Biquadris Design Document

Group: Aarav Surana (a3surana), Kunal Sachdev (k8sachde)

Introduction

This document outlines the design and implementation of Biquadris- a two-player Tetris-like game- providing an overview of the system architecture, object-oriented design principles, and techniques employed to address the design challenges. The goal is to present a cohesive and extensible design, ensuring resilience to change while maintaining cohesion and minimizing coupling between components.

A game of Biquadris consists of two boards, each 11 columns wide and 15 rows high. Blocks consisting of four cells (tetrominoes) appear at the top of each board, and you must drop them onto their respective boards so as not to leave any gaps. Once an entire row has been filled, it disappears, and the blocks above move down by one unit.

Overview

The game employs a modular, object-oriented architecture. The project is divided into several core components, each responsible for a specific aspect of the game's functionality:

- **Game Core:** Manages the overall game flow, including player turns, command processing, and display updates.
- **Board:** Represents the game grid for each player, managing block placement, movement, and row clearing.
- **Blocks:** Encapsulates block types (I, J, L, S, Z, O, T), their shapes, rotations, and behaviors.
- **Player:** Tracks player-specific information such as score, level, and game state.
- **Level System:** Generates blocks based on the current level using a factory method pattern, supporting extensibility for new levels.
- **Special Actions:** Implements game effects- blind, force and heavy- designed using polymorphism for scalability.
- **Displays:** Includes TextDisplay and GraphicDisplay for rendering game state in textual and graphical formats.
- **Command Interpreter:** Processes player commands and maps them to valid commands, before being processed by the core Game.

Design

1. Game

The Game class orchestrates the game logic and manages the interactions between players, boards, and displays.

- **Key Methods:**
  - initialize(): Initializes game parameters, including starting levels and scores for both players.
  - start(): Starts the game loop and manages turn switching.
  - processCommand(const std::string& command): Parses and executes commands using the CommandInterpreter.
  - switchTurns(): Alternates the active player by toggling currentPlayer.
  - updateDisplay(): Updates both textual and graphical displays.

2. Board

The Board class represents the grid where blocks are placed and manipulated.

- **Key Methods:**
  - canMove(Block* block, Direction dir): Checks if a block can move in the specified direction.
  - clearRow(int row): Clears a full row and shifts rows above down.
  - dropBlock(Block* block): Simulates a block dropping to the lowest valid position.
  - printTextDisplay(): Prints the current state of the board for debugging or textual display.

3. Blocks

The Block class serves as the base class for all block types (IBlock, JBlock, etc.), encapsulating shared functionality such as shape, rotations, and weight.

- **Key Methods:**
  - rotate(Direction direction): Rotates the block using predefined rotation matrices stored in rotations.
  - placeOnBoard(Board& board): Updates the board with the block's cells.
  - initializeRotations(const std::vector<std::vector<std::pair<int, int>>>& rotations): Sets rotation states for the block.

4. Player

The Player class encapsulates player-specific attributes and operations.

- **Key Methods:**

    - generateNextBlock(): Generates the next block based on the current level's logic.
    - applySpecialAction(SpecialAction* action): Executes a special action affecting the player or opponent.

5. Level

Levels (Level0, Level1, etc.) determine block generation logic, employing polymorphism for extensibility.

- **Key Methods:**
    - generateBlock(): Returns a block instance, with behavior varying by level.
- **Resilience to Change:**
    - New levels can be introduced by inheriting from the Level base class and overriding generateBlock().

6. Special Actions

Special actions are represented as subclasses of SpecialAction.

- **Key Methods:**
    - execute(): Applies the effect.

7. Command Interpreter

The CommandInterpreter class processes player commands dynamically.

- **Key Methods:**
    - addCommand(const std::string& command): Adds a new command dynamically.
    - interpretCommand(const std::string& input): Maps input to corresponding commands.
- **Resilience to Change:**
    - Commands can be renamed or extended by updating the map without altering other components.

8. Displays

- **TextDisplay:** Outputs the game state as text using methods like printSideBySideBoards().
- **GraphicDisplay:** Uses XWindow to render a graphical representation of the game.

Key OOP Principles Used:

➔ Encapsulation:

Each class manages its own state and behavior, exposing only necessary methods through well-defined interfaces. For instance, the Board class encapsulates the game grid, the methods to manipulate blocks on the board and row-clearing mechanisms.

Furthermore, private members ensure that internal data cannot be directly accessed by external classes.

➔ Inheritance and Polymorphism:

Block Types: The Block class serves as a base class and specialized blocks inherit from it to define block-specific shapes and rotations. Polymorphic behavior is seen when calling rotate() on different types of Block objects.

Levels: The Level class is a base for level-specific behavior, and child classes override the generateBlock() method to customize block generation logic. The method is invoked on a Level object without knowing the specific subclass, allowing seamless integration of future level types.

Special Actions: The Special Action class is a parent for all special actions, providing a polymorphic interface for applying player effects. The execute() method can be invoked regardless of the type of special action.

Design Patterns

➔ Factory Method (Block Generation)

Each Level subclass implements its own logic for creating Block objects. This abstracts the block creation process, enabling dynamic generation based on the current level.

➔ Observer

TextDisplay and GraphicDisplay act as observers of the Game state. When the game state changes, these displays update to reflect the changes, ensuring a separation between the game logic and the UI.

Resilience to Change

Our design exhibits **high resilience to change**, primarily due to its adherence to OOP principles and use of design patterns. The low coupling between classes ensures that minimal recompilation and additional code is required when adding/removing/modifying functionalities.

Some examples of this include:

Extending Gameplay

- Adding new block types/changing block shapes:

  This can be achieved by simply creating a new subclass of Block for the new type, defining its shape and any possible orientations (rotations). An example of this was the addition of the special * block for Level 4 gameplay.

  Also, the shapes of existing blocks can easily be changed, if needed, as a means to redefine the block types.

- Introducing new levels:

  New levels can be added to the game by deriving a new class from Level and implement its specific block generation logic.

- Adding special actions:

  Similarly, special actions can be added by extending the parent SpecialAction class and define the execute() method.

UI Enhancements

The separation of the display classes ensures that new display modes/ features can be added without modifying any game logic.

Command Flexibility

New commands can be added to the CommandInterpreter class.

Questions

How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

To implement the feature where some blocks disappear if not cleared before 10 more blocks have fallen, the system can leverage the existing vector of pointers that tracks all blocks. However, if this approach proves cumbersome, an alternative design involves the Board class maintaining a queue to track the last 10 blocks placed on the board. This queue ensures that as new blocks are dropped, the oldest block in the queue is checked for clearance. If a block remains uncleared by the time it is removed from the queue, it is automatically removed from the board. This design isolates the timed block functionality within the Board logic, simplifying the management of block lifetimes. Furthermore, the generation of such blocks could be easily restricted to advanced levels by modifying the level-specific block generation logic, ensuring these blocks only appear when higher difficulty settings are active.

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?

To enable multiple effects to be applied simultaneously, the system uses a polymorphic approach based on a SpecialAction base class. Each effect is implemented as a subclass of SpecialAction with its own execute method. The Board class maintains a list or queue of active effects and iterates over this collection to apply all effects concurrently without relying on complex branching logic. When new effects are needed, they can be added by creating additional subclasses of SpecialAction, making the system highly extensible. This design avoids the need for an else-branch for every possible combination of effects, ensuring simplicity and scalability as more types of effects are introduced.

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.

The system accommodates the addition of new commands, changes to existing command names, and the creation of macros through a flexible command interpretation mechanism. The CommandInterpreter class uses a map or dictionary to associate command names with

their corresponding actions. Adding a new command or renaming an existing one involves simply updating the map, minimizing changes to the source code and recompilation effort. To support renaming commands, a method could update the map with new key-value pairs, allowing users to redefine commands dynamically. For macros, a specialized class can group sequences of commands, storing them as lists within the map. This design enables users to assign custom names to command sequences, enhancing flexibility. By abstracting the command logic through the map, the system ensures that new features or updates do not disrupt existing functionality, preserving backward compatibility. We note that this implementation of CommandInterpreter is currently not what we have implemented, but is a next step in the process.

Final Questions

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?

One key takeaway was the value of version control tools like Git in managing code changes, especially when multiple developers are contributing simultaneously. It also reinforced the importance of writing modular code, as it allowed different team members to work on separate components independently without causing merge conflicts.

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

If given the chance to start over, we would have spent more time on upfront planning and designing a more robust architecture. And rather obviously, better time management and adherence to deadlines for individual milestones would have minimized last-minute stress.

Conclusion:

This design document captures the architecture and implementation details of our two-player Tetris game, demonstrating object-oriented design principles and adaptability to future changes.