Plan of Attack

Breakdown of Tasks

Administration

- Progress Checker – Keeps track of current progress in the project. Creates documentation that records the work done by other members.
- Post-Document Creator – Meets with the group near the end of the project and creates the post-documentation.

Development

- Testing – In charge of make test cases and performing regression tests regularly throughout the development process. Includes creating test-specific methods/functions.
- Code Structure - Creating the skeleton class structure of the program making sure all classes have the proper relationships.
- Class Implementation – Adds functionality to the classes and methods.
  - Block Maker – Designing and implementing the class structure of the Block classes in accordance to the UML
  - Board Architect – Designing and implementing the class structure of the Board class in accordance to the UML
  - Game Master – Designing and implementing the class structure of the Game class in accordance to the UML. As well as creating the main function and command interpreter.
- Wishlist – Create extra implementations for extra credit. NOTE: All the above classes MUST be done before we tackle these.

Delegation and Tentative Schedule

Testing:

Assigned: George Cheng

Time Required: All throughout

Estimated Completion: Dec. 4 2016

Block:

Assigned: Daniel Morales

Structure Design

Time Required: 4 hours

Implementation

Time Required: 8 hours

Estimated Completion: Nov. 28 2016

Board:

Assigned: Ryan Phillip

Structure Design

Time Required: 3-5 hours

Implementaton

Time Required: 6-10 hours

Graphics (Very last thing we do)

Time Required: 6-8 hours

Estimated Completion: Nov. 28-29 2016 (Before Graphics) Dec. 2 2016 (W/ Graphics)

Game

Assigned: George Cheng

Structure Design

Time Required: 4 hours

Implementaton

Time Required: 4 hours

Estimated Completion: Nov. 30 2016 (Before Graphics) Dec. 2 2016 (W/ Graphics)

Extra Functionality/ Final Testing:

Assigned: Everyone

Estimated Completion: Dec 4 2016

Questions & Answers

Q1: **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?**

The way we would design our system is to create a new variable within the SingleBlock class to check how long a block has existed for. By default this value will be -1, however if the difficulty is sufficiently high, there is a change that this value will be set to 0 instead and begin to increment. Once the counter hits 10, the block will be removed accordingly. This way, we can confine this method to be confined to advanced levels. The number will be incremented after every block is placed.

Q2: **How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?**

Adding more levels and modifications of the behavior dependent on difficulty is confined only to the game class. Difficulty is stored as an integer which means it's easy to accommodate extra levels because all we need to do is create a new case for a higher difficulty as needed. Since it's only confined to one class minimum recompilation is needed.

Q3: **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.**

New command names can be handled by creating another case to the command interpreter. The command interpreter will be structured in a way that takes in standard input and reads the command which maps it to a certain function call using a selection statement. Changes to existing command names can be handled by checking string values rather than literals. We could assign a string value to each command that can be changed by another command, this would also allow us to rename functions by changing the string values. The way we could support a macro language is by creating a macro operation that takes in the name of a macro followed by an arbitrary amount of commands then storing such macros in a macro vector. Once this macro is called, it would execute all the commands stored inside the macro.