Design Document for Quadris Project (Assignment 5)

By: Alvin Tran (a2tran) and Guoyao Feng (g2feng)

1. Design Approaches

1.1. To Blocks

To implement the seven different blocks into the project, it was decided that all blocks would derive from a single "Block" class. This was decided upon because it was seen that all the different blocks had similar capabilities, such as shifting left and right, falling, rotating, and shifting down. The fact that the program involves the user changing between different block types to control, it was deemed suitable to derive them all from the same class. This is because it would then be possible to simply create a pointer to a Block in the code and assign it as any of its child classes, which mirrors the game's requirement to give the user control of any of the block types.

The primary difference between these blocks was how the cells (the squares used to compose the blocks) for each of them are arranged. As such, the primary purpose of each child class of Block was be to define the cell arrangements for each angle the block is rotated at. These arrangements were assigned statically for each block type in order to prevent the notion of recreating such arrangements that should be associated with one block type. However, in order to avoid the event of recreating code for the rotation of each block type (since the code would simply involve selecting which arrangement from its arrangement set to use), it was decided that the parent Block class would also store an empty arrangement of cells for each rotation angle. By doing this, it would then be possible to define the rotation methods in the Block class instead of each of the child classes, as the Block class would then have access to a set of cell arrangements for each rotation angle. It would be job of the initializer in each block type to populate the cell arrangement set of their parent class using the static list assigned to each of them.

A Block must deal with a group of cells, along with a specific Board. Therefore, to identify cells and which board the Block is associated with, the Block class is given pointers to a list of cells and a Board object to refer to. Furthermore, to help with displaying the Block object, the class was also given fields for a symbol (character used to represent the Block on the textual display), and a colour.

1.2. To Board

Conceptually, the board is composed of a grid of pointers to individual cells. It is responsible for creation, destruction and the retrieval of inquired cell given the coordinate. In addition, the board handles such operation such as row reduction for calculating the new score.

At the beginning, the board is supposed to handle all operations on the individual cells. Efforts have been made to design the board to make it she only access point to the cell. However, this approach will does not provide sufficient support to the block objects when it comes to complex operations performed by the block objects. As a result, we decided to treat the block, board and cells as a coherent module. This will also reduce the overhead incurred from designing the interface of board solely to accommodate specific requirement from the blocks.

1.3. To Display

The graphical display is designed to be a thin layer(adapter) on top of the Xwindow. Most importantly, the graphical display encapsulates the relative scale of all objects on the Xwindow. Every

board is assigned a reference to the graphical display object so that there is no need to query to whole board to update the user interface. Also, it potentially eliminates the overhead of unnecessary update on the user interface.

1.4. To Next Block Generation

Based on the specifications for this project, it would be the level of the game that would decide what blocks to generate, based on a pseudo-random number generator. Thus, in the code, it was decided that Level classes would be created to handle the generation of blocks in the game. These Level objects would be used by the Game object used to run the game to generate an object of one of the Block types (determined by the Block generation method in the Level) for use in the game.

As with the Block class and its child classes, the different levels of this program would be derived from a single Level class. This is because all levels share many of the same properties, such as the ability to generate Blocks and draw itself textually, but the way in which they decide what type of Block to generate differs. Thus, the parent Level class declares a pure virtual "getNextBlock" method, which forces any child class of the object to define one. This ensures that when someone attempts to create their own level for this game, that level will be guaranteed to have a new method of Block generation defined.

For the most part, the way in which each level generates its Blocks are quite similar. The levels beyond level 0 use a PRNG object to pseudo-randomly get a number with a given seed (which may be provided by the user, as discussed in section 1.7). That number then has a modulus operation with a specific number for each Level (call it x) applied on it to get a number between 0 and x-1. The numbers between 0 and x-1 (call this number y) would map to the generation of specific Block types for each Level. What differs from level to level is how many y values map to which Block type generation. This would effectively simulate the chances of generating specific Blocks in specific levels. For example, in level 2, all blocks are given an equal chance to be generated, and thus, y is a number between 0 and 6, and each single number maps to a single Block type generation.

Block generation for Level 0 is simply done by extracting a block sequence from a sequence.txt file, and basing which Block to generate on where in that sequence the Level is at. To keep track of where the Level is at in the sequence, a static integer field is given to this class, which increment every time a Block is generated. When this number exceeds the size of the sequence, the number simply reverts back to 0.

1.5. To Command Interpreter

The command interpreter is designed to handle the reading, parsing and managing commands. The command interpreter reads input from the standard input and then parses the command according to the format specified. Incorrect inputs are rejected. It consists of a mapping from the command name to the actual command object, which enables developers to expand the existing set of command and support command renaming.

1.6. To Scoring

The Score class is fairly simple and straightforward. A further improvement might be that the

logic for calculating score can be encapsulated inside the Score class.

1.7. To the Command Line Interface

There exists the possibility that the user may want to add extra options to run the program. As such, to handle the ability to read in extra options passed into the program from the command line, the main method of the program in the "main.cc" file is of the type that accepts an array of constant character pointers. Doing this enabled the ability to look through the different options passed into the program, if desired.

The specification for this project requested the ability to accept the options, "-text" and "-seed xxx". Thus, in the main method, a for loop was placed to look through the array of character pointers passed to the program. These character pointers would then be compared with the string "-text" or "-seed" to see if they are to be set. If so, then for the "-text" option, a boolean variable indicating the option for text-only mode in the program would be set to true (it's default value is false), while for "-seed," the subsequent string would be parsed as the integer value representing the seed. Extra checks were also implemented for the "-seed" option to ensure the option right after it is actually an integer value, and that there even exists an option after it. If an invalid or non-existent value exists after "-seed," it was decided that the program would exit, since it was deemed unreasonable to try and run the program with a default seed after passing any kind of seed value into the program. It would also prevent the scenario where the user mistypes an integer, getting an invalid number, but assumes the number was accepted because the program ran perfectly, while it would be likely that the program would be running with a default seed, instead.

To use these values in the game, itself, the Application class holds fields that hold them, allowing it to pass them to other components in the game. Thus, the constructor of an Application object requires passing in these values. The primary areas where Application may use the seed would be the Level objects, which use the seed in their pseudo-random number generator to produce the next blocks. For text-only mode, meanwhile, the Application class primarily uses it to determine whether or not to create GDisplay objects, which are used to display the game on a window. Checks are made throughout the program for the existence of such GDisplay objects so that when they are NULL, no calls to draw on them would be made.

2. Ways the Design Differed from the UML

For the most part, the design of the project has remained consistent with the UML diagram proposed with it. Much of the dependencies depicted on it have remained the same, such as the notion of all game logic-related portions of this project being derived from a single "Component" class, along with the use of an "Application" to run the game.

The differences mainly exist in the methods associated with each class, as some have had theirs promoted to the parent class. This is most evident in the Block and its child classes. There, the original UML had each Block Type with its own rotation method. The current design has placed these methods in the parent Block class, since it was found that rotation was found to simply involve changing a single value (to indicate the rotation state) and getting the cell arrangement associated with that value. Along with this, the enumeration, "RotateState," was eliminated from the project, and instead, was replaced by an integer value to represent the index of an array (which was used to store the cell arrangements). The initialize method used in the current design was used to help accommodate this change in design.

The way the Level classes were designed slightly differs, too, as the old UML design had only 3 classes, used to represent Levels 1 to 3 (the fact that all three were Level1 in the UML was a typo). For the current design, Level0 also exists as a class with its own field to host the sequence it uses to produce Blocks by. Levels 1 to 3, however, remain fairly consistent with their original UML designs. For Level, though, new fields were added to let Levels 1 to 3 function properly, such as the introduction of a static

PRNG to help those levels produce Blocks pseudo-randomly, and a static bool indicating if this PRNG has already been set. The latter is to preven the PRNG from being reseeded during the game, as that would cause the game to produce the same sequence every time a level is entered (which is undesirable, due to not being random enough), rather than every time the game is entered (which is tolerably random).

3. Extra Features

An extra executable "walllessquadris" is available inside the package which has feature that blocks can transport through the left and right boarders and show up on the other side of the board. This feature supports such operations as LeftShift, RightShift, Clockwise and Counterclockwise. The source code for this feature is under the directory "project_feature/project".
Following the same procedure, one could produce the executable using the makefile inside the directory mentioned above.

4. Answers to Assignment Questions

Question: How could you design your system to make sure that only these seven kinds of blocks can be generated, and in particular, that non-standard configurations (where, for example, the four pieces are not adjacent) cannot be produced?

Answer: To ensure that only these seven kinds of blocks can be generated, it would be preferable to defined each kind of block as its own class, perhaps deriving from one single Block class to make use of their similar capabilities. This way, the game can only ever choose the arrangements from these block only, since the game would then need to use a Block to operate, and only these seven types of blocks exist for use. This also helps to ensure non-standard configurations cannot be produced, as each block is given a specific arrangement of cells according to what the programmer decides the arrangements to be, and thus, the game can never deviate from these configurations.

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

Answer: Assuming that the new level will only involve the relative probability of getting specified blocks, we made an interface which decouples the generation of random blocks from the game module, which simply provides a getNextBlock function. When a new level is introduced to the system, the game.cc file will be updated to include the new command. The game.cc and application.cc will be recompiled in this case.

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)?

Answer: Every concrete command inherits from the Command class. In each command, the actual implementation is propagated to the game module while the command needs only to implement the steps. When a new command is introduced to the existing system, the first step is to update the application.cc file to include the new command. Since all the commands are all generated and pushed into the command list inside the command interpreter. Only the application.cc and the *newcommandname*.cc will be

compiled (if the new command requires operations not supported by the existing design of the game, it would be necessary to incorporate the new functionality and recompile the game module).

As mentioned above, renaming the name of an existing command can be handled by the command interpreter which maintains a map from command names to the pointer to command objects. It is flexible to update the name of a command.

It is believed that the design above make it easy to adapt the system to support a command as well as command renaming.