Zheng Wang

Computer Science 32

Professor David Smallberg

May 22, 2018

Project 3 Report

**General Design** *(more detailed explanation can be found in* **Class Description**)**:**

1. When designing this game, I took the Piece.h as mostly an information storage file. In the beginning of the file, all the possible graph (plot as a 4 by 4 grid) of each piece are defined as constant 2D char arrays. This allow the program to easily access the standard template of all graphs and compare its copy to the current environment of the Well and make judgements of different actions (e.g. moving down, right …). I also declared two non-member functions (*pieceWrite* and *blockWrite*) that facilitate the copy of graph of each piece. Since the Piece most gives information, the piece itself does not actually move or rest, the Well will be major site that carries out the operations.
2. The Well is the “place” where most actions in Chetyris actually happens. The Well have a pointer to the current piece being played. Whenever an action is requested, the Well will check if that is valid, and if it is valid, execute it. The Well will also take care of fixing a piece.
3. The Layer is a private class of Well, the Well is made of many layers. The layers have many helper functions that can simplify the execution of the Well.
4. The Game is the part of the program that actually interact with the **player**. It provides Well with the next piece it is supposed to use and instruct the well to move down the piece when the time is up. It also responds to the commands from the **player** and send them to well to actually process. All the interface a **player** will see is credit to Game.

**Assumptions Made**:

1. That the prompt will not be overwritten until another prompt is issued by Game. For example, the initial prompt “Press the Enter key to begin playing Chetyris!” won’t go away after the **player** pressed Enter, it will only be overwritten if another prompt is displayed.
2. When a Level ends, the next piece for the previous level will be discarded and a new one will be issued, as the sample program behaves
3. When the level ends, the player will be able vaporizes more rows simultaneously than “rows left”. In this case, the “rows left” won’t go below 0 but the **player** get the full score by completing those rows
4. When a Level is end, Game won’t display the Well with completed rows cleaned, instead, the rows supposed to be cleaned will be displayed as full. However, these rows are actually cleaned and the scores will be updated.
5. That the Well is constructed with a valid size, and the initial position of the piece must not force the piece to be written outside the Well or on the well’s wall.

**Unfinished Task**:

According to all the tests carried out so far. The game is **fully functional and there is no unfinished implementation**.

**Class Description:**

**File: Piece.h**

1. class Piece (the base class, a virtual class)
   * 1. Class Piece’ s **public member**
        1. Piece(PieceType chosen, Well\* wl, const char graph[4][4]) The constructor of the virtual class, it store the PieceType of the piece constructed, the well it is in, and construct the piece’s graph using the a reference graph.
        2. virtual ~Piece() The destructor of the Piece type, it is virtual since the piece class itself is an abstract class, and by programming rule, it’s destructor must be virtual, otherwise it will be undefined behavior.
        3. void move\_down() The function that move down the piece, since the piece class only store information about the piece, the function tells the y position of the piece, defined by private member int m\_pos\_y to increment by 1. It is not virtual since all piece moves down in the same way.
        4. PieceType pieceKind() const The accessor function (so it is const) that identify the type of piece, it is not virtual since all the piece belongs to a kind and this should behave the same.
        5. int getX() const The accessor function (so it is const) that give the x coordinate of a piece. It is non virtual since the all piece have a coordinate.
        6. int getY() const The accessor function (so it is const) that give the y coordinate of a piece. It is non virtual since the all piece have a coordinate.
        7. std::string\* get\_graph() const The function that returns the graph of the piece. It is not virtual since all kinds of piece contains a graph. The return type is std::string\* since the graph is stored as a array of pointer to std::string class.
        8. int get\_orientation() const The function returns the orientation of the piece. It is non virtual since all of the piece have orientations
        9. int get\_next\_ori() const The function returns the orientation that a piece will have if it is rotated. This is not virtual since all the piece have an orientation, and therefore have an expected orientation after rotation.
        10. virtual int leftmost() const = 0 The function gives the column index of the left most ‘#’ of a piece in a particular orientation. The return values are directly obtained by the author of the code from the graph of each well-defined constant reference piece, so no calculation is needed for finding the position. Given its property, this function is made to be pure virtual since the all piece have different left most ‘#’ in their graph.
        11. virtual int rightmost() const = 0 The function gives the column index of the right most ‘#’ of a piece in a particular orientation. The return values are directly obtained by the author of the code from the graph of each well-defined constant reference piece, so no calculation is needed for finding the position. Given its property, this function is made to be pure virtual since the all piece have different right most ‘#’ in their graph.
        12. virtual void rotate() = 0 The function is needed to tell the piece that it has been rotated. This include update of the orientation and the graph change. The function is pure virtual since the rotation of different piece result in different graphs. Also, some piece (e.g. piece O) will not need to change its graph
        13. virtual void move\_right() The function is needed to tell the piece that it has been moved right. It is virtual since all piece but Crazy Piece moves to the right when it is called. Since Crazy Piece moves differently, a special version will be needed, the function is virtual to allow this implementation. The body of the function simply increment the x position by 1.
        14. virtual void move\_left() The function needed to tell the piece that it has been moved left. It is virtual since all piece but Crazy Piece moves to the left when it is called. Since Crazy Piece moves differently, a special version will be needed, the function is virtual to allow this implementation. The body of the function simply decrement the x position by 1.
     2. Class Piece’ s **protected member**
        1. void change\_graph\_to(const char ref[4][4]) The function needed by the derived classed to change their graphs. The function take a reference graph as a parameter and copy the reference to the std::string\* block private member. It is listed as protected to reduce the risk of change the graph of a piece from an external class. This function is not virtual since change the graph is generic for all derived pieces
        2. void update\_orientation() The function needed to updates the orientation of the piece when it is rotated. This function is protected to limit risk of changing orientation by other classes and functions. This function is not virtual since the orientation of all piece changes in the same way: from 0 to 1, to 2, to 3, and go back to 0.
     3. Class Piece’ s **private member**
        1. PieceType m\_piece Store what kinds of piece it is
        2. std::string\* block Store the graph of a piece
        3. int orientation Store the piece’s current orientation
        4. int m\_pos\_x Store the x position of the piece
        5. int m\_pos\_y Store the y position of the piece
        6. Well\* m\_well Points to the well the piece lived in
2. class PieceI, PieceL, PieceJ, PieceT, PieceS, PieceZ (derived classes)
   * 1. their **public member**
        1. Piece*Something*(Well\* well) The constructor of the these classes, *Something* can be I, L, J, T, S, or Z. Initialize the Piece class using PIECE\_*Something* (its type). the well parameter, and the constant reference graph with orientation 0.
        2. virtual void rotate() The function is needed to rotate these pieces, call update\_orientation() to change the orientation, and change the graph with according to its current orientation (so the graph change uses different reference graph according to the piece). In a derived class, they follow the base class’s virtual function to be virtual as well.
        3. virtual int leftmost() const Needed to return the leftmost ‘#’ of these pieces at different orientation (different implementation with similar idea for each of these piece). In a derived class, they follow the base class’s virtual function to be virtual as well.
        4. virtual int rightmost() const Needed to return the rightmost ‘#’ of PieceI at different orientations(different implementation with similar idea for each of these piece). In a derived class, they follow the base class’s virtual function to be virtual as well.
     2. their **private member**
        1. None, inherited from the base class.
3. class PieceO, PieceVapor, PieceFoam (a derived class)
   * 1. their **public member**
        1. Piece*Something* (Well\* well) The constructor of the these classes. *Something* can be O, Vapor, or Foam. Initialize the Piece class using PIECE\_*Something* (its type). the well parameter, and the constant reference graph with orientation 0.
        2. virtual void rotate() The function is needed to rotate them, call update\_orientation() to change the orientation, for these pieces, rotate does not change the graphs, since they have the same graphs for all orientations. In a derived class, they follow the base class’s virtual function to be virtual as well.
        3. virtual int leftmost() const Needed to return the leftmost ‘#’ of these pieces (for all orientation, but the their graphs does not change as orientation change). In a derived class, they follow the base class’s virtual function to be virtual as well.
        4. virtual int rightmost() const Needed to return the rightmost ‘#’ these pieces(for all orientation, but the their graphs does not change as orientation change). In a derived class, they follow the base class’s virtual function to be virtual as well.
     2. their **private member**
        1. None, inherited from the base class.
4. class PieceCrazy (a derived class)
   * 1. It’s **public member**
        1. PieceCrazy(Well\* well) The constructor of the PieceCrazy, initialize the Piece class using PIECE\_CRAZY (its type). the well parameter, and the constant reference graph with orientation 0.
        2. virtual void rotate() The function is needed to rotate PieceCrazy, call update\_orientation() to change the orientation, and change the graph with according to its current orientation. In a derived class, they follow the base class’s virtual function to be virtual as well.
        3. virtual int leftmost() const Needed to return the leftmost ‘#’ of PieceCrazy at different orientations. In a derived class, it follows the base class’s virtual function to be virtual as well.
        4. virtual int rightmost() const Needed to return the rightmost ‘#’ of PieceCrazy at different orientations. In a derived class, it follows the base class’s virtual function to be virtual as well.
        5. virtual void move\_right() The function need to implement the special action of Piece Crazy’s moving-to-right motion, which actually lead the piece to move left. So it will just call the Piece::move\_left() function.
        6. virtual void move\_left() The function need to implement the special action of Piece Crazy’s moving-to-left motion, which actually lead the piece to move right. So it will just call the Piece::move\_right() function.
     2. It’s **private member**
        1. None, inherited from the base class.

**File: Well.h**

1. class Well
   1. It’s **public member**
      1. Well(int well\_depth, int well\_width) Constructor of Well, takes the two parameter to determine the well’s dimension. Constructor is never virtual.
      2. ~Well() Destructor of Well, not virtual since there is no derived classes and it the only one form the destructor needs to be. In the body of the destructor, it deletes all the dynamically allocated objects including (see private member section for detail): Piece\* curPiece Layer\*\* m\_layer Layer\* m\_bottom
      3. void display(Screen& screen, int x, int y) The function that display the well. Taking the advantage of the private class Layer, the well’s display is to repeatedly display each Layer (call Layer::display()) and then display the bottom of the well. The screen parameter determines which screen is the well going to be displayed on and x & y determines the position of the upper left corner of the well (which is the boundary of the well). There is no need for this function to be virtual since there is no derived class.
      4. bool addPiece() The function write the curPiece (member) to the Well at the designated location (the upper left corner the piece’s graph is at x = 3, y = 0. Return true if the piece is added without overwriting already-occupied space in well, return false if adding the piece will overwrite the an occupied spot in the well. The judgment is made using the overloaded function bool attemptOverWrite(int x, int y, const std::string\* graph) const, which use std::string\* to be an reference graph. There is no need for this function to be virtual since there is no derived classes
      5. void updatePiece(Piece\* another) change the curPiece of the Well, update curPiece to the another parameter passed in (if another is not curPiece). It is used when the old curPiece is rested and a newly drawed piece is passed by the Game class to Well. There is no need for this function to be virtual, since there is no derived class.
      6. bool canGoDown() const The function that check if the curPiece, given its location, can go further down in the well. If it can without meeting barriers of the occupied spots, return true. If it’s expected position after moving down would overlap with an occupied spot, return false. It is used as a safety guard in function moveDown() to prevent it from doing an invalid move. There is no need for this function to be virtual as there is no derived class.
      7. bool canGoRight() const The function that check if the curPiece, given its location, can go further to the right. If it can without meeting barriers of the occupied spots, return true. If it’s expected position after moving down would overlap with an occupied spot, return false. It is used as a safety guard in function moveRight() to prevent it from doing an invalid move. There is no need for this function to be virtual as there is no derived class.
      8. bool canGoLeft() const The function that check if the curPiece, given its location in the well, can go further to the left. If it can without meeting barriers of the occupied spots, return true. If it’s expected position after moving down would overlap with an occupied spot, return false. It is used as a safety guard in function moveLeft() to prevent it from doing an invalid move. There is no need for this function to be virtual as there is no derived class.
      9. bool canRotate() const The function that test if the curPiece, given its current location, can go be rotated without overwriting the occupied spot in the well. If the rotation does not overwrite any existing spot, the function returns true and otherwise returns false. This function is used as a safe guard for the function rotate() to prevent it from doing invalid move. (note that this is Well::rotate(), not the same as Piece::rotate(), whose function is update information stored in the piece) This function is not virtual since there is no derived class who have potential to use another version of this function.
      10. void fix() The function that let the curPiece to become rest when it can no longer move down. The function identifies the type of the piece and do actions required for that piece. The reason to write fix function inside the Well class but not the Piece class is to limit the used of mutator that can access the arbitrary location inside the well and change the character in that position. For normal pieces, the function calls Layer’s mutate function to change all ‘#’ to ‘$’ in a designated layer. For a foam bomb, the private function FoamWrite is called to recursively generate the “foam” filled with ‘\*’. For the vapor bomb, it calls Layer’s clear function to clear the pieces (if any) in the desired region. After a piece is fixed, the curPiece is deleted. Again, the function is not virtual since there is no derived class of Well.
      11. bool moveDown() The function that is used to move down the curPiece. It first check if the piece can be moved down based on the judgement from canGoDown(). If the curPiece cannot be move down further, the function returns false. Otherwise, it clear curPiece’s current graph, calls the Piece::move\_down() function to update the position of curPiece, and write a new graph in the well based on the updated position. After these actions are done, the function returns true. Again, the function is not virtual since there is no derived class of Well.
      12. void moveRight() The function that is used to move the curPiece right. the logic of the function goes like this: If curPiece is Crazy Shape if(! canGoLeft) return; else if(! canGoRight) return; otherwise clearPiece; curPiece->move\_right; write updated piece; The function is not declared virtual since there is no derived class for Well.
      13. void moveLeft() The function that is used to move the curPiece left. the logic of the function goes like this: If curPiece is Crazy Shape if(! canGoRight) return; else if(! canGoLeft) return; otherwise clearPiece; curPiece->move\_left; write updated piece; The function is not declared virtual since there is no derived class for Well.
      14. void rotate() The function clear the piece first, check if a piece can be rotated. If it cannot be rotated, write it back. If it can, update the piece’s orientation and write the updated piece’s graph to the well at the original location. Again, the function is not virtual since there is no derived class of Well.
      15. int clean() The function that clean the clean Well’s empty layer, and rearrange the layers so that emptied layers are moved to the top and non-filled layer are moved down. Again, the function is not virtual since there is no derived class of Well.
      16. void discard() The function that empty everything in the Well and bring it back to newly created state. Again, the function is not virtual since there is no derived class of Well.
   2. It’s **private member** (brief information)
      1. bool attemptOverWrite(int x, int y, const char graph[4][4]) const Check if writing the graph provide in the parameter at location x and y will over write a occupied spot
      2. bool attemptOverWrite(int x, int y, const std::string\* graph) const An overloaded version of the function in (i) so that it can be used for type string\*. check if writing the graph provide in the parameter at location x and y will over write a occupied spot
      3. void FoamWrite(int cur\_x, int cur\_y, const int bomb\_x, const int bomb\_y) Recursively write the ‘\*’ character into the well when a foam bomb become rest
      4. void writePiece(int x, int y) The function that write the curPiece at location (x,y)
      5. void clearPiece() The function that clear the curPiece in the Well
      6. int m\_depth Store the depth of the Well
      7. int m\_width Store the width of the well
      8. Piece\* curPiece A Piece type pointer points to the piece currently being played
      9. struct Layer a private class specifically designed to allow easy implementation of each row of the Well. It has these members:
         1. Constructor: Layer(bool isBottom, int layer\_width)
         2. Function that write a part of the layer according to reference: void write(int start\_pos, const std::string reference, int length)
         3. Function that clear a part of the layer: void clear(int start\_pos, int length)
         4. Function that change characters from one to another for a layer: int mutate(char org, char after)
         5. Function that display a layer: std::string display() const
         6. Function that empty a filled layer: bool empty()
         7. Function that empty a layer even when it is not full: void destroy()
         8. Function that access an element in a layer: char getElement(int position) const
         9. Function that update how much a layer is filled: void updateFullness(int newFixed)
         10. Store how wide a layer should be: int m\_layer\_width
         11. Store the character to be used to represent boundary: char boundary
         12. Store the content of the layer: std::string m\_array
         13. Store if a layer is treated to be the bottom: bool m\_isBottom
         14. Store if a layer is full: bool m\_isFull
         15. Store how many spot is occupied in a layer: int m\_filled
      10. Layer\*\* m\_layer The 2D array that represent the Well
      11. Layer\* m\_bottom The bottom of the Well

**File: Game.h**

1. Class Game (***The class has no derived class, so none of the function is virtual)***
   1. It’s **public member**
      1. Game(int width, int height) Constructor of Game, using width and height parameter to construct the Well. Initially, the level is set to 1, score to be 0 and rows left to be 5
      2. ~Game() Destructor of the Game when game over, delete the pointer points to the next piece that has been drawn but not been used.
      3. void play() The function that allow the user to play the game responsible to check if the game has ended and display the related prompt.
      4. bool playOneLevel() The function that check if the user has complete one level of the game. If the game ends without completing one level, return false. If one level is completed, return true. Otherwise, update the ask the well to move down the piece when time is up or ask the well to respond to the user’s command.
      5. void displayPrompt(std::string s) display the string s as prompt at designated location.
      6. void displayStatus() Display the score, level, and rows left at designated location.
      7. void drawPiece() create the m\_next\_piece based on random chosen piece.
      8. int get\_score(int linesCleaned) The function that return the score players get using the lines they vaporized simultaneously
   2. It’s **private member** (brief overview)
      1. Piece\* m\_next\_piece the pointer points to the next piece to be played
      2. Well m\_well The Well players must manipulate
      3. Screen m\_screen The screen everything is displayed
      4. int m\_level The level player is currently playing
      5. int m\_score The score player has got
      6. int m\_row\_left The rows player must vaporize to complete current level