Biquadris

A5

m39khali, q6nie

Introduction:

We made this biquadris/tetris game which supports two players. This game has seven different block types: I, J, L, O, S, T, and * (in level 4). When a player fills a line on the board using the different blocks the line gets popped and the player that filled the line earns a point. When the program determines that the block cannot be placed (i.e., there is only three lines remaining from the top). After the game ends, the players can choose to end the game or restart.

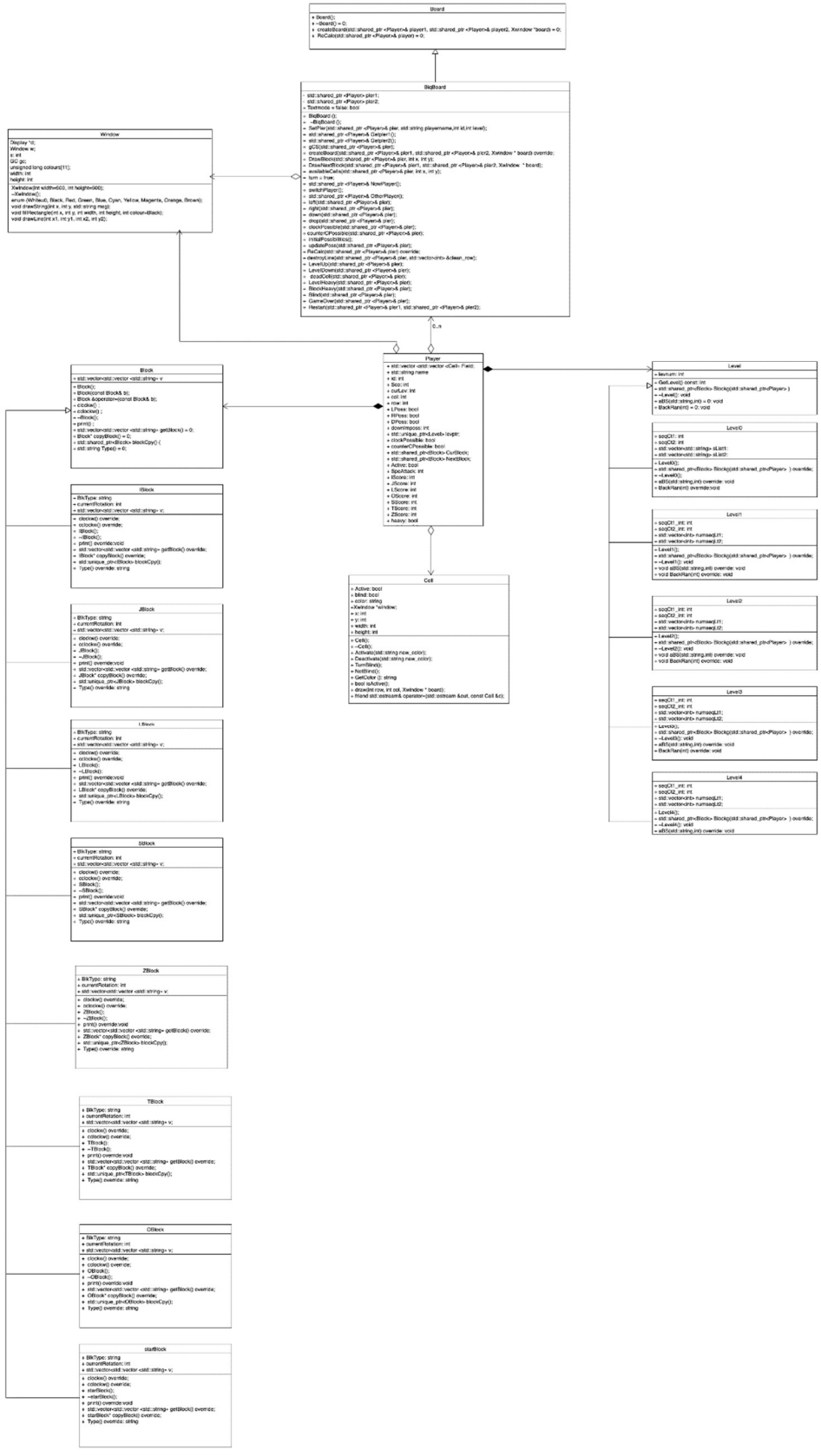
Project Overview:

This game we created utilizes 5 different classes:

- Board
- Cell
- Player
- Level
- Block

We use smart pointers to manage these classes. There will be more detail on these classes in the design section of this report.

Updated UML


Iboard
+ Iboard();
+ ~Iboard() = 0;
+ createBoard(std::shared_ptr <Player>& player1, std::shared_ptr <Player>& player2, Xwindow *board) = 0;
+ ReCalc(std::shared_ptr<Player>& player) = 0;
BigBoard
- std::shared_ptr <Player> plar1;
- std::shared_ptr <Player> plar2;
- Textmode = false: bool
+ BigBoard ();
+ ~BigBoard ();
+ SetPlar(std::shared_ptr <Player>& plar, std::string playername,int id,int level);
+ std::shared_ptr <Player>& Getplar1();
+ std::shared_ptr <Player>& Getplar2();
+ gC(std::shared_ptr <Player>& plar);
+ createBoard(std::shared_ptr <Player>& plar1, std::shared_ptr <Player>& plar2, Xwindow * board) override;
+ DrawBlock(std::shared_ptr <Player>& plar, int x, int y);
+ DrawNextBlock(std::shared_ptr <Player>& plar1, std::shared_ptr <Player>& plar2, Xwindow * board);
+ availableCells(std::shared_ptr <Player>& plar, int x, int y);
+ turn = true;
+ std::shared_ptr <Player>& NowPlayer();
+ switchPlayer();
- std::shared_ptr <Player>& OtherPlayer();
+ left(std::shared_ptr <Player>& plar);
+ right(std::shared_ptr <Player>& plar);
+ down(std::shared_ptr <Player>& plar);
+ drop(std::shared_ptr <Player>& plar);
+ clockPossible(std::shared_ptr <Player>& plar);
+ counterCPossible(std::shared_ptr <Player>& plar);
+ initialPossBlock();
+ updatePoss(std::shared_ptr <Player>& plar);
+ ReCalc(std::shared_ptr <Player>& plar) override;
+ destroyLine(std::shared_ptr <Player>& plar, std::vector<int> &clean_row);
+ LevelUp(std::shared_ptr <Player>& plar);
+ LevelDown(std::shared_ptr <Player>& plar);
+ dropCell(std::shared_ptr <Player>& plar);
+ LevelHeavy(std::shared_ptr <Player>& plar);
+ BlockHeavy(std::shared_ptr <Player>& plar);
+ filled(std::shared_ptr <Player>& plar);
+ GameOver(std::shared_ptr <Player>& plar);
+ Restart(std::shared_ptr <Player>& plar1, std::shared_ptr <Player>& plar2);
Window
Display *d;
Window w;
s: int
GC gc;
unsigned long colours[11];
width: int
height: int
Xwindow(int width=500, int height=500);
~Xwindow();
enum {White=0, Black, Red, Green, Blue, Cyan, Yellow, Magenta, Orange, Brown};
void drawString(int x, int y, std::string msg);
void fillRectangle(int x, int y, int width, int height, int colour=Black);
void drawLine(int x1, int y1, int x2, int y2);
0..n
Player
+ std::vector <std::vector <Cell> Field;
+ std::string name
+ xp: int
+ Bscr: int
+ curLev: int
+ col: int
+ row: int
+ LPoss: bool
+ RPoss: bool
+ DPoss: bool
+ downImpos: int
+ std::unique_ptr<Level> levptr;
+ clockPossible: bool
+ counterCPossible: bool
+ std::shared_ptr<Block> CurBlock;
+ std::shared_ptr<Block> NextBlock;
+ Active: bool
+ SpclState: int
+ IScore: int
+ JScore: int
+ LScore: int
+ OScore: int
+ SScore: int
+ TScore: int
+ ZScore: int
+ heavy: bool
Level
- levnum: int
+ GetLevel(): const: int
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>)
+ ~Level(): void
+ allS(std::string,int) = 0: void
+ BlackRan(int) = 0: void
Level0
+ seqCt1: int
+ seqCt2: int
+ std::vector<std::string> sLint1:
+ std::vector<std::string> sLint2:
+ Level0();
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>) override;
+ ~Level0();
+ allS(std::string,int) override: void
+ BlackRan(int) override:void
Level1
+ seqCt1_int: int
+ seqCt2_int: int
+ std::vector<int> numseqLI1:
+ std::vector<int> numseqLI2:
+ Level1();
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>) override;
+ ~Level1(): void
+ void allS(std::string,int) override: void
+ void BlackRan(int) override: void
Level2
+ seqCt1_int: int
+ seqCt2_int: int
+ std::vector<int> numseqLI1:
+ std::vector<int> numseqLI2:
+ Level2();
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>) override;
+ ~Level2(): void
+ void allS(std::string,int) override: void
+ void BlackRan(int) override: void
Level3
+ seqCt1_int: int
+ seqCt2_int: int
+ std::vector<int> numseqLI1:
+ std::vector<int> numseqLI2:
+ Level3();
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>) override;
+ ~Level3(): void
+ allS(std::string,int) override: void
+ BlackRan(int) override: void
Level4
+ seqCt1_int: int
+ seqCt2_int: int
+ std::vector<int> numseqLI1:
+ std::vector<int> numseqLI2:
+ Level4();
+ std::shared_ptr<Block> Block(std::shared_ptr<Player>) override;
+ ~Level4(): void
+ allS(std::string,int) override: void
Block
+ std::vector<std::vector <std::string> v
+ Block();
+ Block(const Block& b);
+ Block &operator=(const Block& b);
+ cclock() ;
+ cclock() ;
+ ~Block();
+ print() ;
+ std::vector<std::vector <std::string> getBlock() = 0;
+ Block* copyBlock() = 0;
+ std::shared_ptr<Block> blockCpy() {
+ std::string Type() = 0;
Cell
+ Active: bool
+ blind: bool
+ color: string
+Xwindow *window;
+ x: int
+ y: int
+ width: int
+ height: int
+ Cell();
+ ~Cell();
+ Activate(std::string new_color);
+ Deactivate(std::string new_color);
+ TurnBlind();
+ NotBlind();
+ GetColor (): string
+ bool isActive();
+ draw(int row, int col, Xwindow * board);
+ friend std::ostream& operator<<(std::ostream &out, const Cell &c);
IBlock
- BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ IBlock();
+ ~IBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ IBlock* copyBlock() override;
+ std::unique_ptr<IBlock> blockCpy();
+ Type() override: string
JBlock
- BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ JBlock();
+ ~JBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ JBlock* copyBlock() override;
+ std::unique_ptr<JBlock> blockCpy();
+ Type() override: string
LBlock
- BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ LBlock();
+ ~LBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ LBlock* copyBlock() override;
+ std::unique_ptr<LBlock> blockCpy();
+ Type() override: string
SBlock
+ BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ SBlock();
+ ~SBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ SBlock* copyBlock() override;
+ std::unique_ptr<SBlock> blockCpy();
+ Type() override: string
ZBlock
+ BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ ZBlock();
+ ~ZBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ ZBlock* copyBlock() override;
+ std::unique_ptr<ZBlock> blockCpy();
+ Type() override: string
TBlock
+ BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ TBlock();
+ ~TBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ TBlock* copyBlock() override;
+ std::unique_ptr<TBlock> blockCpy();
+ Type() override: string
OBlock
+ BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ OBlock();
+ ~OBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ OBlock* copyBlock() override;
+ std::unique_ptr<OBlock> blockCpy();
+ Type() override: string
starBlock
+ BlkType: string
+ currentRotation: int
+ std::vector<std::vector <std::string> v;
+ cclock() override;
+ cclock() override;
+ starBlock();
+ ~starBlock();
+ print() override:void
+ std::vector<std::vector <std::string> getBlock() override;
+ starBlock* copyBlock() override;
+ Type() override: string

Design

Board:

This is a pure virtual class with two functions ReCalc and createBoard so we can possibly have either multiple types of boards/views or add special effects for later using the decorator class, but we could not get to that due to bugs arising using up our time.

The main subclass for this was BiqBoard which consisted of these main functions:

- Fields (Private): Pler1 “owns-a relationship”
- Fields (Private): Pler2 “owns-a relationship”

Public functions:

- Textmode: which checks if the board will be in text mode.
- SetPler: sets player with default values
- Getpler1: gets shared_pointer to player one.
- Getpler2: gets shared_pointer to player two.
- gCS: gets the current score of the respective Player class
- createBoard: creates the board graphically and in text mode if Textmode is true. This makes it so the level and score are displayed at the top above each of the player’s game area. Then we add the space where the blocks will drop and then another dashed “-“line to separate that from next blocks.
- DrawBlock: essentially spawns the block at the top left of the player’s game Field and checks if a down is possible until it is not then it proceeds to turn off (not possible to occupy) the cells that the block was in last. (Uses fillRectangle to fill XWindow with the next block at the bottom.)
- DrawNextBlock: just displays the next block that will spawn for both players.
- availableCells: this is a helper used by the DrawBlock to if there are any availableCells to move down to.
- left, right, down, drop, clockPossible, and counterCPossible: left, right, down, and drop decrement or increment x/y position of the block if possible (using LPoss, RPoss, etc.). clockPossible and counterCPossible are activated when the Player initiates a cw or ccw request. This is essentially done by rotated a clone of the block and seeing if it would be inside of the Field, if not then cw or ccw will not happen.
- initialPossibilities: sets all possibilities to default values.
- updatePoss: updates all move (i.e., left, right, clockwise, and counterclockwise) to true if they are possible and false if not.
- ReCalc: calculates if lines are completed and need to be destroyed.
- destroyLine: gets called by ReCalc to destroy the lines that are completed and add to score.
- LevelUp: Levels up when levelup command is requested.
- LevelDown: Levels up when leveldown command is requested.
- deadCell: called after every move made by the player to update the display.

- LevelHeavy and BlockHeavy: are used to implement the heavy command in the assignment.
- GameOver: ends the game when it detects that the next spawn of the block would spawn in the wrong area (the three top lines after the Score).
- Restart: asks player to reset the board and restart.

Player:

Private Fields:

- Field: holds the 2D vector of Cells representing the vacant and occupied blocks on the board.
- Score: keeps track of total score.
- curLev: shows current level.
- row/col: used to determine block placement.
- LPoss, RPoss, DPoss, clockPossible, and counterCPossible: used to determine possibility of movement L = left, R = right, and D = down.
- CurBlock: current block pointer. <shared_pointer>
- NextBlock: current block pointer. <shared_pointer>
- The IScore, JScore, LScore, OScore, SScore, TScore, and ZScore are used to calculate the total scores and are updated when a line is destroyed.

Cell:

The Cell class is used in the Field variable for Player. A cell is used to determine where a Block is on the Field, if there is available space to move left, right, or down.

- Active: tells us if the Cell is Active or not (i.e., True or False respectively).
- blind: used for the blind feature where a portion of the Field is hidden.
- x, y: used to determine the position of the cell.
- width, height: used for Xwindow size.

Public Functions:

- Deactivate, Activate: to Deactivate or Activate the Cell.
- isActive: returns a bool True if Active and False otherwise.

Block:

Block is an abstract class. There are seven different blocks used as points of reflection on the Field. Upon rotation of a block, Block in this class goes through modification first before getting updated on the board.

- v: is a 2-D vector of strings (4 x 5) that represents the respective Block with spaces around it corresponding to empty Cells.

```
v = {
    {" ", " ", " ", " "},
    {" ", " ", " ", " "},
    {" ", " ", " ", " "},
    {"J", " ", " ", " "},
    {"J", "J", "J", " "},
};
```

- clockw: This virtual method is means to introduce a clockwise change for the 2-D Vector. It is used to keep track of current_Rotation of the Block and introduce the changes expected.

```
if (currentRotation == 0) {
    v = {
        {" ", " ", " ", " "},
        {" ", " ", " ", " "},
        {"J", "J", " ", " "},
        {"J", " ", " ", " "},
        {"J", " ", " ", " "},
    };
    currentRotation = 1;
```

- cclockw: This virtual method is means to introduce a counter-clockwise change for the 2-D vector. It a product of calling clockw three times. This does not apply to the OBlock since both the clockw and cclockw produces the same result.

```
} else if (currentRotation == 2)
    v = {
        {" ", " ", " ", " "},
        {" ", " ", " ", " "},
        {" ", "J", " ", " "},
        {" ", "J", " ", " "},
        {"J", "J", " ", " "},
    };
    currentRotation = 3;
```

- getBlock: It is the virtual method which returns the current Block as a 2-D vector (look above for example).
- blockCpy: The copyblck returns a deep copy clone of the block that is being copied. blockCpy then takes that clone and assigns it to a smart shared pointer.

Level:

Level is an abstract class that uses the Factory Method. This holds the sequence of the blocks and how they will spawn based on what level the player is on.

Public functions:

- GetLevel: virtual function that gets the current level based on what level it is.
- Blockg: virtual function that gets the vector of blocks based on the given sequence of the Player:
- aBS and BackRan: virtual function that is used to modify the randomness or give different sequence files instead of the default ones to run in the levels (specific to Level3 and Level4).

Main:

The Main file contains the user interface commands:

The different commands to run the program with:

- ./biquadris -t for text version
- ./biquadris -seed int to enter a seed number for srand
- ./biquadris -scriptfile1 “name of file”
- ./biquadris -scriptfile2 “name of file”
- ./biquadris -scriptfile1 “name of file” -scriptfile2 “name of file”
- The interface can handle multiple flags at once.

Resilience To Change

We have five classes to represent the game and we believe that this has low coupling and high cohesion. In terms of Blocks, block is an abstract class so more blocks can be created as needed for example other than the seven initial blocks, we created the starBlock “*” with ease and implemented it to the code with minimal changes as it was treated as its own Block and had separate .cc and .h files. The Level class is also an abstract class so many more levels past Level4 can be added even though we did not add more the five levels can act as an example. BiqBoard can handle more Players as all it would be is adding more Player classes in BiqBoard and then adding code similar to Player1 and Player2 so we can use that as reference to write code for Player3 and Player4. Thus, it is high cohesion and low coupling.

Answers to Questions

Question: How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

Answer: A static variable could be placed inside the Cell class that keeps track of the moves made then we would deactivate cells using the Deactivate function if the blocks have not been cleared before 10 or more blocks. The other possibility functions would be able to recognize that the cells disappeared therefore that space is vacant now.

Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

Answer: As explained in the resilience to change section the Levels are separated (e.g., Level0.cc, Level1.cc, etc.) so we could introduce Level5.cc/Level5.h which would inherit from Level.h and this would make it so the additional level is added with minimal recompilation as other levels would be untouched.

Question: How could you design your program to allow for multiple effects to applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

Answer: Currently in our program, we add all the effects through the BiqBoard, but it is entirely possible to use the decorator pattern. Because this project was a race against time for us, we added all our effects into the BiqBoard instead where we would add more effects if needed but we would not be able to avoid the else branch without using the Decorator Pattern.

Question: How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g., something like rename counterclockwise cc)? How might you support a "macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

Answer: Although we did not get to this in our program but if we were to implement it, we could do this by having a vector of strings that holds all of the commands that we use for internal purposes (vi) and similarly we could have a vector that the user (vu) can use which at the beginning is defaulted to a deep copy of the vector that is for internal use. And use the index to match vi and vu together if the user wants to switch the command, they can use the rename command to change vu keeping it the same index as vi. Adding extra commands would emplace the command at the end of vi and vu.

Extra Credit Features

Start Page:

We added the Start page which contains the controls and brief description of the program. It was made to give a nice introduction to our program and help a new user to learn the controls of the game without breaking it.

Final Questions

Question: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Answer: There are definitely pros and cons to developing software in a group. Pros would be, brainstorming for as many ideas as possible in a group resulted in 2-3x the results compared to brainstorming alone. Splitting up work makes it much more efficient than going at it alone and editing code could inspire more ideas that can be implemented for the betterment of the code. Cons would be, you don't always agree on some ideas and ways to tackle a few roadblocks that we came across when developing the game, but we learned to compromise and combine some of our ideas. We worked tirelessly over the last week for at least 5 hours a day and somedays it would be 7. Overall, it was a stressful but fun two weeks for us even though it was a two man project.

Question: What would you have done differently if you had the chance to start over?

Answer: we would definitely rethink on where to start as we wasted quite a bit of time attempting to make the board but, in the end, we realized that we couldn't then moved on to blocks so we would start with the smallest things like Blocks and Levels then move up to the Player, Board, then Interface. We would also learn more about XWindow so that we could've used more colours and made the rendering and gameplay a lot faster than it currently is right now. Other than that, make more friends and try to get a 3 man team for more efficiency.