Biquadris

Calder Lund, Divyam Bhasin, Alex Xu

** See end of document for UML design*

Introduction

Biquadris is a game like Tetris with support for multiple players (we implemented it so it can handle any number of players). A board of biquadris is 15x11, and each player has their own board. A player has as much time as they want to make a move, and their turn won't end until they drop their current block. There are 7 different types of blocks to choose from and 5 different levels to play through. Once a block can't be dropped on a grid, that player is removed from the game as they can no longer make moves. In our implementation, even if a player is eliminated, they can still win if they accumulated the highest score.

Overview

There are three major components of Biquadris: Interpreter, Game, Player. These components do tie the additional classes together to allow for a well-designed game.

Interpreter:

The Interpreter serves as the point of interaction between the user and the game by providing an interface of commands to interact with the game. All printing is handled through the Interpreter as well. The Interpreter class contains methods that determines when certain Game and Player methods get called. This is useful as it makes interacting with each player, in the main.c file, very easy to do. The client does not necessarily need access to all the classes to be able to run this game.

Game:

The Game class handles the logic of running a game, such as printing each Player's Grid, handling the game logic, etc. (it does not do the printing itself however). It also keeps track of whose turn it is at any point in the game, along with which player has the most points. Overall, it is used to combine any number of player into one Game class.

Player:

The Player class represents a single player in a game. It stores information such as the player's score, their current block, their current level, etc. Player contains various objects as fields and methods that can manipulate these fields. For example, is a player was in level 2. Player::levelup() will mutate Player's current_level field (type is Level *) to be one of type Level3, a subclass of Level. Similarly, when a players turn is finished, Interpreter will use Player::endTurn() which manipulates the current_block field to be whatever Level::generatenext() (a pure virtual method) determines.

Then, there are smaller classes that pertain to more specific properties of the game and players, such as **Grid, Cell, Block, SpecAction**, and **Level**.

Grid is a 2D array of **Cell**s which represent the “board” for each player. **Block** pertains to all the different types of blocks and how they will change under translations and rotations. Each Block has 4 distinct Cell pointers in the Grid. **SpecAction** handles all the application of special actions that apply after clearing 2 or more rows. **Level** handles how blocks are generated based on which level the player is at.

Design

There were many design challenges that were encountered when making this project. The objective was to make the design based on strong OOP principles to write expressive code that adapts to change well. Additionally, we strived to follow the Single Responsibility Principle as much as possible to increase code reusability and maintainability.

Impl Idiom:

In an effort to minimize recompilation, we decided on having an Interpreter class utilize the pimpl idiom (Interpreter class explained above). By separating our implementation from the object representation of Interpreter, we can easily facilitate the adding of new commands in the future with minimal recompilation. In the implementation file (InterpreterImpl class), commands can easily be added and removed in only one recompilation. It stores a map comprised of a core command name as the key and a vector as its value. The reason we chose a vector is because it allows for renaming sequences of commands. The map acts as a lookup table for any alias that the user might define and also as a checker for any illegal commands (since any possible command will already be in this table). By having a vector as the value of each key-value pair, we allow for multiple aliases to map to one core command if the sequences defined are different.

Observer Pattern:

We used a variation of the Observer pattern to connect the Player and Game classes. The Player class generates the data, whereas the Game class receives the data and reacts to it. For example, one “tick” of the game represents a turn being executed for each of the players joined to that game. As the player classes manipulate their grids individual, Game observes this and determine which player is winning and which players can still make moves. This implementation of the Observer pattern allows for many dependency between objects without making the objects tightly coupled.

Decorator Pattern:

Since special actions can to be applied cumulatively, we found the Decorator pattern to be highly effective when implementing all the special actions. The Decorator pattern allowed us apply effects on top of each other, so that, for example, we could achieve the heavy associated

with special actions on top of heavy associated with level 3 and 4. Additionally, it allows for special actions to be added to (and removed from) the Player class dynamically at run-time.

Factory Method pattern

We used the factory method pattern to handle generation of blocks. Since blocks are generated with different probabilities depending on the level, we added a 'generatenext()' method to our Level classes that encapsulated the logic behind the generation of blocks. We used a random number generator and computed probabilities based on which number was generated to return a character that would represent the type of Block ('T', 'Z', 'I', etc). Each time a Player dropped their current block, we would use their current level to generate the next block, which would show up at the bottom of their board. In the case that a sequence file was present, we would grab the next block from there instead.

Additional Patterns:

Through research, we were able to come up with other design patterns we utilized that were not taught in this course. For example, for printing, we used the Chain-of-Responsibility pattern. This allowed us to set priorities when displaying a grid. For example, when a user is blind, we use the Blind class implementation to print '?' characters. When they are not blind, it reverts to using Cell::get_type() to choose a character to display.

Another pattern used from external research was the State design pattern for the Level class. The player specifies a current_level and the player's block generation is dependent on the state of their current_level field (ie which level they are in).

Grid Manipulation:

Lastly, a key design idea we used to implement that is worth noting is how we manipulate the grid. There are 2 major components to grid manipulation. One being the movement of a player's current_block and the other being removing a completed row. See Overview to understand how Grid/Block/Cell works.

When a player starts their turn, they can choose various command from the Interpreter. If they choose left, right, down or drop, Player::<command>() is run. Player::<command>() however has a very simple implementation. It calls the Action::<command>(Block *) method of the Action class (which, as we already described, uses the Decorator Pattern). It takes in a Block * for the current_block of the Player. Using the Block ptr, a Normal action will call Block::<command>(), while a Heavy action will call Action::<command>(Block *) followed by Action::down(). To make this clearer, we use an example. A Player at Level 3 wants to move a block right. As we know by the game description, the movement should be one to the right, followed by a downwards movement of one space. So the LevelHeavy decorator will call Action::right(Block *) followed by Action::down(Block *). The reason we call Action::right(Block *) as opposed to Block::right() is because the action could be another Special Effect such as SpecHeavy which would drop the block another 2 spaces (combining for a 3 space drop).

At the end of a players turn, Interpreter calls Player::checkfilled() which returns the number of rows filled. If two or more rows are filled, a player can choose to place a special effect on another player. When Grid::checkfilled() finds a row that is filled, it uses vector's clear

method to remove the existing cells. Cell's destructor is called which determines if the Block that contains this particular Cell should be deleted or not (deletes block if it is last cell in block). Otherwise, the Cell is destructed as normal. The row (a specific vector) in the Grid is removed and a new empty vector of Cell's is added to the top. A new refresh() method is called to update the x and y coordinates of each Cell in the Grid. (see grid.cc line 55 for the code)

Resilience to change

The principal design feature behind our project is the Single Responsibility Pattern, which makes our project very resilient to change. If we want to change or add anything with the way we interact with the game, we know that it should go in the Interpreter since that is Interpreter's only job. Anything related to game logic should go in the Game class, and anything related to Block movements should go in the appropriate block classes. We have well-defined places for adding certain enhancements to our program, which avoids confusion and gives us a ways of changing things relatively quickly.

By using classes and inheritance extensively, we can facilitate the addition of additional levels and blocks easily. For each new Level or Block, we would just need to override the appropriate methods and the rest of our codebase would integrate perfectly well with these new additions. By using the Decorator pattern for our special actions, we have provided the ability to add special actions at runtime with minimum recompilation. Moreover, the effects are cumulative, which allows us to specify actions that work on top of other actions.

If we needed to add additional commands, we could do it by just modifying the implementation of our Interpreter class. We could specify all the actions that we want this new command to perform, and also be sure that it works in the context of our 'macro' language that supports renaming of sequences of commands. We would achieve minimum recompilation (due to the pimpl idiom) and all our core Interpreter features would still function properly.

If we ever needed to add additional Blocks, we could just create a specific block class, have it inherit from Block, and override the relevant methods for defining how translations and rotations should work. If we needed to add any Player specific information, we could define it in the Player class, and it would extend to our multiplayer feature.

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:
We would add a field (ex: ‘count’) in our Block class that would be unique to each block. It would represent the number of blocks that have fallen since that block was dropped. Each time a block is dropped, we would go through each cell of our grid, query the block it belongs to for the count, and decide whether to remove all cells of that block from the grid or not. This can also be easily confined to higher levels, as before updating the count of a block, we would query the block’s ‘spawnlevel’ that would represent which level the associated player was at when they dropped the block. If the spawn level is below a certain threshold, we would not update the count of the block at all.

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

Answer:
We implemented a level hierarchy that keeps all levels separate from each other. We also used virtual methods and “is-a” relationships to minimize recompilation when adding another level. In the future, for adding another level, we would just have to create another class and include it in the Player class, without recompiling any other classes, since our Player class is the only class that has to manage levels extensively. By overriding the ‘generatenext()’ method, we could specify alternative schemes of generating blocks, since Level uses the Factory Method pattern.

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:
To have multiple effects applied simultaneously, the Decorator pattern would be highly useful. It allows us to add new effects at runtime, without having to modify any of the other classes. This limits the amount of recompilation and provides us with a scalable way of interesting g new effects. With the Decorator pattern, we don’t need a central point of control with if-else statements to govern what happens. Inheritance and polymorphism can help us reduce the code we have to write and make things more understandable. The Decorator pattern helped us considerably with the cumulative effect that the special action heavy has on the level heavy (in level 4). Also, we were able to control what each special action does more tightly with the Decorator pattern. For example: special heavy should not apply heavy on rotations, but level heavy should. By giving them their own classes, we were able to specify this behaviour easily.

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?

Answer:
For the addition of new commands or changes to existing commands, we could use the pimpl idiom. It would allow us to add new commands easily, without much recompilation. We could add support for the use of a “macro” language by having a map comprised of String keys and Vector<String> values. Whenever a command is called that is not one of the default ones, we would go through our Vector for each key and look for this command. The reason why it would be ideal to have Vector<String> as the value is because it would allow us to specify multiple aliases for the same command if it appears in different sequences of commands. For example:

rename clockwise right down crd vs. rename counterclockwise right left crl

Both ‘crd’ and ‘crl’ should invoke the right command, so by having a the value of ‘right’ in our lookup table be a Vector<String>, we can have ‘crd’ and ‘crl’ both be able to call right, as they should. A future adjustment we could make to maximize efficiency is to convert our lookup table into a hash table.

Extra credit features

For extra credit, one of our main features is the **support for more than 2 players**. The number of players participating in the game can be specified as a command line argument ‘-numplayers’, and our game is able to handle an unlimited number of players playing at the same time (subject to memory constraints). When a player gets out, the game doesn’t end until there’s only one player left. A player can still win the game if their high score is left unsurpassed until the end of the game.

We also added **support for a ‘macro’ language** in our Interpreter. The player can rename entire sequences of commands and combine them with multipliers as well. Commands can have multiple aliases if they are part of different sequences, and players can combine sequences of commands however they want.

Additionally, we designed a special action called **Lock**, which prevents a player from performing any rotations on the current block. This special action is not one that a player can call when completing two rows; instead it occurs in our other bonus feature, **Level5**.

We created an extra level (level 5), which has all the effects of Level4 plus the Lock special action applied, by default.

We used these extra features to show how simple creating an additional levels and effects can be. In this case, we only had to recompile player.cc to make the changes.

Final questions

1. What lessons did this project teach you about developing software in teams?

This project has taught us about prioritization, delegation, and teamwork. We prioritized what we needed to work on based on how important those features were to the game. We delegated tasks so that each person was responsible for handling a certain portion of the project, and the UML made it easy to plan this out. Finally, we helped each other out with any errors and worked collaboratively to bounce ideas off each other instead of just focusing on our own tasks.

In terms of writing large programs, we learned the importance of having a UML diagram before we start coding. Our UML diagram made it very easy to visualize our project and figure out how things fit together. It helped us split work and delegate tasks so each person knew exactly which portion of the project they were working on. We also learned the importance of employing good design patterns and following the Single Responsibility Principle. The importance of building a program that is resilient to change is emphasized when such a large project has to be maintained. It would be tedious to go and change unrelated parts of our codebase everytime we wanted to add a new feature in a large codebase. By using effective design patterns and the Single Responsibility Principle, we were able to make our project resilient to change and learn more about effective software design.

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

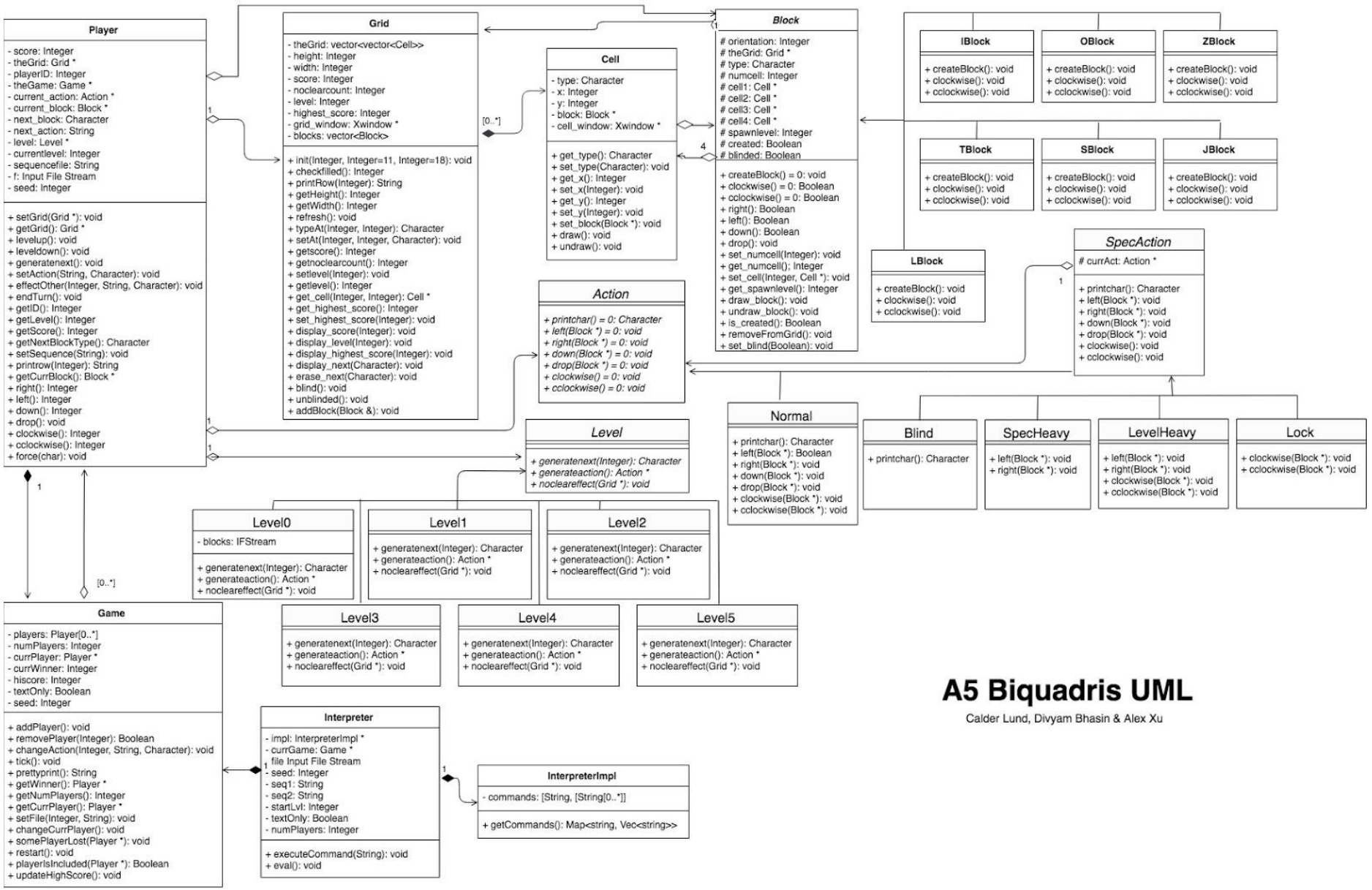
If we had the chance to start over, we would consider changing our design to allow for minimal direct manipulation of an object outside its class, and give more careful consideration to the access level of our methods (whether they should be public, private, or protected). Both of those things should increase the security of our codebase and prevent it from bad actors in the future. Since this was just a project that we were going to work on, we left those things out. But, in the real world, the codebase we work on will probably be modified by future programmers as well, so we want to ensure proper access.

We would also manage our time more effectively to allow us to add more extra features. We focused mainly on getting the base project done, and while that is important, we would also allow ourselves more time to finish some more extra features by doing the base project sooner. This would also allow us plenty of time to test and refine our base features. Another thing is that we did not make sure that our program compiled from the start. About ¾ of the way through our project, we had a lot of errors since we hadn't been compiling and testing continuously. This caused us some headache, but after that point, we always made sure to compile and verify everything works. If we were to do the project again, making sure that the project compiles from the very start would be our utmost priority. To guarantee that we always have a running program, we would also make use of more version control features such as branching. Our

runnable program would then be on ‘master’, for example, and all our extra features would be on separate branches. After testing, we would just merge these branches with master, and guarantee a working program at all stages of the development process.

Conclusion

By completing this project, we have learned a lot about the importance of good software design. We also gained valuable experience in creating projects using Git. We found that a well thought out plan, created before the coding portion, can make programming much more efficient. Our UML was a great tool to help us visualize the different components of our project. The design patterns we learned in class and researched helped us solve problems in an elegant and scalable way, making our project more resilient to change. In conclusion, while coding Biquadris helped us solve logic problems and refine our debugging skills, the biggest takeaways have been all the design steps and how good design can make coding large projects so much easier.



A5 Biquadris UML

Calder Lund, Divyam Bhasin & Alex Xu