Plan of Attack

Timeline

We've decided to go with the Biquadris project and planed out what needs to be done. We'll focus on getting the basic parts of the game working first, like moving and placing blocks correctly. Along the way, we'll get feedback from others and make sure our project report and plans are solid. As we progress, we'll add more features to make the game better, like allowing blocks to rotate and adding special actions for certain events in the game. Finally, we'll wrap everything up, making sure our report and diagrams are complete before the final submission. Throughout the process, we'll work together, test our game, and keep everything well-documented.

Due Dates	**Task Description**
March 13th	**First Meeting Recap** ● Confirmed decision to proceed with the Biquadris project. ● Initiated work on our plan of attack, outlining the tasks and responsibilities. ● Began drafting the UML class model based on our plan. ● Started developing the starter code for the project. ● Submitted group.txt to Marmoset as required.
March 25th	● Aim to have a fully functional game with all basic functionalities implemented, excluding XWindows graphics and special effects. ● Ensure blocks can be moved, rotated, and dropped onto the board correctly. ● Implement board logic to handle block placement, row clearing, and game over conditions.
March 26th	● Complete the UML class model, incorporating any changes or refinements based on the evolving project requirements. ● Ensure the UML accurately represents the structure and relationships of the game components.
March 27th	● Provide a comprehensive explanation of how design patterns are utilized

	within the project. ● Finalize and submit the project report, detailing our design choices, implementation strategies, and any challenges encountered.
March 28th	**<u>DD1 components submitted</u>** ● Conduct peer feedback sessions to gather insights and suggestions for improving the report and UML diagram. ● Make necessary revisions and finalize both documents. ● Submit uml.pdf and plan.pdf to Marmoset by 5 PM.
March 29th	● Complete implementation of interface files, ensuring they interact correctly with the game logic. ● Start developing the command interpreter early to facilitate interaction with the program. ○ Gradually add commands and features, testing each addition thoroughly. ● Begin building a comprehensive test suite for fast and efficient verification of the implementation. ● Determine the approach for random number generation as required by the game.
March 30-31st	● Focus on completing the core functionalities of the game, ensuring that it is fully functional and compiling without errors. ● Ensure that the command interpreter, board logic, block manipulation, and next block generation are all working seamlessly. ● Implement text-based display and user input handling for smooth interaction.
April 1st	● Integrate extra features such as block rotations, graphics (if time permits), and additional difficulty levels (Level 3-4). ● Implement special actions triggered by specific game events, such as clearing multiple rows simultaneously. ● Ensure that special actions, including blind, heavy, and force, are

	implemented correctly and have the intended effects on gameplay.
April 2nd	● Begin drafting the final report, outlining the design of the project, including any deviations from the initial plan outlined in DD1. ● Document how the implementation differs from the design and the rationale behind any changes made.
April 3-4th	● Ensure all extra features and special actions are working properly and thoroughly tested. ● Finalize the UML diagram to accurately reflect the actual structure of the project. ● Prepare for DD2 submission, ensuring all components are completed and meet the project requirements.
April 5th: DD2 @5PM	**DD2 components submitted** ● Implementation of Basic/Final/Special features ● Final Report ○ Outline design of project ○ How it differed from design in DD1 ● Updated UML

Task Distribution

Jane

- Mainly responsible for plan of attack and final report
- Implement the core game logic, including board management, block placement, and row clearing.
- Ensure the correct functionality of game mechanics such as block movement, rotation, and dropping.
- Handle game over conditions and scoring calculations.

Grace

- Mainly responsible for UML and updating the UML for DD2
- Develop the text-based user interface and command interpreter for interacting with the game.
- Implement user input handling and command execution.
- Ensure seamless interaction between the player and the game components.

Ivy

- Mainly responsible for peer feedback and implementation design
- Implement graphical display using XWindows graphics (if time permits) and color-coding for block types.
- Ensure the visual representation of the game board, blocks, and scoreboards is visually pleasing and intuitive.
- Handle the rendering of game elements and updates to the graphical display.

Everyone

- Collaborate on implementing extra features such as block rotations, additional difficulty levels, and special actions.
- Ensure the correct implementation and integration of special actions triggered by specific game events.
- Test and refine the functionality of extra features to enhance the overall gameplay experience.

[Blocks] 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: To accommodate the generation of special blocks that disappear from the board if not cleared within 10 more blocks have fallen, we can introduce an additional attribute isSpecial to the VirtualBlock class. This attribute will help identify whether a block is special or not.

Additionally, we will include a counter attribute in the VirtualBlock class to track the number of drops since the block was generated. (On every drop iteration, keep a list of all blocks currently in the game, increment every block's counter).

The design of the system will include 3 main ideas: a) Triggering Block Clearing Check, b) Clearing Blocks and Shifting Down, c) Handling Boundary Conditions.

a) In the Board class or its relevant functions responsible for updating the board state, implement logic to trigger a check when the block's fallen count reaches the threshold of 10. This check should iterate through all blocks on the board, checking their individual block counters.

b) Call the eraseBlock method when the threshold has been reached and call drop for all blocks still currently present on the board. eraseBlock blocks removes the block from the board, while calling drop on all other blocks entails updating their positions such that they can drop no further.

c) When dropping blocks, ensure that boundary conditions are checked to prevent blocks from moving out of bounds or colliding with other blocks. Adjust block positions accordingly to maintain the integrity of the game board.

Yes, blocks that disappear after a set number of fallen blocks can be confined to more advanced levels. This fits with the usual progression of games, where new features are introduced as players get better. This can easily be achieved by leveraging the Factory Design pattern to customize block generations for different levels. For more advanced levels, configure the given level to include the special disappearing blocks. Making sure the game can grow in level and block types provides high scalability and flexibility, crucial to later developments. A further improvement would be to generate these special blocks depending on how well the user performs. This motivates and incentivizes players to continue playing the game.

[Next Block] Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?
Answer: To accommodate additional difficulty levels in the Biquadris game system with minimal recompilation, we employ the Factory Method design pattern alongside a hierarchy of subclasses for each level. In this design, the Levels class serves as the base, with five subclasses corresponding to levels 0 through 4. Each subclass encapsulates the logic for generating blocks specific to its designated difficulty level. By using polymorphism, the game selects the appropriate block generation logic based on the player's chosen difficulty level. If we want to add more difficulty levels later on, developers can just make a new type of level within the Levels class. They can customize how blocks are generated for this new level: probabilities at which certain blocks are generated, randomness generation of blocks or if the order is read in from sequence input file, etc. This method makes it easy to add additional levels to the game without having to change a lot of code or recompile the whole program.

[Special Actions] Question: 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?

Answer: Employ the Decorator design pattern to manage multiple effects simultaneously in the game. Define a base class that represents the common behaviors, grouping special actions having similar effects (ie.special actions that affect the type of block being generated). Decorators can dynamically add new behavior or modify existing behavior without altering the underlying classes, ensuring flexibility and scalability. By using decorators, effects can be stacked or layered on top of each other, allowing for complex combinations of effects to be applied simultaneously to the game state. Conflicts between effects should be resolved by deciding on an order at which concrete decorators will be applied. In the case where more effects are invented we can create another concrete Decorator class for the new effect and implement its supposed behavior. Overall, the Decorator design pattern provides a clean and modular approach to managing multiple effects simultaneously in the game, enhancing its flexibility and maintainability.

[Command Interpreter] 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. The board should be redrawn, both in text and graphically, each time a command is issued. For the graphic display, redraw as little of the screen (within reason) as is necessary to make the needed changes. For multiplied commands, do not redraw after each repetition; only redraw after the end.

Answer:

Command Registry

Maintain a mapping of command names to their corresponding actions in a map like data structure. This mapping can be dynamically updated during runtime. Implement a command processing system that dynamically looks up commands from the map and executes their associated actions. This allows for additions, modification or removal of commands without recompiling the code.

Command Renaming

Extend the command mapping to support renaming. Allow users to specify the new names for existing commands, updating the map accordingly. This can be achieved by an additional rename command that takes two arguments: the old command name and the new command name.

Macro Language Support

Introduce macro language that allows users to define and execute a sequence of commands under a single name. Macros can be stored in the map along with regular commands, enabling them to be invoked like any other command.

Redrawing of the Board

Ensure the board is redrawn both in text and graphically after each command is executed. Use efficient rendering techniques to update only the necessary parts of the screen and minimize redrawing time. For sequences of commands, such as though with a repetition factor or macro language, redraw the board only after the entire sequence of commands have been executed.

Possible Alias complications

Command aliases introduce additional complexity as users are allowed to input as much of a command as needed to distinguish it from other commands. If we were to keep a map of commands to their prefixes needed to distinguish it from other commands, this would not be able to handle renaming commands during runtime. We would have to construct a DFA where each state represents a single user input character and define transitions to be the next state based on user input. The accepting states would be states where a valid and unambiguous command prefix has been entered. To rename a command, we would have to modify the DFA during runtime by deleting transitions and states associated with the old command and adding transitions and states for the new command.