A5 Plan of Attack (BiQuadris)

Henry Yang, Martin He, Ray Wang

Introduction and Overview

Our implementation of BiQuadris will include three main areas of focus: First, the gameplay engine and rendering system. Second, the levelling system along with block generation and the implementation of storing blocks, cells, and the board. Third, the implementation of validating, auto-completing and accepting commands, which would then interact with our centralized 'game engine' that would be running and managing the overall game and its many subsystems. Not only do we intend to fully implement all of the assignment's required features, but we also intend to include various extra additions to the project, such as an extended "next block" preview with a queue of the generated next blocks, a visual preview of where a block will land if it were to be dropped, as well as a 'hold' feature for blocks. These extra additions will all be toggleable at runtime with the bonusOn, bonusOff commands.

Each of the aforementioned three main areas of focus will be able to be constructed somewhat independently. The implementation of the game commands can be completed fully separately, and tested separately with small, simple test harnesses that only test a particular subsystem. The levelling system, block generation, and storage of blocks, cells and boards, can also be completely separately and tested via a testing harness in order to probe the public classes that the gameplay engine will call. Finally, the gameplay engine and rendering system can be constructed in parallel, but only be tested with the completion of the level system and board implementation. Player commands to the gameplay engine could be tested with a light test harness if the game command classes are not completed yet.

In terms of a concrete order of completion, we plan to finish the levelling system and board implementation first. More specifically, the implementation of the way the board is stored, as well as the Blocks and BlockCells and their methods must be robustly tested first. Once the board is working correctly, then the levelling system can be constructed as well and tested with the board to make sure that the integration of the two functions is working. The board class more or less serves as the backbone of the entire gameplay system, with the gameplay engine and hence the rendering system all relying on it to function. Therefore, it is crucial that the board and any classes the board itself depends on are implemented first and tested to be correct. Although the levelling system and board implementation are to be completed first, as mentioned previously, the other aspects of the program could also be worked on in parallel. Next after the levelling system and board implementation, we would aim to have the game engine and rendering system to be finished next. Parts of the rendering system, mainly the console-based rendering system, could be developed alongside the board to facilitate its testing. This functionality could then be seamlessly copied into the observer model of the actual rendering system. After the board class is completed, the graphics rendering and console-based rendering functionality could each be developed independently, and be tested using the board class. The gameplay engine could also be developed independently, and tested via a light test harness program of various commands, or tested using the game command system if it is

completed. Once the gameplay engine and rendering systems are completed, we would then aim to have the game command system completed, with full features of auto-complete and command validation. However, since the game command system can be developed fully independently from the other systems, this may or may not be already finished by the time the gameplay engine and rendering are finished. The last step would be to link all three main areas of focus and run a full integration test of the game.

What we have mentioned so far is a general overview of how the implementation of the overall Biquadris program, as we have designed it, might proceed. Below, we will detail a proposed allocations of tasks, and our best guess at when they could be completed by:

Allocation and Deadline Breakdown

Subarea: Levelling system and board implementation

Task	**Estimated Completion**	**Deadline**	**Contingency if deadline not met**	**Assignee**
Implement all static block-type classes	11/23	11/24	N/A (Continue until completion)	Martin
Implement Block and Block cell	11/24	11/25	Continue with development until completion, test Level class with a “dummy” Block class (one only with a type of block)	Martin
Implement Level class and all subclasses	11/24	11/25	Continue with development until completion, test Board functionality without random Blocks	Ray
Implement Board and BoardProxy	11/25	11/26	N/A (Continue until completion)	Martin

Subarea: Gameplay engine and rendering

Task	**Estimated Completion**	**Deadline**	**Contingency if deadline not met**	**Assignee**
Implement Observer and ConsoleView	11/24	11/25	Continue with development, test Board and Levels (or game engine) using test harness without console display	Ray
Implement GraphicsView and	11/27	11/28	Continue with development, test game engine and logic	Ray

optimization			using test harness without graphics display	
Implement Subject, BiQuadris, BiQuadrisProxy and all proxy subclasses	11/25	11/28	N/A (Continue until completion)	Henry

Subarea: Game command system

Task	**Estimated Completion**	**Deadline**	**Contingency if deadline not met**	**Assignee**
Implement Individual command classes	11/25	11/29	N/A (Continue until completion)	Henry
Implement CommandTree and all class dependencies	11/28	11/29	N/A (Continue until completion)	Henry

*All documentation and the final design document will be jointly worked on throughout the week

Assignment 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?

With the way we have designed the levelling system, any special gameplay feature, such as allowing a block to disappear from the screen if not cleared before 10 more blocks have fallen can be easily added to the program. Whether this includes modifying an existing level or creating a new level, each concrete level class will have access to the board via the BoardProxy, and can define the specialEvent() for this level to be if 10 or more blocks have fallen without clearing, and define executeSpecialAction() to be making some generated blocks to disappear from the screen. Our system accommodates easy implementation for this rule in a new/modified level by just adding/modifying a derivative of the Level class, without any other changes to the existing interface, as the tools needed to implement this are already provided via our existing Level API. This, and any other possible special feature could be easily confined to a specific level. Thus only concrete level classes that override this method will be able to execute a special event. More details on design methodology are also explained in the next question.

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

The principal way in which we have designed our program to accommodate introducing additional levels into the system with minimum recompilation is by delegating all level functionality to a Level abstract class, that then has concrete Level classes for each individual level. The level class would define block generation as well as a special event/special action rule (nothing by default, level 4 overrides this) for the board. This approach allows the Levels to be loosely coupled from the rest of the program, as the only point of interaction between the levels and the rest of the game is when the Board class calls the generateBlock() method from a Level class instance. Each concrete level class inherits from Level, and levels with randomised block functionality, such as levels 1-4 inherit from the class RandomLevel, which itself inherits from Level. Inside Level, there are methods for specialEvent() and executeSpecialAction(). These methods allow a concrete level instance to specify whether or not it implements a "special event", as well as execute that "special event". We define a special event to be a special rule defined by a level involving some event happening in the board. For example, level 4 will have a special event that consists of dropping a 1x1 block in the centre column every time 5, 10, 15, etc. blocks have been dropped without clearing a row. The default behaviour of speicalEvent() is to return false, and the default behaviour of executeSpecialEvent() is to do nothing, thus there being no special events in a level by default. With this design methodology, any type of level with any unique rule can be easily created simply by creating a new concrete Level class and overriding its specialEvent() method and executeSpecialAction() method. Thus, anytime a new level is added, only that individual concrete Level class needs to be recompiled, along with the main game engine class. The rest of the program can operate normally without any knowledge of the change.

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?

Our program currently already supports the built-in base effects (i.e blind, force, and heavy effects) being toggled simultaneously using methods within the BiQuadris class, (if the assignment allowed more than one effect to be applied after clearing at least two rows). These method calls get passed down to corresponding classes for forward facing functionality. Overall, since these methods can be called independently from each other, this pattern allows our program to freely apply different effects, removing the need for prewriting all possible combinations of the existing 3 effects.

Our program could be designed to support new, additional effects outside of the base effects in a more graceful way by making an abstract base effects class with a pure virtual "activateEffect()" method, that can be derived into different classes for each effect (e.g. HeavyEffect, BlindEffect, etc.) that all define their own activateEffect method. By classifying Effects, we can now store either one or multiple Effects (using a container such as std::set) within the BiQuadris class (or any of its relevant subsystem classes) with the application of polymorphism, which can be implemented to activateEffect(s) whenever it is the appropriate time to apply the effect. This allows our program to more efficiently integrate an arbitrary amount of different effects that can all be applied to an object simultaneously without having to add additional fields for each effect that determines if each effect is applied and without needing to have an else-branch for every possible combination of effects. As

for supporting new commands for the new effects, this is already well-supported in our current system, which will be demonstrated by the responses for the following questions below.

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.)

Our system is designed to accommodate the addition of new command names as well as changes to existing commands through adding/changing an existing derived child class from the defined abstract base class Command<Dependency>, which has a pure virtual function execute(int) (as well as execute(int, vector<string>), which takes in any number of additional parameters), which can be overridden to call method(s) in the Dependency type's class. The implementation of the execute(int) function can be modified to do anything that the Dependency class would allow, which could also be a bad thing if the derived Command child has access to a Dependency class that can do too much. Hence, our system is designed such that the template Command class is further categorized into abstract base classes that specify a certain Proxy class of BiQuadris, which enforces high encapsulation for any command that derives from the category abstract base class. Not only is creating Commands and changing the existing execute implementations of existing Command implementations very easy through this interface, but adding new commands to the collection of existing commands and having them support command auto-complete is also easy through the CommandTree interface, which supports an easy to use addCommand API that adds the command to the command decision tree. With this system, anyone who wishes to create and add on their own Command to the list(s) of possible available commands and having it support auto-complete at the same time can easily do so.

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)?

It would take no effort as our system already supports renaming of existing Commands through the rename command, which uses the CommandTree::remove method then CommandTree::add method under the hood to seamlessly adjust the command decision tree to reflect the rename. This is possible because the rename Command derives from the MetaCommand abstract base class, which derives from a Command<CommandTree *>, where the CommandTree * is the dependency with the intent of the pointer pointing to the current CommandTree. With this, the rename command is able to call the CommandTree::remove and CommandTree::add public methods necessary for a rename of a command within a CommandTree. Renaming a command will not have any effect on preexisting commands.

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 macro language is already supported in our system. Similar to rename, macro also derives from the MetaCommand abstract base class, which would allow for adding of new commands that can also take multiple arguments (using execute(int, vector<string>)). Adding a macro command will also not have any effect on preexisting commands. Our macro command will be used like this:

macro [macro name] {command names separated by spaces (can be auto-completed)}