Plan of Attack

Authored by r3kaushi

Team Members:

Navya Mehta	njmehta
Rohit Kaushik	r3kaushi
Sahil Pahooja	spahooja

As part of assignment 5 and the final group project in CS246, we chose to implement biquadris – an alternate version of tetris.

In order to implement the game, we are following the plan below with the corresponding estimated completion dates, to be finished by the name of the partner that appears in **bold** after the description of each step.

Initial Game Setup

At first, the three of us read the guidelines and the game descriptions, choosing biquadris out of mutual interest and curiosity.
njmehta, r3kaushi, spahooja

Next, we thoroughly debated the dynamics of the game that would produce the best and most efficient gameplay while using least memory and being the fastest. We considered different implementations to ensure that we include all relevant concepts taught during lectures like – low coupling, high cohesion, single responsibility principle and the different design patterns. We came up with a satisfiable system architecture which uses multiple OOP concepts like polymorphism, inheritance, encapsulation, and which successfully inculcates the visitor pattern as well.
njmehta, r3kaushi, spahooja

Next, we created a draft of the UML so we could start writing the header files and implementing classes and methods. The UML was later completed as the project progressed.
r3kaushi

The implementation of the game itself began with the creation of the Grid class which would be the core of the game. Each Grid was defined to contain multiple cells, and the pieces were to be placed in cells. We started by implementing the Grid, Piece and Cell classes. We also developed a “level” class to encapsulate the different levels of the game within themselves as a unit.
njmehta

The deadline for all the above tasks was set to be ***November 18th.***

Piece Implementation

Once the Grid, Piece, Cell and Level classes were made, we began creating the pieces of the board. We divided writing classes for each of the 7 pieces amongst the 3 of us – with a 2, 2, 3 split.

Pieces O, I, T – **njmehta**
Pieces Z, S – **r3kaushi**
Pieces J, L – **spahooja**

The deadline for implementing the pieces was ***November 20th.***

Autocommand

We then implemented the autocommand feature where commands manipulate block positions and typing part of a command completes the command itself and runs it.
njmehta

Next, we created the sequence[1-2].txt files from which Level0 objects will read input from to create upcoming blocks for players 1 and 2.
r3kaushi

Level 0 and Testing

After fully implementing Level0, we worked on random bugs that occurred during our test trials for the game. Before we began implementing the program, we decided to employ test driven development in order to ensure a more robust application and minimal bugs. As part of this process, we came up with multiple test cases that we would specifically test for beforehand. Once we were satisfied with Level0, we referred to the test cases we had come up with and tested the program. We ran into some errors relating to the rotation of pieces (pieces J and L in particular), which were fixed right away.
spahooja and njmehta

The deadline for the testing process and completion of Level0 was ***November 21st****.*

Level1 and Testing

Once we finished Level0 (and ensured we have a working product), we employed the Decorator Pattern to add further functionality to our game, i.e, further levels 1, 2, 3 and 4 – completing a 5 level hierarchy for the game.

We began working on the implementation of level1 which includes a more complicated algorithm for the generation of the next block that will appear on screen for the players to manipulate and drop. This algorithm involves the generation of random numbers from 1-12, which include a two-number frequency for the appearance of every block except Z and S which appear less frequently.
njmehta

Then, we followed a similar process for testing level1 with our predetermined test cases – the same ones we came up with initially would still be relevant for higher levels, so we used them.
spahooja

*The deadline for completion of Level1 was **November 22nd.***

Level2 and Testing

Next, we worked on the implementation of Level2, which was a modification of Level1. In Level2, the generation of pieces is completely random and is not skewed in any way. We generate numbers between 1 and 7 inclusive and each piece is associated with a number. To implement this, we used and STL map which associated numbers to each of the characters that characterise blocks. (Eg. J – 1, O – 2, …).
njmehta and r3kaushi

We then worked on the testing to ensure the complete functionality of Level2.
spahooja

*The deadline for completion of Level2 was **November 23rd**.*

Level3 and Testing

The implementation of Level3 involved the generation of pieces, skewed once again, with Z and T occurring more frequently than the others. This was done in order to create higher difficulty. We used STL maps for this as well and associated multiple keys to the same value.
njmehta

We completed the testing for this level after it's implementation.
r3kaushi

*The deadline for the completion of Level3 was **November 24th.***

Level4 and Testing

Level4 involves more complicated features apart from skewed generation of pieces. We implemented a feature wherein if a row is not cleared in 5 moves, the game generates blocks of only one kind. This would make it easier for players to clear rows if they are unable to otherwise do so. If they then succeed in clearing rows in fewer moves, the control switches back to a previous level to continue the game. The implementation was then succeeded by testing it's implementation.
njmehta, r3kaushi and spahooja

*The deadline for the completion of Level4 is **November 26th.***

Special Actions

1. ***Blind***
   When a player clears >= 2 rows in one move, and the user enters blind, the character field for all corresponding rows and columns would remain as they were, and during display, a conditional block would render '?' on required rows and columns.
2. ***Heavy***
   After every input acceptance for a move, the 'down' pointer for the corresponding piece would be incremented by 2 so that the piece moves down by 2. There would also be the routine checks in place to see whether the cells below are occupied; if they are, the game ends.
3. ***Force***
   We would use one player's input to render the required block on the other player's screen.

njmehta, r3kaushi and spahooja

The deadline for implementing special features will be ***November 27th.***

Graphic Display

While creating our project plan, based on our familiarity with the graphic display implementation, we decided to implement the graphic features last. This was meant to be the final addition to our game which would give it the appeal that simple CLI wouldn't. Once we finish implementing all the special features and ensure that they work consistently without any bugs.
njmehta, r3kaushi and spahooja

The deadline for this implementation will be ***November 29th.***

Questions

1. To remove blocks that have been on the screen for longer than the time that is taken for 10 blocks to have fallen, we can introduce a field in the Piece superclass that counts the number of blocks that have fallen while the current block is on the screen. Once this counter hits 10, we can use a loop to go through the vector of cells that contain the piece and set their values to null (or initial value), which effectively clears the block.
2. To have minimum recompilation, we can use forward declarations instead of using *#include* in certain header files. For example, since Levels don't directly depend on Cells or Pieces, but rather they are facets of Levels, we can forward declare the Cell and Piece classes to reduce compilation times. To add more levels with minimal

recompilation, the level classes can also be forward declared since the only difference in the levels is the algorithm for generation of pieces.

3. Having multiple special effects run simultaneously would not be too difficult considering the architecture we have decided to go with. Special effects like blind, heavy and force can be decorator classes independent from the core game data structures, which simply manipulate display and Pieces depending on the effect. Again, using a field that counts the number of rows cleared, we can add a conditional block that runs only if >=2 rows are cleared; this block would contain the commands for the effects. Every time >=2 rows clear, this block will be run independent from the previous run, and effects will be added on independently. The original state of the game will be maintained by the Grid, Cell and Piece classes, so that can be used to decide how blocks drop and if the opponent loses the game when the board has '?' symbols or if the block cannot be placed in the initial position. Here, there will be no else-block at all; simply if conditions that run if an effect is given as input. Other effects can also be added on and would work the same way.
4. Adding more commands to the game would constitute adding conditional blocks to the command accepter. Changing command names would again constitute conditional blocks which run when a Boolean field is set to true once "rename" is read from input. Further blocks would run depending on which of the renamed commands are requested by the user. The commands can be stored as a map whose value would be the original command. To implement a macro language, the sequence of commands would be placed in a vector corresponding to every command; on running, the entire vector would be traversed and commands would be run in order.