CS246::Biquardis

Introduction

For our final project, we decided to implement Biquadris! To store our project and update between all of our members, we used a Git repository: GitHub Repository

Overview

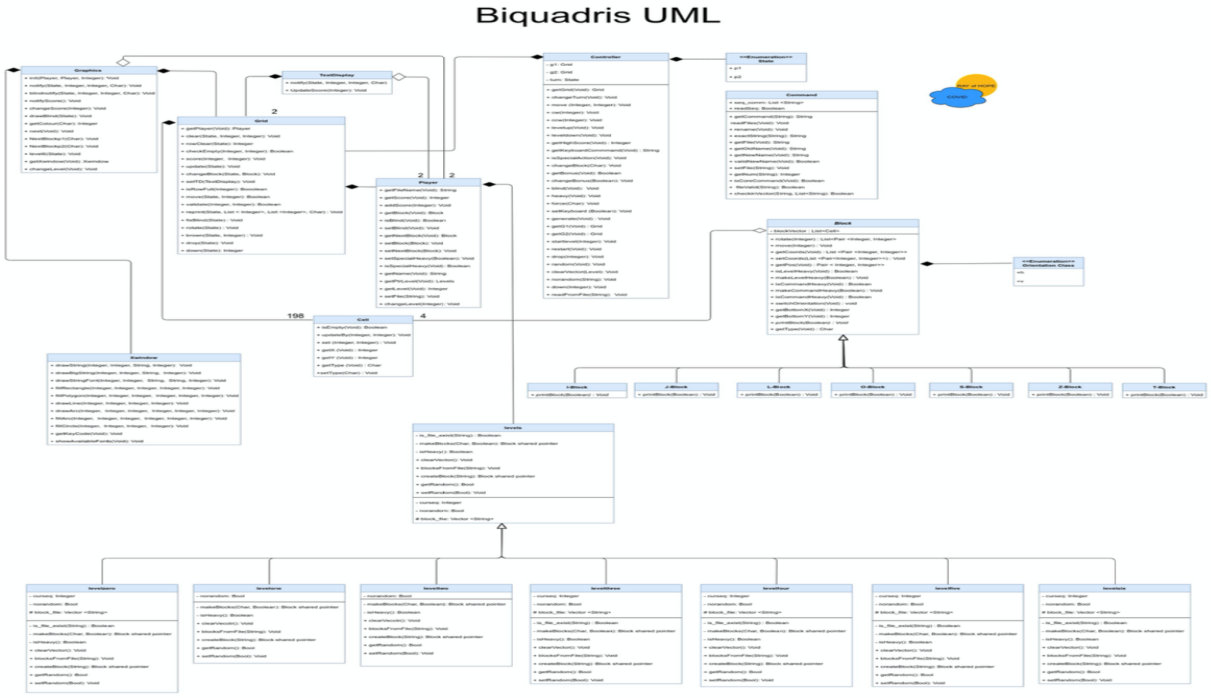
In our project, we decided to make use of a Controller class to support the entire game, and make for a much cleaner main file. Hence, any command from the command interface then calls the appropriate method from Controller to accurately play the game. The Controller class owns 2 Grids, the TextDisplay and Graphics. The Grid has a HAS-A relationship with the TextDisplay and Graphics, and the Grid each own one of the (via a OWNS-A relationship) Players. Using these relationships, we were able to toggle which player's turn it is between moves, and accurately call functions on the grid based on the commands inputted. As a Grid OWNS-A a player, in order to move, the Grid simply determined the availability and validity of the coordinates returned when a block returned its updated coordinates when moved in the desired direction. Each Player owns their current block and next block, and that information is shared between the TextDisplay and Graphics who also have a (HAS-A) relationship with both Players, to make Displaying the current block, the score, and player's level much simpler.
In order to accept commands easily from both "sequence" files and user input, we made a Command class in order to accurately understand which commands were being called. Furthermore, by having a Command class we were able to easily interpret commands with several different notations/forms for each command. For example, "left" can be called my "left", "lef", "lf", "lft", etc. The Command class essentially stored different versions of commands as a vector that was read in from various textfiles being supplied to the commands. This allowed us to additionally create a bonus feature to rename commands very easily, by simply emplacing a new element in the appropriate vector. Data being stored in textfiles was a strategy we also used for saving the high score of the program, which was updated if the high score was exceeded by updating the correct text file accordingly. Aside from interpreting commands, the Controller was able to call on methods for most other classes in order to have the desired effect. For instance, the command to move left 5 times would be a loop used to call the Grid's appropriate

method to move left either 5 times, or until the Grid has a return value that signifies it is unable to move left any more. In order to support the "restart" command, the Grid, and Graphics rather than having a defined constructor, have an initializer method in order to erase the Grid, and redraw the board. Whenever the Controller's drop method is called, the Controller's change turn method is called, and thus the automatically the next Player's turn begins and their block generates on the Grid and becomes visible.

Summary of Important Classes:

Class	Summary	HAS-A	OWNS-A
Controller	Controls the entire program, toggle turns, move blocks, generate blocks, restart and control special actions.	Player	TextDisplay TextDisplay Graphics
Command	This class is used to interpret commands, rename commands and check validity of actions.	N/A	N/A
Grid	Stores the entire Grid of the game, physically moves and clears blocks and determines when a special action is required from the Controller.	TextDisplay Graphics	Player Cell
Cell	Each cell contains coordinates to its position on a Grid, and a type.	N/A	N/A
TextDisplay	Used to output the display on the command line.	Player	N/A
Graphics	Visually display the Board using XQuartz and the Xwindow class.	Player	N/A
Levels	Used to generate new blocks for the players based on the levels specification.	N/A	Block
Blocks	Has 4 cells, and is able to return accurate coordinates for when they are moved/rotated. Contains the block's orientation as well.	Cell	N/A

Updated UML



Design

Changes since DD1:

In our Design, we had a full working Text version of the program prior to DD1, and thus we really didn't have much change after DD1 except for a majority of the Command interpretation and Graphics. When we first started the project, each if statement in the game loop came with several different comparisons (for example: if (command == "left' ll command == "lft")). This was very poor design, and instead as stated in the Overview Section, we changed this entirely to have a command class entirely in charge of understanding commands, checking for validity, and ensuring short forms are accepted and used by the program. Hence, while our design supports default typo's we acknowledge can occur, users can easily rename their commands, and our design ensures its validity as well. Another huge change to our Design was that in order to properly allow all the commands to be read from a sequence file, rather than the Controller class reading in input when a special action occurs, the Controller throws an

exception, and allows main to read in input, which is then passed back to the appropriate special action method within Controller.

Design Patterns:

1. Observer Design Pattern (TextDisplay::notify and Graphics::notify)
   Our TextDisplay and Graphics classes use a partial Observer Design Pattern. They each had a notify function that would be able to notify the TextDisplay and Graphics about when a particular cell changed it's type, or moved to accurately reflect those changes. When a block moved in any direction, that particular method would update TextDisplay with that block's type as a character, and if the program was not in "-text" mode, Graphics would also be updated with the same parameters, and would use a getColour method that would return the corresponding colour for that block's type. Our reasoning for doing this was to ensure a more efficient system to update the TextDisplay and Graphics, rather than constantly updating and outputting every single cell each time a move occurs. This way, the TextDisplay can work by updating a vector of characters, and the Graphics can instead simply re-draw them using the particular dimensions of a cell on the display. These notify methods can be found in textdisplay.cc and graphics.cc of our code. In our bonus feature level 6, we include a special notify function that is used to re-update the graphics with every single cell, and can only be used if a Player is on level 6 and has successfully finished their turn.
2. Factory Method Design Pattern (Player::createblock)
   Every level designed for the game uses the Factory Method Design Pattern to provide the user with a block. In levels 1-4, the block created is based on specific probabilities, as per the assignment specifications, and level 5 and 6 uses the same probabilities as level 4. This was a very important design to the game, as it allows for variability within the block's generated, making the game much more entertaining for the player, rather than having a predefined list of block types to return or perhaps to just repeat a loop of particular blocks. Every level comes with a createBlock and makeBlock method. The createBlock method is in charge of determining what the *type* of the block should be, based on whether or not the Player has a particular file of blocks to read from. If the Player does not, the createBlock method will use the rand C++ function, as well as a seed to randomly

generate a number, each of which is associated with a particular block type. Hence, the createBlock method would manage to create a character. For example, if the character was I, it would suggest that an IBlock should be made. This character is then passed to makeBlock, which will return a block of that type. In the Controller::generate method, the block returned is set as either the player's current or next block, which can then also be accessed by the Grid in order to move and drop.

3. Mediator Design Pattern (Controller)
   When we first started making Biquadris, we were really confused as to how we can control how all of our methods worked, or how they should be put together. Our main function was already very long, and we didn't want to have several helper functions in main either. Instead, we decided to use the Mediator Design Pattern and make a Controller. The controller indirectly has access to every single aspect of the game. It own's the Grid's, TextDisplay and Graphics. Hence, through public methods by the Grid, it also has access to the player's and can access the next and current block of each player as well. We chose to use this design, because it made the execution of the game loop very simple, and led to a very clear design as to how every function should work. If the user inputs left, the Controller immediately knows which Grid to use based on whose turn it is, what function to call from Grid (move with direction specified as left), and can accurately look for particular return commands from Grid's methods to determine if the move happened, and if so, output the Grid, using an overloaded output operator that will call to TextDisplay's overloaded output operator. Hence, this design pattern made the main file more abstract from the user, and much cleaner.

4. RAII and Exception Safety
   We decided to use smart pointers at all instances of allocating heap memory in our implementation, this kept us from explicitly allocating memory and freeing them when they would go out of scope. As the smart pointers would call their destructors and free the allocated memory automatically, and make memory management much easier. We decided to mainly use shared pointers for most cases like blocks, levels, and players as they are copied at many instances but our most important class Controller is a Unique pointer as it is one of its kind at each instance of the program.

5. Inheritance (Block & Levels)
   Our block and levels classes both used inheritance to be able to use block and level pointers, despite both of which being an abstract class. Hence, this allowed the support of many types of blocks and levels, despite only having the code the containment of these particular classes once.

Resilience to change

1. Cohesion and Coupling
   For our group, this was a hard aspect of the project. Originally, we didn't sit down and plan what should connect with what. Our first design had it such that each Grid class had 2 players, and a vector of cells for each player. This was very poorly designed and we quickly changed that due to the significant amount of code that was very cohesive, but repeated based on which Player's turn it is when the methods are called. We instead decided to have a player per Grid, which made the code much cleaner. Additionally, rather than having TextDisplay have a Grid, we found a way to simplify this to Player, so the correct score, name and level can be printed. The notify method was added to ensure the TextDisplay knows the type of every cell for fast outputting. Each Grid has a TextDisplay and Graphics as well, so Grid can call notify on each respective class. When the rest of the code with levels and blocks had to be incorporated together, we soon realized the benefits of having a Controller, and implemented that class as well. However, there ended up being far too many pointers to the Players, which we considered poor design. Hence in order to prevent this, we ensured that there were similar goals between every controller method and a corresponding method in an appropriate class, to ensure that our code was cohesive and to the point. Hence, rather than making one significantly large move command within our Controller, we simplified this based on horizontal and vertical moves, making for a cleaner design. In order to minimize coupling, we removed unnecessary pointers to player's in blocks and levels, and did it such that we made the code more efficient. By having a Controller that had access to most Classes, we didn't have to worry about excessive coupling and storing blocks in the Grid and Controller. Instead, we could pass around the Player in order to ensure everyone was accessing the same information that was managed by the Controller.

See the table in Overview for a complete breakdown of the relationships between the classes.

2. Various changes to the program specification
   We believe we did an excellent job at this, given our extra features already. We were able to add a new way to play the game, additional levels, renaming commands already. We do know there are more possibilities of changes that can occur based on the functionality of our program already.

a. Renaming Commands (Implemented)
   In our command class, we store all the acceptable forms for a command within vectors. Hence, in order to add a command all that needs to be done is replacing the element to the new desired name, or emplacing it to the back of the vector if both forms are desired. Hence, all that needs to be done is check to make sure the command doesn't exist in another vector already.

b. New Levels (Implemented)
   As the levels class was abstract, it was very simple to create a new level, by simply indicating its inheritance and overriding the pure virtual methods. Furthermore, safety nets such as ensuring level-up isn't called when the player is at a maximum level must change based on whether the program is being run with enhancements or not. For instance, if the program is running with additional features

c. Using Arrow Keys as Input (Implemented)
   Most tetris games work by using the arrow keys. It is quite hard to play by typing in a direction or command, rather than using the keys to move themselves. Hence, by using the Xwindow class to return computer keys and accurately determine them via a Controller method, it is possible to obtain input generated by playing with keys and return a string of the appropriate command via the command class. Hence, this is a very easy command to implement and makes for a very easy user experience.

d. New Blocks
   Due to the nature of the abstract block class, it allows other Blocks to be incorporated into the game very easily. Simply just providing the coordinates at which the Cell's must spawn on the Grid will allow for the Block class to perform the required calculations for its movement and rotations. Hence, all that is required is

the particular shape of the Block, and an addition of the Block to the possible options in the levels create block using Factory Method Design Pattern.

e. Changing Colour/Size of Graphics
Colours can be assigned via the Graphics::getColour method, and hence can easily be reassigned for aesthetic purposes. Furthermore, the size of the Grid's can also be updated as the height and width are two modifiable variables throughout Graphics to allow the user to control how they want their program to be viewed. Simpy update the size of the Grid's rectangle row and height accordingly and update the height and width and the entire view of the Graphics can be altered.

f. Single Player vs. Two Player
As change turn only occurs within a few sections of the code, it would be very simple to change the functionality to be single player, and redirect the special actions to be benefits to the player instead, such as an added score bonus and removing the current tactic of the effect they have. While this would be slightly more difficult for the Graphics and TextDisplay based on controlling the view of the files, executing it within the game loop is quite simple based on the code's nature of toggling between the players.

Answers to Questions from Project (including changes from DD1)

1. **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?**

To implement such a feature we could simply have a field inside the block class which keeps track of the amount of blocks that have dropped after this one. This field would be initialized as -1 for classes where this feature is off and would be initialized as 10 in the levels we want this feature to be on. This can be done by passing -1 or 10 when the block is created by the correct level. Then we can simply have a vector of pointers to keep track of these blocks to not lose them and a method to check if the "counter" has reached 0 which would be called each time before the turn is changed which would not be hard for us because of our change turn method. Another possibility, would be to manually go through all cells to see

those that have exceeded 10 or more turns. We used a similar aspect with dropping blocks in levels 4 and higher and simply increment a counter in each grid and increased it every time a block was dropped.

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

   Our current implementation is perfect for this kind of feature. We currently have a different subclass/child of our levels abstract class for each different level in different files. So, for an additional level we can simply create a new file with that level class which wouldn't require the recompilation of any of the other levels or blocks or anything unnecessary. Simply the new level class and its implementation in the main and our controller. Creating levels 5 and 6 took very little time for us, due to this abstract design pattern, making it very efficient and easy!

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

   We thought about implementing this as a bonus but due to time constraints never got around to it. Our idea was to have each effect defined as an enum class with a map holding these classes with a bool as a pair. Then if we needed to implement multiple effects we could simply go through the map and apply the effect for every effect that's true. Then when a change turn occurs our change turn method would simply “turn them off” in a sense by changing the bool to be false. Alternatively, we could implement the Decorator design pattern. We could implement our Grid with the decorator design pattern allowing different effects to be “wrapped” around the board. Either of these implementations would allow us to avoid having a different else-if for each effect and possible combination.

4. 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.

Well if we are introducing new commands to simply replace/rename old commands then it would be as simple as changing the contents of the corresponding text file in our repository. We have implemented a rename, but we deliberately implemented it in a manner such that they are unable to rename core commands such as Left, Right, Drop, etc, as a safety guard. In terms of adding a whole new feature with a new command, it would be as simple as creating a file for its possible shortcuts and a vector for those shortcuts to be read in. From there it would be as simple as make a method in our controller the additional feature and implement it wherever we need to before calling for it in the main when the command is called. We implemented this as a bonus feature and found it very simple due to how we were saving these commands for easy access.

Extra Credit Features

1. Level 5
   As suggested above, making new levels was a very simple task. Hence, we created two new levels, level 5 and level 6. These can be activated by providing the command line flag "-bonus" when running the executable, or by providing the command "bonus" to the game loop during run time, which will toggle the bonus settings, and allow the user to level up to level 6. If "bonus" is provided while the player's levels are above 4, the Player's are level'd down to level 4. Level 5 works very similarly to level 4, however rather than placing a brown block in the centre column after 5 turns with no row cleared, level 5 places one on every column. This can significantly increase the player's advantage if they play evenly, but can have negative effects if they do not.
2. Level 6
   Level 6 was designed to be extremely difficult. While each player can see their Grid when it is their opponent's turn, they are unable to see their Grid on their turn, and there the regular output statements on the command line from TextDisplay are also

paused until the player drops their block. Neither player is able to see their first block before they drop it either, increasing the difficulty of the game as well. The point of this level is to have the Player try to determine (while their opponent plays), what their next block is and where they want to place it, or potentially change the next block to be.

3. Keyboard Commands/Input (Mac & Linux tested only)
   The point of this feature was to allow the user to simply play Biquadris the game. So all the debugging/extra features are turned off during this mode, see demosheet for more details. To play in keyboard mode a flag is to be passed ("-keyboardmode") as a command line argument when launching biquadris. It allows the basic commands like left, right, down, drop, rotate, etc. To implement this feature we had to do research on XLib's programming manual to find the correct methods and features of Xlib needed to make this happen. Since this command was made possible by Xwindow, in order to use this your mouse's focus must be on the XQuartz window that opens and not in the command line. Finally, as this uses the Xwindow class, it is not possible to play using keyboard mode when the -text flag is passed to the program.
4. Rename
   This function renames commands that exist for shortcuts. As a safety guard, commands that are core commands, example "left", "right" will still be functional, and will allow for the addition of a short form as well. Trying to rename non core commands, example "lef", will allow you to replace the command. This was made by creating a vector of possible commands and emplacing back new elements.

Final Questions

1. 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?
   When we first started our program, we decided to work individually a lot. Our first time we all tried to compile our level zero code, we spent over 3 hours debugging. Thankfully, we had started this process early and had 2 weeks left to finish. For instance, the class for a level was called "levels", while in the block class we had assumed it was "level" and in the Grid class, it was called "Level". After that, we

called daily for at least an hour, to put together our separate methods in the Controller. First and foremost, one of the most important aspects about developing software in teams is **communication**. It's extremely vital to make sure that the team members are able to vocalize their opinions, talk about their progress, obstacles and also be able to ask for help whenever necessary. It's essential to divide the work between the members depending on their preferences and expertise and set up a tentative schedule for the members to deliver. This'll ensure maximum productivity (including for members depending on other member's code) and quickest delivery of the software. Technologies such as Zoom, Slack etc can be quite helpful for communication. A version control system such (Git) could be extremely helpful for collaboration on software products. It's important to not underestimate the time that integration of code could take. Integration almost always leads to errors and requires small or large changes depending on your work. Lastly, always test the code after making a change to ensure that the change has not broken the software, and make sure you all work on separate Git branches!

2. What would you have done differently if you had the chance to start over?

   When we started this assignment, our first thought was to get a levelzero working, and we wrote lots of code in order to accomplish this. However, we began to realize problems in our design with very high coupling. While we did fix this, fixing written code leads to bugs, and a large amount of time trying to determine what wasn't working. Also trying to link each part of the code was difficult. For example, how the Grid's rotate can use the Block's rotate, and that has to connect to the Controller's rotate, and what happens if it's heavy, or blind, or both heavy's. Had we planned, we could have each coded part by part, and have realized what we needed from each other, before it was too late to make those changes. While we soon had everything working, we would have saved a lot of time. Also, carefully read assignment specifications. Our team found out the day before DD2 that you must be able to turn on/off enhancements, which was stressful as there were over 5000 lines of code to carefully check over and make sure this was okay! We did it, but would definitely be more careful next time. *Overall: plan ahead and compile every edit!*

   Note: Page limited exceeded as per Piazza post question @1083.