bomb).

This is in the BBPiece host class because its function relates most to pieces. It is not virtual because, to the extent of the pieces as described in the spec, there is only one way to change # to $ and have the piece come to rest

**Public Member Functions of ScreenImpl:**

**ScreenImpl::justoutput(int m\_data)**

Used to right justify output in a block of seven places (as described by the spec). This is done by counting how many characters are taken up in the desired text to be printed out and printing out the number of spaces of 7 minus the size of the desired text.

This is in the ScreenImpl host class because its function relates most to the screen implementation (printing things out onto the screen). It is not virtual because, to the extent of the spec, there are not multiple ways we wish to implement this.

**Public Member Functions of Screen:**

**Screen::justouput(int m\_data)**

Calls the justouput function via the screen implementation pointer of the screen class.

This is in the Screen host class because it directly relates to the screen class. It is not virtual because, to the extent of the spec, there are not multiple ways we wish to implement this.

**Public Member Functions of Well:**

**Well::display(Screen& screen, vector<vector<char>>& b, int x, int y) const**

Displays the walls of the well in the way specified by the project.

This is in the Well host class because it directly relates to the well. It is not virtual because, to the extent of the spec, there are not multiple ways we wish to implement this.