CS246 Project – Biquadris Final Design Document

Group Members: William Bai, Peter Qi, Tian Xia

OVERVIEW

The project goal is to produce the video game Biquadris, a Latinization of the game Tetris, with main difference at two player competition. It enables a side-by-side display in a turn-based competition with two players and multiple adjustable levels.

The game itself is written in C++, follows Object-Oriented Programming design principles, and incorporated with design patterns such as Observer, Factory, Visitor, and MVC model and concepts native to C++ environment such as Polymorphism, Encapsulation, Inheritance with attention to principles such as RAII, single responsibilities, coupling and cohesion.

It utilizes the MVC model, where the players interact with the game by providing input to the Player class to achieve actions, which is the controller component. The Player class is responsible for implementing the actions and operations through the Board and Block classes, which is the model component. Any changes would then be reflected and updated on display by the Display class, which is the View component.

DESIGN

Classes and components

- Block with its subclasses

Block and its seven direct subclasses are the concreate and functional classes for accomplishing the generation and operation of blocks in the gameplay, as part of the Model component in MVC structure. Created using the Factory design pattern, the blocks are generated with dedicated creator method to facilitate the block creation in Board class. When they become the current block of one player, universally initialized to the middle of the board, each with different move mechanisms.

It is then attached to the Board to enable display and operation commanded by player. The properties such as direction enumerated, current heaviness of block, character indicating types are stored in each block along with four grids taken up by the specific block stored separately with Posn structure for the purpose of easier gird management, space checking, row clearing and score counting. Through the four Posn stored, each of the left, right, down and drop moves can be achieved by checking the corresponding grids under each of the stored gird, for direct access and assess whether it is able to perform the move.

The generic left, right, down, drop moves are implemented directly in Block class since the underlying logic is identical in such function. The clockwise and counter clockwise rotations are implemented independently in each subclass to distinguish the different types of block rotation behaviour. The above structure also has the advantage of interacting with the board when checking boundaries for performing moves and rotation around the edge of the board to ensure no undefined behaviour and no partial blocks outside the board.

- Board inherited from Subject

Inheriting from the Subject, the Board class is designed to be a concrete class with fields and methods enabling operation, determining the game ends or not, and different states/changes trigger by action or special action with method to get the current block the player is operating. Within the Subject class, changes are past to both observers, text display and graphic display by notify method, and regenerate new displays by attach and detach methods when restarting.

As the bridge between Player class and Block class, the Board class stores current and next blocks for the player and manages the game grid. It centralizes the function of calling block generation, ordering moves to blocks, checking board status and clearing full row when needed.

The inherited field of 2D vector gird manages all the position and content in the board, interacting with blocks to allow board to operate on the current block and clear any row through checking grid. It is also responsible for performing special action when the Player class calls the corresponding special action methods in SpeicalAction class.

There is a Bridge Design Pattern variant between Block and Board, the Board interacts with different types of Block objects via generic operations. So all the operation is listed in Board to accommodate the different types of Block objects in action.

- Special Action with its subclasses

Special Action is implemented with a Decorator design pattern in SpecialAction class, with various actions such as Blind, Heavy, Force in subclasses. SpecialAction as a Decorator class wraps on concrete Player class to apply special actions chosen by player. Via the board pointer in Player, such operation is enabled while also reserve possibility to implement more effects or combination of effects.

- Level with its subclasses

Level is designed to be an independent abstract class; we use methods to adjust levels inside Player class and methods to respond and implement the adjustment in levels inside Level's concrete subclasses (e.g. Level0). This allows the intended interaction between independent classes of Player and Level, dedicated for each individual level's parameters. Inside the concrete subclasses of Level, Level 0, 3, 4 have fields to use file input for fixed block generation. And Level 1 through 4 all have the methods to randomly generation with different rate. Level 3 and 4 specifically have both set of block generation methods to implement the corresponding random switch to work to facilitate testing and mode switching.

- Player

The Player class is the concrete class with ability to perform actions and special actions triggered by the player, implement any command for players, and options to adjust level and board settings with pointers to own Board and Level classes, pointer to opponent Player class. Player has the ability to interact with own or opponent's Board to achieve the actions to operate blocks through Block and special actions on opponent's board through SpecialAction.

The interaction with Block object is done with a modified Visitor design pattern where there are sets of corresponding methods in Player and Block classes to enable the "visit & accept" relationship, allowing such modification on Blocks.

The interaction with Level object is done with a pointer to the Level object. Leveling up and down obtain new Level object for the player, enabling the different block generation mechanisms associated with different levels, whether through file or random generation, reflected in the corresponding Board object.

Player class as the Controller component of the MVC model has the ability to use command to achieve move of blocks, clearing rows, and adjust levels without regarding the specification in each level. Move operations of player are parsed from input to be linked with the methods of making such move in Player class, which communicates with Board class to achieve the actual move on Board and associated Block class. Level operations of player are directly applied to the current Level object associated with the player to introduce the new, desired Level Object. And if special action is triggered, the opponent is mutated via the Player pointer to apply the special action of choice, with SpecialAction class decorators on opponent's Board.

- Text Display and Graphic Display inherited from Display Observers

To enable text and graphic displays in the design, we adopt Observer design pattern where the Subject class has subclass Board class, and the abstract DisplayObserver class has subclasses TextDisplay and GraphicDisplay dedicated for corresponding type of displays with Board class pointer to access Board information. Each of the display observer have two Board pointer belonging to each player to produce side-by-side board display during the game play. Player information such as level, score, next block is displayed with the board information and current block in the middle. Graphic Display in addition have colour-coded blocks for better visualization.

Features

- Moves

Player is allowed to operate the current block with moves such as Left, Right, Down, Drop and Clockwise or Counterclockwise rotation to manipulate the position before it is dropped. Valid move with available space on board would result in the change of block position displayed; otherwise, the block would not move as a result of no available space. Rotation actions behave similarly, except the rotation preserves the lower-left corner of the block and checks different spaces required to perform rotation.

All moves and rotations would check the boundaries in addition to move specific board requirement. Near the edge of board, the moves and rotations do not move blocks outside or partially outside the board.

- Adjust Level

The game is initialized to be Level 0 for both players, but player can choose to level up or down for better scoring. During game play, the level up and down is effective instantly and applies to the next block automatically. Each player can only change own level, the two players does not share level information and choices.

In addition to change current level, the Level 3 and 4 have options to disable and re-enable random block generation with filename supplied as alternative input source.

- Restart

One player has the option to restart the game which clears both game board and restarts the game on the previous level. Maximum scores for both players are preserved while their current scores are reset to zero.

- Special Action

When one player clears two or more rows at once, special action is triggered with the following three actions: Blind, Heavy, Force. Blind covers certain area in opponent's board until one block is dropped. Heavy sets opponent's current block with heaviness of 2, which means left or right move is always followed by 2 rows down, and drops the block is not enough space. Force changes the opponent's current block to be one of the player's choosing.

As SpecialAction is implemented with Decorator design pattern, multiple effects can be wrapped onto the same player when such situation is triggered. Multiple effects on the same player can be in effect simultaneously in a wrapped manner and all ends when one block is dropped.

- Command Shortcuts and Multiplier Prefix

Player can use command line input to operate the block in game. Also, the command interpreter and parsing enable shortcuts for existing command. As long as the input is enough to distinguish between other shortcut for existing command, it is a valid shortcut and input. For example, lef is enough to distinguish the left command from the levelup command, so it is a recognizable shortcut for left which performs the left operation.

In addition, command interpreter can take a multiplier prefix, indicating that that command should be executed some number of times. For example, 3lef means move to the left by three cells. Such multiplier prefix can be used in combination with the shortcuts.

RESILIENCE TO CHANGE

Adding Players

Player class can be instantized to be a specific player, with corresponding Board class object of its own. Such design allows the main/control to initialize more players as desired, all with the same specification and properties to allow game upgrade to accommodate more players in each parallel game. Via shared pointers of Player class and Board class, such addition can be introduced without interrupting other functions and classes of the game. As a controller, Player only handles the commands and operations player wanted on current blocks, demonstrates high cohesion principle as its methods and functionalities focus on the tasks of achieving player action.

Adding Blocks

Currently, there are seven types of blocks with generic directional moves, separate clockwise and counter clockwise rotation implementations. If new block types are introduced, new subclasses can be added as inheritance with only the need to implement rotation operation specific to the block shape. Then, the existing mechanism of attaching, detaching, moving, clearing of blocks would accommodate the new block type, making such addition easy to implement and requires minimal recompilation.

Adding Levels

Currently, there are five types of levels with two types of block generations, random and from file with different probability chances. If new level types are introduced, new subclasses can be added as inheritance with fields and methods in subclass to achieve more rules and variations. Only players can adjust the levels and its subclasses are relatively independent, it demonstrates low coupling principle as different modules has low dependencies on each other.

Adjusting Board Specification

The position and grid in the Board class is done through 2D vector of structure Posn. Resize is implemented through standard operation of vector, thus the dimension of the Board can be easily altered to fit new requirements. Such change does not impact the high-level implementation of existing data structures and methods.

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: As previously proposed, a Boolean variable invisibility is added in each of our block types (SBlock, ZBlock, LBlock, …), set to true when it should be invisible. As a result, charAt is called on such object with invisibility set to true, the function will return a whitespace character. An Integer counter is also added to account for the generation of invisible blocks when blocks are not cleared after 10 blocks have fallen and the counter resets. When the counter reaches 10, the creation of the next block will have true initialized in the Invisibility field. To achieve this, we need to add an extra field in the constructor of each block Type (SBlock, ZBlock, LBlock, …) for object creation.

For example, suppose that a block, named B, is moving from the top of the board to the bottom until it hits another block. To determine whether B can continue moving downwards, our program checks whether there are any blocks to the bottom of B by simply calling on charAt at the indices below. If charAt returns a non-whitespace character, there are blocks below B and B cannot continue moving downward. However, this implementation will not work with invisible blocks present since charAt will return a whitespace even when the given row and column are in the range of the invisible block. Therefore, we will throw an exception with the actual character pattern of the bock in charAt whenever the given row and columns in charAt is inside a box with Invisibility set to true. When we check for whether there is a block at a specific row and column, we can implement a try/catch to account for invisible blocks as well. This will also allow us to see the underlying character in invisible boxes.

The generation of such blocks can easily be implemented to higher levels since our class Level, where additional levels will be implemented, will not affect our block Type classes, and the function charAt should not change with different levels.

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

Answer: Levels are defined an abstract class with a subclass for each of the levels. These subclasses contain fields that are initialized to various values based on the level's specific properties such as file use and different random rates. For example, to implement the Heavy special action, we will include an integer field in our Level subclasses called Heaviness.

This implementation will require minimum recompilation when the new level is added as the subclass inherited from abstract Level class. Thus, no need to recompile other Level classes, Player or Board classes. For example, if we want to define a new level called level5, we can define the corresponding properties of the new subclass of Level, and no changes need to be made to other classes.

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: We use decorator design pattern to allow for multiple effects to applied simultaneously. The Abstract base class is SpecialAction class, wrapped to the Player class to allow effects to be applied to Board and Block. The Concrete Decorator subclasses are Blind, Heavy and Force. We have a pointer of SpecialAction stored in class Player.

Once a player triggers and chooses a special action, for example Force, the action would point to a Force class and then performs the corresponding action implemented in the Force class method. If the second special action is in effect while the previous lasts, it would then be created and wrapped on the existing special action. Each time the special action is accounted for, both actions would be applied on Player. Once a block is dropped, all special action lost effects, we would delete the pointer of the SpecialAction class in player to restore.

Thus, such method of implementation prevents your program from having one else-branch for every possible combination and decorator design pattern has advantages on this area and implemented as subclass, new special actions can also be introduced easily with minimal recompilation.

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: We adapt STL container data structure map to accomplish the requirement and methods to execute single command one at a time. Maps are associative containers that store elements formed by a combination of a key value and a mapped value, forming a unique key-value pair. In this map, the key values are type string used to sort and uniquely identify each existing command name in full, while the mapped values of type Vector of strings store the individual command(s) associated to this key.

We would input the command name(s) to operate, the value would be retrieved as a sequence of command(s) associated with the command name. The map ensures the association between the high-level command interface can fetch the corresponding internal implementations of the command with pre-defined command as listed. This structure enables the system to accommodate the addition of new command names, changes to existing command names since these values are stored as keys in the map, which can be easily altered while preserving the internal command implementation association. Modification of command names does not change the originally assigned sequence of commands. Also, if we wish to add/implement new command with new features, this structure simply allows the addition by inserting a new key-value pair to the map. And since the value is of type Vector of strings, one command name can support a sequence of commands all stored in the form of Vector, with the method to execute individual commands, each element in this Vector of commands would be executed sequentially as desired.

The above structure requires minimal change and adjustment and offers the minimal recompilation since the only change concerning the above function does not change the overall structure and only includes necessary implement addition. No other associated parts need to change as this structure minimizes its affect to other features and functionalities. On a "macro" level, the shortcuts of existing commands can be identified by comparing input command to the keys, the input would be identified as valid command name if it is only one keys in the map starts with the input string. If the inputted command name matches the beginning of more than one key, it is not a valid input as it cannot be distinguished from other shortcuts of existing commands. This structure achieves the above functionalities while preserving convenience of applying shortcuts without additional implementation.

EXTRA CREDIT FEATURES

Memory Management

In this project, smart pointers are used for memory management, with no delete keywords in implementation. No manual and explicit handling of memory is required in this case. Also, by pointers are stored inside vector containers, in combination of smart pointers, the entire project is completed without leaks, and without explicitly managing own memory.

Hi Score Display

No max scores for the players are required to display in specification. We added the Max Score Display beneath score of each player to show this information which is remained unchanged when game restarts (i.e., board is cleared and current score resets to zero).

New Command: Rename

We implemented a new command named "rename" which is used in the following format: "rename A B" to rename the current command named "A" to new command name "B". Now, enter command "B" calls the original function represented by "A".

In addition, rename commands is restricted to changes that does not conflict with existing command names. Rename commands "rename" itself and change command names to existing ones are prohibited, regarded as invalid command. On a "macro" level, the shortcuts and multiplier prefix functionalities are persevered.

New Command: List

To accommodate the renaming and provide reminder to player during game play, we implemented a new command named "list" (which can also be renamed) to display the set of commands allowed for player. All the renaming changes are also updated and displayed in "list" command.

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: Developing software in teams has to constantly communicate and update about the relevant parts with other members to prevent code conflict and misuse of new changes, as well as the presence of wait time for allowing sequential development commissioned by different members. This difference often introduces more fractions about different ideas, frustration when trying to understand other's implementation, and difficult to convince others when trying to advocate new changes. But the upside is, instead of tackling challenges and difficult designs entirely alone, you have more creative solutions, proposals and dedicated work distribution helping along the stage of development that results in better overall design, efficiency, and reliability.

The most critical lesson is to be humble and always actively listen to other members. The first thing is to realize other's capability and potential and realize own's weakness and mistakes.

After getting into sync, the combined productivity is greatly stretched to accommodate and make-up the time used to discuss directions, solving version issues, and overcome blockers more effectively. More importantly, without the designer/implementer's own assumption and knowledge of the underlying data structure, other members are more likely to spot the incoherent places, untested usage bugs, and potential improvements. Overall, working with group members that have aligned goals, reasonable response rate and optimal efficiency made up a unique experience similar to industry standard, understand the importance of teamwork, and introduces new possibilities and responsibilities as future developers.

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

Answer: If to start over, we could schedule more in-person sessions prior to and during the implementation of separate parts to fasten the overall decision making and discussion progress. Since we all have other projects/assignment parallel to building Biquadris, we scheduled most session via Zoom meeting which is a convenient way to quickly meet up and discuss any urgent issue affecting the project direction and design, be more focused on the project and get the core implementation done in a relatively more concentrated period, spending less time on compiling and debugging each other's parts.

Also, in terms of design, change would be made to board to be more adaptive to multiple players, allowing multiple players' board to be created during the game. Also, if more time is available, eliminating the use of public fields and methods for accessing to improve encapsulation and reduce coupling. Such change would improve user-interface, readability, data and memory management and maintainability.