**PROJECT 3 REPORT**

1. A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual. For example, "I chose to define a pure virtual version of the sing() function in my base Piece class because all Pieces sing, but each kind of piece sings in its own way."

**Added Function Implementations in Game.cpp**

Game::Game(int width, int height)

I made no changes to this function.

void Game::play()

I made no changes to this function.

void Game::displayStatus()

This function displays the game status by printing the current level, score, and number of rows left as right justified strings. To do this, it relies on a helper function printStatusItem(), explained below. In addition, it displays the next piece to fall by accessing the back of the queue in which the pieces are stored (a first in, first out data structure).

void Game::printStatusItem(std::string label, int value)

This function serves as a helper function for displayStatus() and provides a generic algorithm for creating a right justified string composed of the status label (i.e. “Level: ” ) and its corresponding value (i.e. “5”), filled with the correct number of spaces to ensure uniformity in the display. The algorithm uses the length of the words to add in the proper number of spaces and meet formatting requirements.

bool Game::levelPassed()

This function performs a simple check, returning true if the number of rows vaporized is greater than or equal to the current level \* 5, as described in the spec. Otherwise, it returns false. This check is performed in the playOneLevel() code, which will continue executing until the level is passed.

bool Game::playOneLevel()

This function contains the code to play one level, returning true once the level is passed. It begins by pushing two random pieces to the queue and displaying the initial status. While the level has not yet been completed, it adds the current piece to the well, checks for collisions, and executes the falling of the piece using the timer, as implemented in the provided skeleton. It waits for user input and calls the corresponding movement functions (implemented in well) or, in the case of ‘q’, exits the program.

NOTE that the reversed movement of the Crazy Piece is dealt with here, as the cases denoting left and right movement first check the piece type before calling any movement functions. In the case of the crazy piece, the left/ right user inputs simply call the opposite movement functions.

All of these functions are implemented in the Game class because they relate directly to the status or execution of the game as a whole.

**Function Implementations in Piece.cpp**

In my program, I made the decision to have a SINGLE piece class, containing the common properties shared by all pieces. The class itself is fairly simple, managing the initial construction of the piece within its grid, its display, and a couple additional functions to check and adjust certain properties.

NOTE: The movement and special actions of the pieces’ are implemented in well. This is explained below.

Piece::Piece(PieceType type)

The implementation of the piece constructor sets the type of the piece (chosen at random) and manages the initialization of a four by four grid within which each piece is bound. Within the constructor body, the function sets the coordinates of the piece to those of orientation #0 using a function setOrientation(), explained below. This constructor is used to construct ALL pieces, as all have a type, exist in a 4x4 grid, and are initialized to orientation 0.

void Piece::setOrientation(int orientation)

This function sets the member variable m\_orienation and then assigns the coordinates of the specified orientation within the piece’s 4x4 grid. It checks the piece type then operates based on a switch statement, hard coding the specific piece coordinates for the case of each orientation.

void Piece::display(Screen& screen, int x, int y)

This function simply displays the piece, going to the proper x,y position on the screren and looping through the piece’s 4x4 grid, printing the filled coordinates and therefore displaying the piece on screen.

void Piece::erase(Screen& screen, int x, int y)

This function operates similarly to the display function, looping through the 4x4 grid and overriding the filled coordinates with the empty character ‘ ’, in order to reset the grid. This function is used in the circumstances that the piece must be redrawn, such as in the case of rotation.

bool Piece::isCoordinateFilled(int x, int y)

This function does a simple check to see if the inputted coordinate is occupied by the current piece. This is a helper function to increase the readability of code later on, when the comparison of coordinates occupied by the current peace and coordinates filled within the well becomes a common operation (such as when checking for collisions).

Plus two accessor functions defined and declared in Piece.h:

int getOrienation() const

Simply returns the current orientation of the piece. This function is used within the Well class, in order to rotate the piece by storing and then incrementing the current orientation.

PieceType getType()

Simply returns the piece type of the current piece. This function is used within the Well class, to check for the need of a special action once the piece has come to rest.

It should be noted that my personal implementation of the pieces DOES NOT rely on an inheritance structure, and therefore does not involve virtual or pure virtual functions. I recognize that another possible implementation would have been to create a base class for Piece, with derived classes for special pieces and normal pieces. This implementation would rely on virtual functions for the purpose of overriding functions that should be defined uniquely in the derived classes, such as the movement or action of a special piece. However, as stated before, I made the choice to keep my Piece class as simple as possible -- including only the most basic, common properties of each piece. Most of the manipulation, movement, and special actions of pieces occurs in Well -- a decision that I made in order to localize all need for coordinate comparison to a single class.

**Function Implementations in Well.cpp**

This class contains the implementations for all manipulation/ movement of pieces, the vaporization of rows, and the management of an overarching coordinate system within which all pieces and boundaries exist.

NOTE: The entirety of my collision checks relies on a vector named filledCoords which exists as a member variable of well and at any point contains a list of every single occupied space in the well, including its outer boundaries.

Well::Well()

My well constructor initializes a 12x18 grid within which the well exists, including its boundaries. The code sets the boundary positions, existing at row 18, column 0, and column 12 to ‘@’. It adds these coordinates to filledCoords, marking them as occupied and therefore making it illegal for any piece to attempt to move into these positions.

void Well::display(Screen& screen, int x, int y)

My display function for well loops through the grid in which the well exists, printing each index of the grid to the screen at position x,y and refreshing the screen once complete.

void Well::newPiece(Piece\* piece)

This function passes in a pointer to the next piece to fall, setting it to well’s member variable m\_piece, which points to the current falling piece for the purpose of checking collisions.

It also sets the current piece’s x y position within the well’s coordinate system -- specifically the (x,y) coordinate of the TOP LEFT corner of the piece grid within the well. Note that the y position starts at -1, because the piece will fall immediately to the position of 0.

bool Well::willCollide(int xPos, int yPos)

This function manages the checks for collisions that occur before any piece can be moved. It takes in x,y parameters of the piece’s tentative, NEW position. It then performs the check by translating the filled coordinates of the piece within its bounding 4x4 grid into coordinates of the well’s coordinate system. If any of these coordinates already exist in well’s record of occupied spaces are found, then that coordinate is already filled and the piece cannot move into that location.

The function returns a boolean, true if the position of the piece will not collide with any existing filled spaces in the well.

bool Well::rotatePiece(Screen& screen)

This function manages the rotation of well’s current piece (the piece that is falling). It first checks if the rotation is possible, tentatively setting the piece’s new orientation and checking if it will overlap with any of the already filled spaces in the well (recorded in filledCoords vector). The orientation is changed from 0 -> 1 -> 2 -> 3 -> 0 and so on, in a circular manner. If a collision is detected, the piece reverts to its valid orientation. Otherwise, the piece is displayed with its new orientation and the screen is refreshed.

The function returns a boolean, indicating whether or not the rotation was successful.

bool Well::movePieceLeft(Screen& screen) AND bool Well::movePieceRight(Screen& screen)

These functions operate in similar ways, first checking for a collision between the piece in its tentative new position and any existing filled coordinates within the well. If a collision is detected, no movement occurs. Otherwise, the piece is erased from its current position and redrawn with the proper incremented/ decremented x or y position. The screen is refreshed, and the piece has moved.

The function returns a boolean, indicating whether or not the movement was successful.

bool Well::movePieceDown(Screen& screen)

This function manages manages not only the downward movement of the piece (due to gravity or specific user input) but also what happens when the piece comes to rest. The downward movement itself is executed in same way as the left/ right movements -- checking for collisions and redisplaying the piece in its incremented position if the move is deemed valid.

The second major part of this function determines what happens when the piece has come to rest -- that is, once the function has detected a collision.

If the current piece has a special action, then the piece performs its special action using a helper function.

The piece’s final coordinates are added to the well’s log of all occupied spaces and the function checks for filled rows, vaporizing them if possible using a helper function described below. The score is then updated depending on the number of rows vaporized at once.

void Well::changeToDollarSign(Screen& screen,int x,int y)

This function simply changes the coordinates of the current piece to ‘$’, once it has come to rest in the well. It is called in movePieceDown().

bool Well::isCoordFilled(Coord coordToCheck)

I added this function to make for cleaner collision checks, as I found myself performing duplicate code in many places. The function looks for a specific coordinate in well’s record of occupied spaces. If the coordinate is found, then moving to that coordinate would result in a collision and the function will return false. Otherwise, the function returns true -- indicating that the coordinate is unoccupied and could be filled by a piece.

bool Well::isRowFull(Screen& screen, int y)

This function checks to see if a particular row is filled, which would indicate that the row should be vaporized. It performs this check by looping through well’s list of filled spaces and keeping track of the number of coordinates found within the target row. If 10 spaces are detected as being full, then the row is filled and the function returns true.

void Well::vaporizeRow(Screen& screen, int yFilledRow)

This function vaporizes a row by shifting everything down, thus overriding the row. It then updates well’s record of filled coordinates by removing the coordinates of the now vaporized row and incrementing the remaining filled coordinates (to account for the downward shift).

void Well::vaporBomb(Screen& screen)

This function performs the special action of the vapor bomb. It records the coordinates affected by the vapor bomb (the coordinates of the piece itself plus all blocks up to 2 below and 2 above it) and checks whether or not these coordinates are filled in the well. If they are, they are removed from well’s record of filled coordinates (because the vapor bomb has ‘vaporized’ them) and the well is updated.

void Well::updateWell(Screen& screen)

This function is used to update the well when there has been a change made to its record of filled coordinates, as in the case of vaporization. It first loops through and clears out the well by overriding with empty characters, then displays the adjusted filled coordinates.

void Well::emptyWell()

This function empties the inside of the well by clearing grid spaces from well's record of filled coordinates. This occurs at the end of a level, when the game essentially needs to reset.

Plus four accessor functions defined and declared in Well.h:

int getNumRowsVaporized() const → allows Game to access the number of rows vaporized, which is used to determine when the level is passed

int getScore() const → allows Game to have access to the score, which is determined once the piece has come to rest in the well and rows are vaporized

int getxPos() const AND int getyPos() const → used by Game to check if the game is over, by checking to see if the initial x position and y position of a new piece are already overlapping with an existing filled coordinate in well (aka, the well is full and the piece cannot fall).

2. A list of all functionality that you failed to finish as well as known bugs in your classes.

I could not get the FoamBomb to perform its special action recursively, so for now it works just as a normal piece would.

3. A list of other design decisions and assumptions you made (e.g., It was ambiguous what to do in situation X, so this is what I decided to do.)

As far as I am aware, I followed the spec very closely and did not come across any ambiguous circumstances. My program seems to function exactly as demonstrated by the executable (besides the special action of the foam bomb).