Biquadris Final Documentation

By: Ingrid Qin, Jen Tat, Kimberly Ko

Introduction

As a team, we chose to recreate the game Biquadris because of its similarities to Tetris, a game in which we deeply enjoy competing against each other in, via Tetr.io. This is evident in the way we strived to do our best to accurately depict the game based on the original through the way blocks move to the colour choices in our graphical display, and through every other decision we made throughout the project's lifetime.

Overview (describe the overall structure of your project)

Our implementation of Biquadris features main game operations including graphical and text displays, block placement, boundary checking, line clearing, score calculation, level setting, and special actions.

Our design utilizes the Model-View-Controller (MVC) design pattern. Our view class is XWindow (window.cc), which displays the Biquadris game boards using X11 Graphics. Our model is the class Board (board.cc), which stores information about the grid: placed blocks, the current block, and next block. Board has methods which can choose/set the next block based on level, and methods to clear rows of blocks and update player score. We have 2 instances of Board: board1 for player 1, and board2 for player 2. Our controller, Controller (controller.cc), communicates with both XWindow, board1 and board2 to facilitate gameplay. Controller acts as the "brain" of our program, by controlling turn-taking, restarting the game, storing score and level information, and processing user inputs to send to the model.

For block movements, we have an abstract Block class that serves as the base class for all our block types. Each block has a vector storing its coordinates for each of its squares and a block orientation. Given this information, it is able to be rotated both clockwise and counterclockwise, moved leftwards, rightwards and downwards, and dropped. We also keep track of the block's "next coordinates" allowing us to easily do boundary checking on the coordinates before officially setting the current coordinates based on how the player wants the block to be moved.

There is an isInBoundary() function in the Controller class to ensure that the block movements are valid. It essentially checks the desired coordinates of the block to verify that the coordinates of the block are in the bounds of the board (18 rows and 11 columns) and if they can be placed in a vacant spot because a block cannot be placed in any cells that are already being occupied by other blocks. We include this function to make sure the player moves are valid.

Other essential functions include those that handling score. We have a vector that holds all of the blocks that are currently on the board (this is done for each board for each player). This is a vector of pair, where the pair holds both a block and the level that it was generated. We keep count of the number of rows that have been cleared by the player that will later be a value used in score calculation, and using the vector of blocks and generation levels, we loop through and see if the block was cleared by seeing if the block was in any of the cleared rows. Using getters and setters, we are are then able to update the score for the player and then check if the score is greater than the high score and update the high score, accordingly.

Our final design differs from how we planned out the design to be initially. For instance, we initially intended to use both MVC and a Decorator class to help us apply effects to the blocks and the board. Hence, we initially planned on using a *BoardVariations* class. However, based on our needs, we realized this would add more complexity than required as we can allow the controller class easily to achieve our goals by adding the required functions. Our Board is not a superclass as depicted in our initial UML idea. Instead of having a PlayerBoard subclass, we realize we include two boards corresponding to each player in the Controller class and use a variable to decide which board we want to update. Since the controller class is updating the board, we realize we should include a has-a relationship with the Block class as they need to be placed on the board. It is important to note that we changed pointer types to smart pointer types in our code and in the UML to ease memory management. Finally in our UML, we added a class to deal with graphics which we did not have initially.

Updated UML

Please refer the last page of this report or to this link: https://lucid.app/lucidchart/3511da3a-5d9c-42b5-8280-4c1c98aef7ab/edit?viewport_loc=-6324%2C-1106%2C6621%2C3120%2C0_0&invitationId=inv_e9d3ccd8-cbdc-4747-87e5-4657cbff2192.

Design (describe the specific techniques you used to solve the various design challenges in the project)

Some techniques we used to solve the various design challenges in the project include using smart pointers. To avoid an abundance of memory issues ranging from leaks and needing to worry about deletions, we utilized shared pointers and unique pointers across our code. We quickly realized the benefits of smart pointers as they automatically managed memory. We utilized shared pointers when referencing Block objects and Board objects in the Controller that handled all of the main game operations. We used unique pointers for the Board objects because they are good at representing ownership. When one object dies, the unique pointer fields will run and clean up the associated object. We used shared pointers for the Block objects because both the Controller object and the Board object use Block objects, we wanted it so that some memory is deleted only if all the pointers pointing to it are deleted.

We also took full advantage of C++ libraries and data structures as it would alleviate some of the code we would need to write and it took care of many frivolous worries so we could focus on the critical aspects of implementation. Some of these include deque, vector, cmath, etc.

Another technique we exercised is the utilization of getters and setters. We have many classes, which also means there are many variables of different visibilities, and some variables that we want to keep private/protected. To ensure we respect these fields but to also be able to read them and modify them, we used getter and setter functions. Such fields include player scores, block coordinates, etc.

When using source control, namely Git, we made sure to follow the practice of committing early and committing often. This became increasingly more important when we wanted our team members to also take a look at our work to help debug or for them to be able to get an idea for what to do in their contribution of work. It was also important because we wanted to ensure that all of our work is saved and never lost. With the deadline quickly approaching, we did not want to have to face the possibility of losing work.

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

Our design supports the possibility of various changes to the program specification through how our code utilizes abstract classes. For example, if new types of blocks need to be added to our design, it can be easily done because we have an abstract class for Block. We also have a function, "triggerSpecialAction()," that takes care of all special actions and effects, thus new special actions and effects can be easily added through this function. Furthermore, our Controller class, which oversees all operations and objects, can easily have new classes and modifications be added.

Answers to Questions (the ones in your project specification)

1. **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?**

To design a system to allow for some generated blocks to disappear from the screen if not cleared before 10 blocks have fallen, in the Board class we could add an integer variable to our existing design named **clear** keeping track of when a row in the board was last cleared. For instance, if the row was cleared in the current round, we reset **clear** to 0. Each round the board is not cleared, we increment this variable by 1. Once this variable reaches 10, we clear the last row of Board.grid. Then we clear the last row and shift all other blocks one row down. This can be easily done as we keep track of the locations of each block.

This generation of such blocks could be easily confined to more advanced levels.

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

We could design the program to accommodate the possibility of introducing additional levels into the system with minimum recompilation by simply updating our variable indicating the level (attributes level1 and level2 in Controller). If the new levels affect block probabilities, we can edit our chooseBlock() function to add in the probability of each block being used at that level. If the new level includes special features (like heavy, blind, force), we can create a new methods in Controller, similar to how heavy, blind and force were implemented.

3. **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?**

We could design the program to allow for multiple effects (blind, heavy, and force) to be applied simultaneously because we have a function in Controller that handles all of the special effects, instead of storing a single variable that tracks a single special effect that is being triggered, we can create a vector with all of the desired special effects that need to occur simultaneously, and then be able to trigger them simultaneously. We can easily add more special effects because of how we implemented our Controller. The Controller does not have all of the special effects, it simply accesses them, thus when a new effect gets created, it can easily access it.

4. **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.**

We could design the design system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to the source code and minimal recompilation because our code follows the Model-View-Controller (MVC) Architecture creating a controller class that takes in input commands, processes them, and then tells our model what to do.

It would not be too difficult to adapt our system to support a command whereby a user could rename existing commands since the MVC Architecture allows for the Controller to handle input and facilitates control flow between classes, in a way such that it can communicate with the user for input.

We could support a "macro" language which would allow us to give a name to a sequence of commands by having the controller class process the macro into a sequence of individual commands that will be translated into function calls that then model and view can execute. We do this by using the **processCommand** function in our Controller class.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

The extra credit features we implemented include smart pointers, namely unique and shared pointers. To avoid an abundance of memory issues ranging from leaks and needing to worry about deletions, we utilized shared pointers and unique pointers across our code. We quickly realized the benefits of smart pointers as they automatically manage memory. We utilized shared pointers when referencing Block objects and Board objects in the Controller that handled all of the main game operations. We used unique pointers for the Board objects because they are good at representing ownership. When one object dies, the unique pointer fields will run and clean up the associated object. We used shared pointers for the Block objects because both the Controller object and the Board object use Block objects, we wanted it so that some memory is deleted only if all the pointers pointing to it are deleted. Essentially, we utilized them for their convenience because we did not have to worry about calling "delete" or needing to worry about where to call "delete."

Smart pointers were challenging to implement because it was the first time, we as individuals have ever been exposed to smart pointers and we had to fully understand each of the types of pointers, and when to utilize each. To overcome this challenge of understanding and comprehension, we relied mostly on trial and error, trying out the use of smart pointers, seeing what the specific errors are during compilation, and then handling them accordingly. We also made sure to reference course notes and C++ articles from online.

Final Questions (the last two questions in this document)

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 us how vital open and honest communication is to the development of software. We noticed we were able to avoid many issues and conflicts by having in-depth discussions early on. It is also quite advantageous to do so as we were able to bounce a lot of ideas off of each other and work together to create a plan to face identified challenges.

Another important lesson this project taught us about developing software in teams is how effective and useful source control is. Git is such a powerful tool that we took complete advantage of. It was really important for us to be able to collaboratively together, see each other's work, and be able to cohesively bring all of our contributions together. We also learned how important it is to communicate with one another about our work, as we often faced merge conflicts and git issues.

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

If we had the chance to start over, we would use the opportunity to go more into depth and breadth with our UML Design. We realize that there were some issues and conflicts with our first UML design as we implemented our project. Some parts of our UML Design could not be implemented and other parts needed to be further fleshed out. In our UML Design, we followed a Decorator pattern, but we realized that it was not functional.

Another thing that we would have done differently if we had the opportunity to redo the project is start coding and actual implementation earlier. We promptly started discussing, brainstorming, and planning, but the actual implementation was not until later. We realize that we could have benefitted greatly if we had a greater amount of buffer time.

If we were given the chance to start over, before attempting to fully execute the implementation of our project, we would also thoroughly read the project specifications, take notes, and make sure every detail has been acknowledged while planning and creating a UML Design. This would have been very beneficial because midway during our project, we reached a roadblock associated with calculating the score that also takes into account the specific blocks and their time of generation.

Conclusion

In conclusion, as a team, we are quite pleased with how our rendition of Biquadris turned out. We found it just as entertaining to create as it is to play an actual game of Tetris. Although there were some minor hiccups throughout, in regard to graphics, score calculations, block rotations, amongst other challenges, it was gratifying seeing our final product and being able to work seamlessly as a team.