CS246 FALL 2024 PROJECT – BIQUADRIS: FINAL DESIGN DOCUMENT

Andrea Cen, Nicole Cui, Sophia Yang

1. INTRODUCTION

The Biquadris project is a 2-player adaptation of Tetris designed to integrate the core mechanics of the original game while incorporating new features to enhance gameplay. This project focuses on applying object-oriented programming (OOP) principles and design patterns to build an interactive, turn-based version of Tetris. It serves as a practical application of key design and programming principles learned in CS 246.

2. OVERVIEW

Describe the overall structure of your project.

The game is implemented as a collection of interdependent components, each with distinct responsibilities. These components collectively manage gameplay, player interactions, game state, and visual representation. Below, we provide an overview of the structure of the program, detailing each component as it pertains to its role within the system.

Main

main.cc

`main.cc` is the entrypoint of our program. Its primary responsibility is to receive command-line arguments at execution and user commands on the terminal during gameplay. `main.cc` catalyzes the start of the game and passes all received arguments to `Game` for interpretation.

Game

game (.h, .cc)

`Game` represents the central game engine of Biquadris. Its primary responsibility is to drive the execution of the game, namely game start, end, restart, and play. It ensures continuous and dynamic flow by handling player turns, applying special actions (when appropriate), and passing user commands to `Player` to invoke changes to `Player` boards. It is a key provider of data used to render both text-based and graphical displays.

Players

player (.h, .cc)

A `Player` is a representation of an individual player of the game. Most significantly, a `Player` has a `Board`. The primary responsibility of a `Player` is to store and modify its `Board`. Based on arguments received from `Game`, `Player` invokes `Commands` (user commands and special actions) on the contents of its `Board`.

`Player` also stores all information pertinent to any individual player of the game: namely current `Block`, next `Block`, player `Level`, and score. When appropriate, Player will update its information and its `Board`.

Board

board (.h, .cc)

A `Board` is an organized representation of a typical Biquadris gameboard. Its primary responsibility is to store and manage information on each square on the board. `Board` implements low-level modifications to and information access of `Blocks` on the board. Note: a 'square' refers to a square on a row/column grid system used to represent a gameboard

Blocks

block.h, {i, j, l, o, s, t, z, star} block (.h, .cc)

Each type of `Block` inherits from an abstract `Block` class (see Template Method Pattern in this document). Its responsibility is to store `Block` information: namely start position, type, level of generation, unique Block ID to differentiate it from other `Blocks`, and information on how to modify its coordinates/position during a specific rotation formation.

Display

game (.h, .cc), subject (.h, .cc), observer.h, textobserver (.h, .cc), graphicsobserver (.h, .cc), window (.h, .cc)

Display of this game is implemented using the Observer Design Pattern (see Observer Pattern in this document). It's composed primarily of the following subcomponents:

- `Subject`: notifies `Observers` of changes to a gameboard and that display should be refreshed
- `Observers`: `TextObserver` to implement text-based display, `GraphicsObserver` to implement graphical display
    - `GraphicsObserver` uses the C++ graphical library `Xlib` to render graphical display
- `Concrete Subject` (`Game`): provides information on what should be displayed

Please refer to Observer Design Pattern on this document for more information.

Levels

level.h, level {1, 2, 3and4} (.h, .cc)

`Levels` inherit from an abstract `Level` class and are implemented using the Factory Method Pattern (see Factory Method Pattern in this document). Its primary responsibility is to handle the algorithm that generates certain kinds of `Blocks` of different probabilities depending on level specification. This kind of functionality is used by `Player` to generate its next `Block`.

Commands and Special Actions

command (.h, .cc)

The `Commands` class implements low-level `Block` movements and `Board` modifications based on received user commands (including special actions). Its only responsibility is to determine whether or not a user command is possible, and make changes to a `Player`'s `Board` when appropriate.

3. DESIGN

Describe the specific techniques you used to solve the various design challenges in the project.

Our implementation of Biquadris reflects a deliberate application of the object-oriented-programming principles and design patterns covered throughout the course to address the challenges presented by this project. Here, we elaborate on the key design choices and techniques we applied during development.

OOP Principles

a. Encapsulation and Abstraction

By bundling data and related methods into classes, we have encapsulated relevant information and operations to ensure clear separation of concerns within our program. For example, each `Block` type (`IBlock`, `JBlock`, etc.) encapsulates its type and coordinates relative to its bottom left square. Access to this kind of information is managed and restricted by getter and setter methods.

By dividing this project into its core components, we've used abstraction to emphasize clear responsibilities for each component and create separation between interface and implementation throughout our program. For example, the `Command` class abstracts how user commands (most significantly `Block` movements) are implemented, and the rest of the program does not need to worry about how and whether or not `Blocks` can be moved.

b. Inheritance and Polymorphism

The use of abstract base classes from which individual subclasses inherit from allows for greater organization as well as a clear demonstration of the relationships between components in our program. The act of inheriting from an abstract superclass allows for related objects to be treated polymorphically. As an example, each type of `Block` inherits from an abstract `Block` class, which allows them to be treated collectively as `Block` objects that have access to common methods (such as getters and setters), while relying on their unique implementations of overridden pure virtual methods (`updateCoords()` to manage individual shape).

c. Cohesion and Coupling

We seek to minimize coupling and maximize cohesion throughout our program through the relationships and interactions between our components.

d. Aggregation

An example of a "has-a" relationship in our program is that `Game` "has" a collection of `Players`.

```
class Game: public Subject {
vector<Player> players;
...
};
```

`Game` has access to `Players` but does not own them; the `Players` exist independently of `Game`. This promotes low coupling: `Game` does not directly manage the lifecycle of `Players` or affect its control flow. Further, this kind of relationship allows for `Game` to interact with external `Players` while keeping responsibilities focused, allowing for high cohesion. `Game` is only responsible for the execution of gameplay mechanics and is not concerned with the behaviour of each individual player.

Similarly, `Subject` "has a" collection of `Observers`. (See Observer Pattern)

e. Composition

An example of a "owns-a" relationship in our program is that `Player` "owns" a `Board`, `Level`, and `Blocks`.

```
class Player {
unique_ptr<Board> board;
unique_ptr<Level> level;
unique_ptr<Block> curBlock;
       unique_ptr<Block> nextBlock;
...
};
```

`Player` "owns" its `Board`, `Level`, and `Blocks`, and can thus directly manage its interactions and how they contribute to a single purpose (management of `Player` gameboard), promoting high cohesion. This ensures strongly related functionality of `Board`, `Blocks`, and `Level` within `Player`.

Design Patterns

f. Template Method Pattern

We want each of the `Block` subclasses to inherit a common structure: namely, the presence of attributes such as position, type, level generated, etc., but override certain behaviours. The Template Method Pattern allows each `Block` type to share common characteristics, but dictate individually how it should look after any rotation by overriding the `updateCoords()` method.

g. Factory Method Pattern

We want different `Levels` to all generate `Blocks`, but each `Level` has unique specifications that enable them to generate different kinds of `Blocks` at different probabilities, which different behaviours. The Factory Method Pattern allows the abstract base `Level` class to provide the interface for creating blocks, while subclasses (`Levels 0~4`) can alter the type of blocks to be created by overriding the factory method (`Level::generateBlock()` in this case).

h. Observer Pattern

The Observer Pattern is used in our program to implement two kinds of game display: text-based and graphical. To do this, `Subject` "has" a collection of `Observers`: one `TextObserver`, one `GraphicsObserver`

```
class Subject {
vector <Observer*> observers;
       void notifyObservers();
...
};
```

`TextObserver` and `GraphicsObserver` both inherit from the abstract `Observer` class and override the `notify()` method to display the data on each of the boards. `Game` is the `Concrete Subject`; when changes are made to `Player` boards through `Commands`, `Subject` invokes `notify()` in each of the `Observers`, which then query updated data from `Game` to update text-based and graphical displays. The Observer Design Pattern is especially useful in this case to maintain synchronization between dynamic text and graphical displays.

Further, the Observer Design Pattern promotes low coupling (`Subject` is only responsible for notifying `Observers`, and is not concerned with the details of how the `Observers` display, and nor is the rest of the program) and high cohesion (each class in this pattern has clear and focused responsibilities and share a common goal: to update display).

Other Challenges

i. Efficiency of Board Management

Something that was challenging from a design standpoint was how we were going to represent each `Player's` gameboard in a data structure. Our primary concerns were that it needed to be able to store all the information we needed, resizable, and efficient to search through regardless of how many elements it contained.

From a display standpoint, we needed to be able to access information (the character type of a `Block`) at a board square level. From a logic standpoint, we needed to be able to access information (which specific `Block` a square belongs to, what type of `Block` it is) at the `Block` level to track which `Blocks` had been cleared completely from the `Board` (scoring, `level4` features).

It is for these reasons that we decided to represent our gameboard using a map, such that the key is a `vector<int>` that stores coordinate information for each square filled by a `Block` on the `Board`, and the value is a `vector<string>` that stores the `Block` type, level it was generated in, and unique blockid as an identifier:

```
class Board {
       map<vector<int>, <vector<string>> gameboard;
       // representation of an entry:
       // [<int row = 3, int col = 0>,
          <string type = "T", string genLevel = "2", string blockid = "17">];
       ...
};
```

Using the built-in C++ STL function `find()`, we can very efficiently:

- determine if a square is filled:

```
vector<int> = {row, col};
if (gameboard.find(v) != gameboard.end) // found, do something
// O(log n)
```

- access values of the key:

```
vector<string> value = gameboard[v]; // O(log n)
```

4. RESILIANCE TO CHANGE

Describe how your design supports the possibility of various changes to the program specification.

Our implementation of Biquadris is designed to prioritize flexibility and minimize disruption to the existing codebase. Below, we discuss potential changes to the program specification and possible solutions to address such changes.

Addition of Players

To expand our implementation of Biquadris to a multi-player game, we would only need to do the following:

1. add a `Player` to `Game's` collection of `Players`
2. modify `Game::switchPlayer()` logic to accommodate switching between multiple players `(% n)`
3. modify `Observers` to fit print more `Boards` on display

All `Player` behaviour, and by extension `Board`, `Command`, and special action is abstracted and thus is easily reused/does not require modification upon addition of new `Players`

Changes to Blocks

Modifications to existing `Block` types might include:

1. changes to shape
    a. in this case, modify `updateCoords()` within that `Block` class to change its coordinates on each rotation formation

To accommodate the addition of `Block` types, we would need to do the following:

1. add a new `Block` subclass that overrides `updateCoords()` to return coordinate information for the `Block` at any rotation
2. determine probability of this `Block` being generated at every level
3. implement probability in desired `Level` class's `generateBlock()` method

All logic involving addition/removal/moving of `Blocks` in a `Board` is abstracted by the `Command` and `Board` classes, and is thus easily reused/does not require modification upon addition of new `Players`

Changes to Levels

Modifications to existing `Levels` depend on what changes have been made to the specification. Some possibilities:

1. changes the probability that each `Block` is generated
    a. in this case, simply modify the `Level's` `generateBlock()` method
2. changes/adds command
    a. see (d. Changes to Commands) below

The addition of a new level requires changes that depend on what the level's specifications dictate. At a high level,

1. add a new `Level` subclass that overrides `generateBlock()`

    a. if this `Level` generates existing blocks at different probabilities, a simple override that modifies the implementation of `generateBlock()` is enough
    b. if this `Level` introduces a new type of `Block`, a new `Block` needs to be added to the program (see b. Changes to Blocks above)
2. if this `Level` modifies/adds commands:
    a. see (d. Changes to Commands below)

Note that any changes to how a `Block` behaves (e.g. in Level 5, all O blocks are heavy) is still considered a new/modified command (e.g. modification to existing heavy functionality in `Command`), as `Command` handles all block movements/additions/removals to/from `Player` boards.

Changes to Commands/Special Effects

The addition of a new command, at a high level, would entail:

1. adding implementation of the command in `Command` class
2. adding the command (`string`) as a valid command to be stored in `Game`
3. adding a condition to `Player::interpretCommand()` that evokes implementation of command

Changes to an existing command only require changing the implementation of the command in the `Command` class. Since all implementations of commands are abstracted within the `Command` class, this kind of modification will not require changes anywhere else in the program.

Support for Renaming Commands (including Macro)

Adding support for user renames of existing commands would require changes to `Game::receiveCommand()`. It would primarily affect how the program stores existing commands to check for validity. To implement this feature, we would:

1. use a map: store command string as key, identifier as value
2. if a user renames a command, update key but value stays the same
3. use value to invoke command implementation in `Command`

5. 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?

   DD1 RESPONSE: For each `Player`, we can use a map to represent the player's board, such that the key is the order in which the block was dropped, and the value is a set of coordinates for the block, as well as a counter for the number of blocks dropped thus far without clearing a line. When the counter reaches 10, we would take the block at minimum key and remove it. Then, we would check for any line clearing that may occur after removing the block.

   The logic that generates such blocks is easily confined to more advanced levels through the `Player`'s `generateNextBlock()` function, which calls the overridden `generateBlock()` method in each

`Level` class. The `Block` generated from more advanced `Levels` are added to our map and removed from the screen if 10 more blocks are cleared without a line clear.

DD2 UPDATE: No major update to previous response.

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

   DD1 RESPONSE: We can accommodate the addition of new levels using the Factory Method Pattern. The functionality of each `Level` is encapsulated within its own class, which inherits from an abstract base class `Level`, and overrides the `generateBlock()` function to generate `Blocks` that behave uniquely according to the `Level`. To add levels to the game, we would only need to add another `Level` class that inherits from the base class, and recompile that class, as well as the base class.

   DD2 UPDATE: No major update to previous response. See Changes to Levels above for more details.

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?

   DD1 RESPONSE: Each special action is executed through its own function within the SpecialActions class. To use multiple simultaneously, each of their functions can be called sequentially through the Game class, which applies them to the gameboard. This way, we can avoid having to implement an if-else-branch for every possible combination. New methods will only require the addition of a new function to the SpecialActions class.

   DD2 UPDATE: To apply multiple commands at once, each command's function within the `Command` class should be called sequentially from `Player::interpretCommands()`. To add new commands/effects, see Changes to Commands/Special Effects above.

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.

   DD1 RESPONSE: We can change the properties of an existing command or add new commands by modifying or adding a new function in the CommandInterpreter class. We can store all commands in a map such that the key is the command word available to the user and the value is a unique identifier. The identifier is then used in CommandInterpreter to call the command's corresponding function.

When a variable is renamed, we can very efficiently overwrite its key while the value stays the same. We will still need to check if the command word already exists as a key before renaming it to prevent having multiple commands with the same name. To support shortcuts of command names, we could just check if the shortened input is a unique substring of exactly one command word (key). This allows for minimal modification of the functionality behind the command.

To support a "macro" language, we could simply add its command word and an identifier to our map. When the command is called, we would use this identifier to call the sequence of corresponding functions in CommandInterpreter. The same support for command name shortcuts is available for this macro language.

DD2 UPDATE: No major update; `CommandInterpreter` is named `Command` in our implementation. See Support for Renaming Commands above for more details.

6. EXTRA CREDIT FEATURES

What you did, why they were challenging, how you solved them—if necessary

We challenged ourselves to develop this project without needing to explicitly manage our own memory. To achieve this, we designed the structure of our program with this goal in mind, and thus paid extra attention to the concept of ownership as it pertained to classes participating in "owns a" relationships with other classes or objects. This was perhaps the most challenging part of the project: it was difficult to consider this aspect of our program during the design stage–there was no tangible code or implementation to reference. To mitigate this, we developed the first version of this program while managing memory, then changed all objects "owned" by classes to unique pointers. Since we had already considered ownership as a concept during design, this change was not difficult to integrate. Frequent testing helped us identify and eliminate memory leaks.

Further, we added an indicator over the board belonging to the "current player", to better help identify whose turn it is during gameplay. This is especially helpful after the implementation of command prefixes, and special actions, which apply an effect on the other player's board.

7. 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 provided valuable insight into the process of developing software as a team. We found the most significant takeaway to be the importance of collectively defining clear goals and direction from the very beginning. Establishing a collective vision ensured everyone's thought processes and objectives were aligned, which allowed us to complete tasks independently as we continued development. Further, effective division of work and responsibility, as well as setting concrete deadlines and status updates pushed us to maintain steady progress.

Finally, the use of version control tools proved indispensable in managing complexities while developing code as a team. It facilitated seamless integration of individual contributions and provided insurance while promoting careful coding habits and consistent communication within the group

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

   If we had the chance to start over, we'd devote greater attention to the relationships between classes and how these relationships should be represented and implemented in code, particularly in the beginning of the design process. Understanding what kind of data should be passed to and from classes, as well as what kinds of functionality are dependent on each other would improve clarity and organization of our program.

   Further, assigning each class or component a clearly defined responsibility would allow us to more easily and intentionally determine which class should be responsible for certain tasks during development. This also would have allowed us to better adhere to the single responsibility principle, which would have greatly simplified unit testing and extending features of the system.

   Reflecting on this process emphasizes the importance of thoughtful, thorough design and a healthy, productive group dynamic, providing valuable lessons for future projects.