Quadris Design Document

By Matthew Correia and Rebecca Thomson

UML Report

In order to show how our program had changed from our original outline, we have created an updated, full UML diagram. Here is an explanation for the changes we made: (note, for our final UML, please see UMLfinal.pdf in this folder)

Deviations from the original plan:

- Absence of a "Score" class

Scoring was one of the very last parts we tackled in this assignment. After some thorough thinking, we discovered scoring was not needed to be a full class, and instead is made up of a few different functions. We had originally planned to make it into a class because we needed a way to track the different blocks that were cleared, but condensed these into introducing an "ID" format for blocks (more explanation of this in the "Scoring" section of the report) and a few formulas.

- Introduction of several new "get" functions, in both Tetrimino and GameGrid classes

As we were coding, we noticed some difficulties with functions grabbing variables from our superclasses. C++ complained a lot because the variables were set privately. So, to fix this while we were coding, we had made all of the variables public which worked... but, we knew from paying attention in lecture this would not be a good practice to keep. So, our work-around is to create these "get" functions, which are member functions, to allow our functions grab the needed values. Problem solved.

- Losing "GetPiece" and "UpdatePosition" Tetrimino functions

Originally we had planned to put GetPiece (which was meant to grab the next block) and "UpdatePosition" (meant to give the new position of the block after ortation) in as Tetrimino member functions. GetPiece is now served by the function NextBlock, and through the way rotate works, it also updates the new position of the block after rotation (as rotate uses co-ordinates to calculate the rotation), and so these two functions were no longer needed.

Agreements with the original plan:

- The "Block" subclasses

We still kept most of the block information in the Tetrimino superclass, and have each block type as subclasses of Tetrimino. This was the most efficient and easy-to-code way to keep each Block type the "same" but "different".

- The "rotate" function

"Rotate" is still kept as Tetrimino and GameGrid member functions. The current functionality of rotate matches our original vision, and actually does more for us than we expected it to.

- The "move" functions

"movement" remained as originally planned. The current functionality of movement matches our original vision, with the only additions being that collision is also checked within the movement functions.

Attacking the Assignment

Both of us working on the assignment tend to be organized, hard working, and we each both contribute different areas of expertise to programming. After laying out the tasks that would need to be completed, Matt, being more comfortable with the nitty-gritty of game programming, tackled the GameGrid class and related functions and Rebecca, being more adept at things requiring visual interpretation, took on handling the Tetrimino class and related functions and subclasses. Both of us were dreading calculating the scoring, and so we agreed to work on that together when we were finished our separate parts.

If you look through our code, you will probably notice these two classes were coded by totally different people. Luckily, meshing our pieces of code together after we were finished wasn't too bad. A real challenge came with making our rotate function compatible with the game grid, and aspects of both the function and the class needed to be changed. After that bump, the rest of it was relatively smooth sailing.

Calculating the scoring didn't turn out to be as bad as we thought, and due to some quick thinkng, we completed that quite quickly (Matt had calculated what was necessary from row clearing while he coded his part previously).

Although we had finished everything required, we initially had several other ideas for extra functionality. We didn't end up coding our program as quickly as we would have hoped, and so these extra features were not implemented.

Extra Features

In our original planning we had hoped to include some extra features to out assignment, but due to time constrictions we were unable to do so. Some of the features we had hoped to implement included

custom graphic images that would be used to represent the blocks, keyboard input and the rename function explained in one of the assignment questions.

Aspects of our Program

BLOCKS

To implement blocks, we first have a superclass called Tetrimino, followed by seven subclasses, one for each block type (e.g. "I", "J", "S" etc.). Since all of the blocks would need the same parameters, all variables and member functions are stored in the Tetrimino class. Each individual variable is carefully initialized in the respective Block subclass constructor.

The whole block system works by having first a small two-dimensional vector which contains the shape of the block, and relating that to the larger grid which holds the whole game. The smaller grid, in order to be compatible with the rotate function, is only just as big as is required for that block type. For example, an I-Block would have initial dimensions of height "1" and width "4", a default T-block would have a height of "2" and a width of "3". Two different arrays (of type pair) holding co-ordinates are maintained, one for keeping track of the values in smaller grid, and one for keeping track of the values in relation to the larger, GameGrid.

Rotate works with a simple equation, which switches the dimensions of the small grid (width become height and height become width) and then changes the values within that grid (a letter or NULL) depending on the desired orientation (clock-wise or counter-clock-wise). To make this switch, there has been a liberal use of "temporary" integers and arrays. The co-ordinates relating to the small grid are then converted for use with the large GameGrid, after all rotating has taken place.

The conversion step in rotate needs to happen as the point (0,0) on the smaller grid does not directly translate to (0,0) on the GameGrid (the point becomes more like (3,0)) and is converted by simple addition to the Y co-ordinate. Looking through the rotate function, you may notice there are a few seemingly strange if cases in the conversion step. During execution and testing, we noticed some of the blocks didn't follow the expected pattern during some points in rotation, this is also fixed with a bit of simple addition.

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?

Above I mentioned that each individual variable in the Block subclass constructor is carefully initialized. Each grid for each Block is hard-coded to follow the certain pattern of that one particular block. The vectors containing co-ordinate points are also hard-coded in the constructor. Since there is a finite(and small) number of blocks, we felt this was the most practical choice, instead of trying to derive one, or

several, complicated algorithms to initialize the shape of the Block. Since all of this is hard-coded, and has been tested appropriately, there is no way an invalid shape can be produced by this program.

NEXT BLOCK

The purpose of the NextBlock function is to determine and create the next block for gameplay and give this to the GameGrid class for use. The entire process consists of six different functions.

First the NextBlock function is called with the current level and a file stream. NextBlock then calls the appropriate function for determining the next block, depending on what level has been given. That function (Next_zero, Next_one, Next_two, or Next_three) then determines the next block to be initialized, and passes this decision as a string to the Fetch_Tetrimino function. Fetch_Tetrimino consists of several "if" clauses, one for each Block type. Fetch_Tetrimino creates the desired Block, and then a pointer to the Block is passed backwards until it reaches NextBlock, which then passes it to the GameGrid.

If the level is 0, NextBlock calls on Next_zero with the given file stream. Next_zero then takes the next string from the file stream and gives it to Fetch_Tetrimino. If the file is empty, Next_zero will re-open the file and start the stream over again.

Levels 1, 2 and 3 are programmed similarly to each other. The next Block during these levels are all based on probability, so we created an array in accordance with the desired probabilities for each level. For example, level 2 is to give each block equal probability, so an array of length 7 is used, with each block taking up one position. We used the rand() function (and the seed set as system time) to determine a random position in the array. This value is then passed to Fetch_Tetrimino.

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

Assuming new levels would not introduce new types of blocks, I feel our system is well-equipped for handling the introduction of new levels. Everything to deal with determining the next block for play is contained in this one file. New clauses would have to be added to NextBlock, and a new function created to create the desired behaviour for that particular level (with possibly some changes made in the scoring-related functions), but otherwise all other functions in gameplay are not based on the level at all, and should function properly regardless.

COMMAND INTERPRETER

The command interpreter is the main part of the program which deals with player input and prepares the next block in the grid every time a block is dropped. This program also checks if the game is over and updates the display after every move.

The player input is put through a parsing system which is able to autocomplete a given command if it can associate it with a valid command for the game. This system works very similar to the trie structure that was used in Assignment 3. To autocomplete a given input the program searches through a trie-type structure and determines if the given input corresponds to a valid command. The main functions for this process are split between initialization and searching. The Initialization functions set up the tree by giving it the possible commands for the game. The search functions search the tree and determine if the player has given a valid command, if so the command is returned.

Once the command interpreter knows it has a command it will execute the corresponding function. If a multiplier was supplied with the command the command will be executed as many times as the multiplier.

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)?

To accommodate for new commands there would not be many changes to the source code other than the new set of functions associated with the given command. To add a new command that name will be added to my trie-type structure using the insert function, and an associated if statement will be needed within my main command interpreter loop to call the functions associated with the command. If changes were to be made to existing names then all that will be needed is a delete function that can remove a command from the trie-type structure. To rename an existing command a new function would have to be set up to take in an old command name, delete it from my trie-type structure and then use the insert function to add the new command name. Overall it would not be very difficult to add a renaming function to my command interpreter.

GRID

The Grid Class is a crucial part of the overall game and keeps track of many variables needed by various functions. The member functions of the grid class include movement, collision, graphic and text display, as well as a game over check. The fields of the grid class can be broken up into the game board, flags, level and scores, command-line arguments, graphics, x/y coordinates for keeping track of the blocks and an id number assigned to each block as they drop.

Most of these variables are stored in the grid class because of how much the other programs in this project depend on grid. This dependency resulted in member functions needing to be created that helped set and retrieve these values.

The movement functions include rotation, left, right, down and dropping movements. For the most part these functions are very similar, however to have joined them as one function I fell would have made the function far too large because of how many variables and loops would need to have been changed around. The movement works by creating a temporary array of co-ordinates matching the current block being moved. The temp is then moved in accordance to the function and is checked against my collision functions, which check if there is a block or wall collision. If there is no collision the movement is made with the current block and the changes are made to the game grid.

The Graphic and text functions just output the current grid. The text output is a straightforward loop and the graphic output just goes through the cells of each block as they are moved and updates.

The game over check just checks to see if the next block to be dropped would collide with a current block in that starting position, if so the appropriate flag is set.

DISPLAY

The display function has a graphics and text part. The text function just outputs the game as shown in the assignment. The graphics function only outputs the top of the game (score, level, etc) and the next block. The game board is dealt with by the grid graphic functions.

COMMAND-LINE INTERFACE

This function is located in main and is relatively straightforward. If there are any commands given through the command line this function records them and sets the appropriate fields located in the grid class.

SCORING

To calculate the scoring for when a row is cleared the function in the grid class that checks if there are any full rows in the grid keeps a tally of how many full rows were found. After clearing these rows and making the adjustments to the grid the function then sends it total to a scoring function which updates the score according to how many rows were cleared.

In order to calculate the additional scoring as to if a block is completely cleared, we implemented an “id” system for each cell, done by use of a pair. Each cell contains a pair which contains the level it was given in, and a unique id number given to each block. Once a row is to be cleared, each cell as it is cleared is given to a search function, which then scans through each cell of the current grid to look for a match to this unique id number. If no match is found, our program then realizes this block has been fully cleared, and passes the level of the block to the scoring function, which then increments the score appropriately.

This probably isn’t the most efficient way to do it, since the entire grid needs to be scanned for each individual cell to be cleared, but it was simple enough to implement and it works.