University of Waterloo

CS246 Final Project

Biquadris

Braydon Wang, Dylan Wang

CS246

Object-Oriented Software Development

December 6, 2022

Table of Contents

Introduction

The project that our group made for the CS246 final project is Biquadris, a 2-player game variation of Tetris. From an overall view, Biquadris involves moving various tetrominoes to different parts of the board to try and clear rows, racking up points by doing so. There are different features and actions that the game permits, but in the end, the player with the highest score wins the game. We utilized many OOP techniques and patterns that we learned in the course to make this possible. The rest of this report will outline the overall structure and design of our project, and how we accomplished to make Biquadris.

Overview

The game first starts by reading flags passed in by the command-line interface through the arguments given with the executable. Each flag sets a default variable to the value specified. These values are then passed into the Game component, which is the main component that interacts with the players.

Game

The Game class is the controller of the program. It initializes all components that will be used later on in the game, such as the players and observers. The Game's responsibility is to continuously read input from the user until EOF is passed in. With each input string read, it is designed to interpret the command, while also following the rules with multiplier prefixes and passing in only as much of the command name as is necessary.

We accomplished this by checking if the input string matched the prefix substring of the command's full name that distinguishes it from other commands. On top of this, the input string must also be a proper prefix, in the way that no additional/new letters were added to make it differ from the command's full name. We also used an input string stream to help divide the string into two parts to get the numerical multiplier. After checking the input, the Game lets the Player components deal with each command's respective purpose. The Player component called depends on the turn number, which alternates each time a block is dropped. Though thing's get a little tricky when blocks are heavy. Heavy blocks force blocks to drop if no room is left for the block to move down. This is why each player has a variable that indicates whether a command has resulted in the block being dropped, so the turn can be updated even if the drop command is not called. After every command, the Game needs to check if any player has activated the special action. This works the same as the command interpreter; the program will interpret the input from the user and let the respective Player component deal with it. If the player chooses to quit the game, the program exits the while loop and the game stops.

Player

The Player class is the mastermind behind the game. It's where most of the game's functionality happens. It is also the only subject that the text and graphics observer view. The Player needs to keep track of the level, score, high score, current block, next block, board, and special actions. The class supports many functions that modify and alter the current block, such as moving and rotating it. While the algorithm for it remains in the Block class, the Player is still responsible for calling the Block's method a certain number of times, depending on the multiplier. Once the block is updated, the player notifies all of its observers that a change has been made. The block will also need to be moved down an additional row or two if the block is heavy or the heavy special action is called.

The Player also has the feature of dropping a block, which is equivalent to moving the block down until it can't go any further. Every block dropped is added to our map data structure that maps block ids to the number of individual block parts still on the board. We also check if each row that the dropped block occupies is filled or not. If it is, we delete it by removing the filled vector row from the board, and decrease the number of block parts from the map. If that number reaches 0, the score is updated to account for the block that was fully removed. The score is also updated depending on the number of rows filled. If the number of rows filled is greater than 1, the player's special action variable is activated.

Other features include leveling up and down, which is simply increasing or decreasing the level number by the multiplier and creating a new level object with the respective number. Using the level component, the Player also generates blocks. If a type character is given, the method creates a block of the given type. Otherwise, it calls the level's generate function which returns a character, and then uses that character to create the new block. This method is called when the current block is dropped; the next block becomes the current block, and the new next block is generated by calling the function.

Player's can also reset themselves, which resets all of their fields to the default value. Finally, everytime the current block or next block is updated/changed, the Player class has a method that unsets the old block's coordinates on the board, and then populates the board with the block's new coordinates.

Board

The Board is a 2D-vector of cells that represents the state of the game board. It can easily read and write to each individual cell. The Board class has methods that return a reference to a cell based on the given x and y coordinates. This is so that classes that have a Board (Player, Block) can easily modify certain cells on the board. Board also provides features that make it easy for the board to shift all blocks down when a row is cleared, such as deleting a certain row, adding a row to the top of the board, and setting a row to a new vector.

The Board class also contains a next block grid, which is different from the actual game grid. This is to make it easy to display the next block through text and graphics, as the blocks are displayed in the same way they are on the game grid. The same methods to get and set a cell on the game grid also exists for the next block grid.

Block

The Block is the only component on the game board. Since there are multiple different block types, there needs to be an abstract block class that has fields and methods that all types of blocks need. In addition to the 7 main types of blocks, we also have a star block, specific for level 4, and a brick block, for our bonus feature. Blocks are classified by their id, type, level generation, heaviness, and coordinates on the board. All blocks spawn at the upper left corner of the board below the reserve rows. Since we know their starting position and default shape, we can assign each block type a vector of predetermined coordinates.

Blocks can move left, right and down. To simplify these 3 movements, we made one generic move function that takes x and y parameters that indicate how the x and y positions need to move. For example, a move to the left would be indicated as move(-1,0) and a move down would be move(0,1). This also lets the block move more than one tile from their original position, which helps for the heavy special action that moves the block down twice. The move function also returns a boolean value, to show whether the move is valid or not. In other words, if the block would move out of bounds or onto another block, then the move would be considered invalid. The hard part in the Block class is to rotate it. Originally, we thought that each block needed their own rotate method because they are all of a different shape. However, we came up with an algorithm that rotates any kind of block.

To rotate the block, we first translated the block so that its lower left corner is at the origin of the plane. Then, we used the well-known formula of rotating coordinates about the origin to rotate the translated block. Finally, we need to translate it back to where it was by matching the lower left corner of the rotated block to the lower left corner of the original block.

Level

Due to the fact that different levels generate blocks differently, we created a Level class that takes away the responsibility of generating blocks from the other components. We have an abstract level class with a virtual function to generate blocks. Each specific level should override this function with their own implementation that aligns with how blocks are generated on that level whether that's generating blocks based on probability or a set input.

To generate blocks based on a given input, we read each character from the given file and store it in a vector. Then characters are taken from the vector in order, looping to the start if necessary, to produce the blocks.

To generate blocks based on probability, the rand function is used, which generates a pseudorandom integer from 0 to RAND_MAX (a large number). Then modulo is used to create equal partitions of the large interval, in which we can use if/if-else/else clauses to divide the partitions corresponding to the desired probability.

Observers

To display both text and graphics of the board, we utilized the observer design pattern. A more detailed explanation of the design we implemented will be discussed under the design section. We created a TextObserver and a GraphicsObserver class, which both inherit from the abstract Observer class. When these observers are created, they are immediately attached to the subject it is observing, which in this case are the two players. Both observers also have two Player components to easily get the board, score and level. In the TextObserver's notify function, it displays the level number and score of the player. Then, it loops through all rows and columns of the player's board and displays the character of the cell at that position. If the player is blinded, it does not display the characters at those specific rows and columns. The same is done for the next block using the next block grid. The GraphicsObserver follows a similar process with its notify method by using the Xwindow class.

The main difference is that when it comes to graphics, we need to only display what is necessary. In that case, we kept track of the state of the previous board that was displayed. Every time the board has changed, the previous board would update to the current board, and the current board to the new board.

This way, when displaying the board, only those cells that have changed when comparing the previous and current boards are redrawn. We also have a private method that converts from characters to colors. When we display strings, the tricky part is that we need to first render a blank rectangle over the old string before we can display the new string.

Updated UML

The updated UML that depicts the structure of our project is attached to the end of this document. The main change we made from our old UML is deciding which class to be our "brain" of the program. Originally, we thought that the Board class should offer most of the functionalities of the game, such as modifying the block and updating the level. However, we realized that this would be an issue when it comes to notifying the observers about the change, as the Player class must be the subject, as it contains information about the score, but it doesn't know when a block has been moved or rotated because that is located inside of the Board class. Instead, we made the Player class the main component of our game, while still being the subject of the observers.

That was the only design change that occurred; the only other differences were adding or removing certain methods in a class because they either weren't necessary or could be moved to a different class.

Design

Generating Blocks (Factory Design Pattern)

One of the biggest challenges of this project was the fact that each level generates blocks differently. To create the different levels, we implemented the factory design pattern. We had an abstract level class which contained all essential, common variables and also a virtual generate method. Each specific level is a subclass to the abstract level class and should override the generate function with their own implementation that aligns with how blocks are generated on that level. This made it so that in the Player class, we only needed to have one pointer to an abstract level object. Whenever we needed to generate a block, we would simply call the generate function on that level pointer, which would call its own respective implementation of the generate function. We didn't need to know anything about what level it is to generate the desired block. If the level changed as the game progresses, we would change what the level pointer is pointing to, and the rules for generating a block would automatically change with it.

Displaying both Text and Graphics (Observer Design Pattern)

Another challenge we faced was to accurately display the current state of the game in both text and graphics. To do this, we used the observer pattern. The Player class inherits from the abstract Subject class, which can add, remove and notify observers. The Subject class has a vector of observers that are observing it. We created both a TextObserver and a GraphicsObserver class that both inherit from the abstract Observer class. The abstract class has a virtual notify method that is overridden by both concrete observer classes. The TextObserver's notify method displays the state of the board to the output stream, while the GraphicsObserver uses the Xwindow class. Each observer also has a pointer to the subject, which in this case is both players in the game. Every time the board is updated, the Player class calls notifyObservers, in order for each individual observer to call their own notify method to display the board.

Types of Blocks (Polymorphism)

While there are many types of blocks in the game, they all work similarly to one another. This is why we decided to use polymorphism to represent the structure of blocks in the game. We had one abstract block class that contained information that all blocks have, including an id, type, and coordinates. Each type of block would inherit from that abstract class, but populate some fields with their own information. This included the type character and the vector of coordinates. This is because we know the default shape and position of each block's starting position. Other than that, all blocks act the same way. They can be moved, rotated and dropped by simply changing all of its coordinates. Essentially, each block subclass worries about its own construction, but leaves the rest of the details to the abstract block class to implement. This makes the code clean and well abstracted as the addition of new blocks can be made by simply adding a new subclass and filling in the details of its construction.

Exception Safety (RAII, Vectors, Maps)

To eliminate the possibility of memory leaks from exception throwing, we decided to follow the RAII idiom by wrapping all resources in a stack-allocated object. We converted raw pointers to unique or shared pointers depending on their ownership. For example, in the GraphicsObserver, the Xwindow object should be a unique pointer because only the graphics owns it, so it must be responsible for deleting it. In the Player class, both the current block and next block are shared pointers because they can be interchanged and copied, so they have a true shared ownership. In the Board class, we used a 2D-vector of cells to represent the grid. Every block has access to the board, so once their old coordinates change, they can update the grid with their new coordinates. We also used a map data structure to map from block ids to the number of block parts still on the board. The use of these data structures and smart pointers helped us accomplish the features of the game, while also maintaining a leak-free program.

Model-View-Controller Architecture

This project outlined many moving parts and features to implement. We wanted to simplify the whole process by dividing the main problem into multiple subproblems. In essence, we wanted to follow the single responsibility principle, so that every class only has one responsibility. This allows code to be reused and for classes to only focus on a single problem. We did this by adhering to the MVC architecture. The Game class is the Controller that takes input from the user and interprets all of the commands. The Game then calls the Player class, which is the Model that represents the state of the game. The Player class simply calls its own methods to update the data, but doesn't know any details about how the data is used or displayed. Finally, both observer classes are the View that manages the interface to present the player's data.

Resilience to Change

Low Coupling and High Cohesion

To design our program to be reusable and accommodate change with little modifications and recompilation, our design structure has low coupling and high cohesion. Coupling relates to how dependent distinct modules are to one another. Our classes have very low dependency because they simply call each other's functions using basic parameters and results. Player class calls Block class' moving and rotating function, Block calls Board's function to populate the grid, and Player calls the observer's notify function to display the state of the game. The implementation of the Level classes only depend on components within the Level module, but nothing beyond that. Data is not passed back and forth, and modules have zero access to the implementation of other modules. This makes it so that adding a new module requires no change to occur in other modules since all classes are distinct and independent from one another.

Cohesion relates to the closeness and similarity of elements in a module. In our project, every class has one responsibility and focuses purely on that feature. The Block class focuses only on moving and rotating the block. The Player class only dishes out the commands sent in by the user, including dropping a block or updating the level. The Game class is a controller strictly responsible for dealing with in-game commands as the game is running. Note that the game controller does not worry about getting the command-line flags, that is the responsibility of another module. All elements of a class work together to finish one task, and without one element, the whole feature will not work. Since everything is so cohesive together, the addition of a new feature will only require creating a new class whose sole purpose is to perform that feature. There will be little to no changes in other classes because they are unrelated to the new feature.

Abstract Classes and Inheritance

The entire premise of our Block and Level class is built on abstraction and inheritance. Both have an abstract class that cannot be initialized, but it holds the information that all Block or Level subclasses have in common. Every type of block and level is there own subclass that inherits from these abstract classes. They provide their own implementation of whatever differs between the types of Blocks/Levels. This makes it easy to add new blocks or new levels because it just involves creating a new subclass that inherits from the abstract class. Then, for any virtual function that needs to be overridden, the new subclass would provide its own implementation. Nothing in the main program or other classes need to change because it does not matter if a new Block or Level has been added, the changes to it will automatically be made if it points to the new subclass.

The same can be said with observers. We have an abstract observer class with two subclasses for the text and graphics display. If we ever wanted to add a new text or graphics observer, or even a new observer entirely, we can use the existing subclass or create a new subclass that overrides the virtual notify method.

Reusable Methods and Separate Classes

All of our classes are separate from each other. All of our features are split into methods that accomplish a single task. This makes it so that adding a new feature or command will involve changes to a small number of classes and the addition of no new code. We found that making our bonus features required absolutely no new code needed to be written. Since we split up the bigger problems into smaller methods, we found ourselves using those previous methods to make our new feature. There was also little recompilation needed because we only changed one to two classes. For example, when we added a new command to our game, we only had to add a new condition to the Game class when interpreting input from the user. Then, we created a single function in the Player class for the

Game to use when that command is called. That function relied entirely on previously made methods such as updating the current block, generating a new block, and populating the grid. We didn't write any new code and only recompiled two files to add this new change. This shows that all of our classes are separate in nature and provide implementations to methods that can be reused for different features.

Answers to Questions

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?

The way we can implement this within our system would be by introducing new fields to the Block class. Every Block should now have a private variable that keeps track of the number of turns that the block has existed for and a boolean variable indicating whether the block is a disappearing block or not. Each block should also have a function that deletes themselves from the board, which is essentially unsetting all of their coordinates from the grid. It should also move all blocks on top of them downwards to fill the gap. What we failed to mention last time is that we need to have a data structure that stores all blocks on the board. This way, we can easily update every block's turn counter, and once it reaches the limit of 10, it can call its own delete self function to disappear.

This could be easily restricted to more advanced levels by overriding the generate function. The abstract level class has a virtual generate function and each specific level overrides this function with respect to the criteria of that level. Higher levels will generate these disappearing blocks with greater probability, while lower levels will generate these disappearing blocks with small or zero probability.

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

The way we can design our program to allow for the introduction of new levels with minimum changes and recompilation is by invoking the Factory Method Design Pattern. We can create an abstract level class with basic features that all levels need, such as a virtual method to generate blocks. Then, each individual level will be a subclass of that abstract class and will override the virtual methods with their own feature. That way, each player has their own concrete subclass of the current level, and will use that to generate the next block.

To introduce new levels into the system, it simply involves creating a new subclass of the abstract level class that overrides all the necessary methods. In our case, the new level subclass will need to override the method that generates a block. This performs minimum recompilation because it only involves recompiling the new file created for the new level. Nothing in the Player class or the Board class needs to change because setting the level pointer to point to the new level is enough for the change to be automatically used.

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

To design our program to allow for multiple effects to be applied simultaneously, we can employ the Decorator Design Pattern. The Decorator Design Pattern is perfect because it lets you add multiple features to an object at run-time, which is exactly how the effects are applied. Effects may also be withdrawn, which is another feature Decorators are great at accomplishing.

We will have an abstract effects class that will act as the decorator class. All effects perform some action to the player it affects, so it will also consist of a virtual method for applying the effect. Then, each effect will be a subclass of the abstract class and offer its own feature to the method that applies the effect. There should also be a class equivalent to having no effect, to represent a player with no effect on them. This also allows for the decorator chain to eventually come to an end. Since all effects "decorate" a player, the abstract effects class should also be a subclass of the player class, and also have a player component. This way, every player can be wrapped with multiple decorators to represent multiple effects being applied to them. This eliminates the need to check for every possible combination because it simply involves going through the list of decorators that "decorate" a player. To apply all effects, simply call the method that applies the effect for the current decorator, and then call the same method on the decorator it points to.

If more effects are invented, all we need to do is create a new subclass of the abstract effects class and provide its unique feature to the virtual methods. Now, it can be used, just like all the other effects, as a decorator of the player.

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.

To design our system to accommodate the ability to add and change new or existing command names requires a mix of inheritance and the Decorator Design Pattern. Originally, with a set number of commands, a simple chain of if-statements in the controller can check which command to call based on the user input. Now, with the ability to add new commands and change existing ones, we need to make each command a subclass of an abstract command class. In the abstract class, it will have a string field to represent the name associated with the command, and a virtual method that performs each command. Each concrete command subclass will start off with the default name supplied for their command, and will override the method with their own command feature. To add new commands, it will involve creating a new command subclass that inherits from the abstract command class, so the only recompilation needed is for the new file.

Within the controller, a list of command objects for each default command will be created. At the time, we didn't know which data structure to use, but now, it seems right to use a vector to store command objects. Whenever a user inputs a command, it must be checked with the name of each command in the vector to see if it matches at least the prefix of it. If it does, the command object will call its method to perform the command. If the user asks to rename a command, go through the list to find the specified command, and change the string field in the object to the new name.

To support the ability to give a name to a sequence of commands, we will use the Decorator Design Pattern. Concrete commands can be treated as decorators of the game, so the abstract command class should inherit and have a game component. There should also be a class that represents a game with no command, to make sure the decorator chain comes to an end. Now, when sequences of commands are asked, we can create a new command object that is a chain of decorator commands chosen from the vector of available commands in the controller. Finally, we give this new command object the name the user inputted, and add it to the vector of available commands. Now, the vector of commands acts as a macro language for commands or sequences of commands to the user's choosing, and can be used or changed however they like.

Extra Credit Features

Coloured Text (Visual)

We wanted to give the text display a similar feel as the graphics display. We thought that distinguishing different types of blocks with characters wasn't good enough, so we added color to the text display. The text display now color-codes the different blocks using the same colors as those in the graphical display. This was done with the ANSI escape codes and using them after the cout command. The challenging part with this was researching the different kinds of ANSI color codes and how to use them with the output stream.

Layer (Special Action)

We added an additional special action called Layer that is triggered exactly like the other special actions. This special action, when used, creates a layer of permanent and unremovable blocks at the bottom row of the opponent's board, shifting all other contents on the board up by one. Note that this shift can cause opponents to lose immediately (if the block being dropped no longer fits on the board), and this case is considered in our code. Multiple calls of this special action would create multiple layers of these blocks. In the text display, this block has cell's with the letter B. In the graphics window, the block has cells that are gray in color. The challenging part with this was adding a new block to the game that works differently than other blocks. We also needed to modify the board by shifting all of its contents up by one row. We solved this by adding a new BrickBlock subclass that spans the entire bottom column of the board. Whenever it is initialized, we remove the top row of the board, add a row to the bottom, and fill it with the brick block.

Hold (Command)

The player now has an additional command/option that allows them to hold their current block instead of dropping it. A player can only hold 1 block at a time. If the player is currently not holding a block, the current block becomes the block that is held, and the next block becomes the current block. If the player is currently holding a block, the player's current block and held block are swapped. This command can only be called once per turn. The challenging part of this feature was considering all the use cases for the command, and making sure it didn't provide too much of an unfair advantage. We had to go through lots of testing to know when the hold command can be used and what to do in each case.

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?

This project taught me the importance of planning as a team and writing good code. It was essential that we divided the work in such a way that each person's work would be relatively independent of someone else's work. For example, in our group we initially split the work so that one person can create the Block classes while the other would create the Level classes. Since the two tasks have relatively low dependency on each other, the work was smooth and efficient. On the contrary, one of us eventually decided to take on creating the Player class. However, this class depended on many other components of the game, which were either incomplete or hard to understand. As a result, we were forced to temporarily stop our work with the Player class, and decided to finish the standalone components.

Our code should also support a certain level of abstraction in order for other group members to use it without knowing exactly how it was made. Other group members should be able to use the code that you created, without having to learn how it works, but instead should only need to know how to use it. This was done by creating functions that are solely designed to be used by other components to create ease of use. This ties into the designing of our game, since it helps greatly if our code features low coupling. As a result, we created many small functions focused on smaller tasks, in order to be used throughout the code. This not only reduces unnecessary repetition, but it also makes it much easier for other members to use your code. It is also extremely important to write code that is clear, concise and easy to understand, for not just yourself, but for other programmers. It is extremely difficult to fix broken code written by someone else. Comments are a very easy and helpful way to make your code easier to understand for both yourself and other programmers.

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

A major thing we would have done differently would be creating the components in a different order. When creating the project, we started with the Game and Player classes as centerpieces of the project. These classes referenced a lot of other different components of the project and tied them together. This led to us constantly switching gears and having to focus on creating other components. As a result, in the middle phase of our project, we had a lot of incomplete and unorganized code that was waiting to be pieced together. Instead, it would be much easier to start with standalone parts/modules of the project that have low dependency on other parts. A great example of this is to start the Level class and the implementation behind generating blocks and other independent components. Centerpieces like the Player class should be tackled in the end when all of its needed functions are readily available.

Another major change I would have made would be to put more thought into how the code would look during the designing process. When creating our plan, we never took into account the possible extra features we might be adding. As a result, when creating the layer special action, we encountered more trouble than what was necessary. More specifically, there were not available functions that could be called to perform the task at hand. The functions that were available were so specific to its task that they cannot be used on a broader scale. This problem could be easily solved if we divided these larger functions into smaller more general functions that could then be called to do specific tasks. As a result, many new functions had to be added throughout the components in order to complete a seemingly simple task.

Conclusion

Overall, this project has been a great learning experience for our whole group. We were content with our use of all the OOP techniques learned in class in such a large-scale application. In the end, we were very happy with our end result, and hope for more opportunities like these in the future.