QUADRIS CORPORATION PRESENTS

DESIGN DOCUMENT

PREPARED BY:

HASHIM MIR

HANYANG HU

PREPARED FOR:

THE TA

DECEMBER 4, 2011

AN OUTLINE OF OUR PROBLEM SOLVING APPROACH

From a purely task and process based perspective, we started off by designing a network of data structures that would be a good fit for the requirements of this assignment. We quickly realized that this task was not easy, because each time we'd try to define a class, we were able to think of multiple areas where the class definition was lacking. Nevertheless, after multiple draft updates, we were able to come up with one that seemed to work. The classes we used are described below:

FIGURE 1: A TABLE DESCRIBING THE VARIOUS CLASSES IN OUR PROGRAM

Class	Description	Purpose and Functionality
Grid	The master class that dealt with board logistics and control. This class was the brain of our entire program – it controlled and coordinated all of the other classes, while keeping track of various dynamics of the board at the same time	Board Logistics – managing the entire board ○ Dealt with moving and dropping pieces ○ Monitoring/updating the score, level and highScore ○ Check for collision and bounds testing ○ Clearing lines ○ Restarting the game, setting the seed Control - Calling all the methods of other classes ○ Block: calling a block to rotate, move around ○ Level: keeping a record of the generator and updating as necessary, calling the right methods to update current piece and next piece, deal with levelUp, levelDown ○ Cell: keeping an array of cells and setting its occupied/other values ○ History: keeping track of what blocks were placed, and updating each time some were erased, updating the score as necessary Graphics ○ Setting graphical coordinates of each cell ○ Drawing the board ○ Drawing the score, high score, level ○ Drawing the nextPiece
Cell	This was the base upon which a larger grid (a field inside the Grid class) was built. It stored relevant information such as whether or not the cell was occupied, and the type of the block.	Functionality ○ Providing various accessors and mutators for the Grid to be able to adjust as necessary Graphics ○ Drawing and erasing the cell
Block	Stored the pieces of the game (the 7 different types of blocks), and handles their movement, rotation whether	Supporting various block functionalities: ○ Rotate ○ Move ○ Return whether block is occupying a certain cell or

	block is occupying a certain cell or not, its width and height, the coordinates that each of its individual cells occupy, as well as printing (both in graphics and text) the piece.	not ○ Returning its width and height ○ Returning the coordinates of its individual cells ○ Printing and displaying
Coordinate	A convenient way to have an x and a y coordinate for each cell	
Level	Computing next piece based on implementation of random generator or a predefined sequence (Level0). The purpose of this class was more in order to organize the different levels into one abstract class.	
History	A template that helps stores the pieces that have been placed (in the grid class, we have a vector of type History).	

After having formed the class architecture, we decided to split up the work to reduce overlaps and dependencies as follows:

- Back end – implementing the various algorithms required for program functionality (Hashim)
- Front end – dealing with and interpreting user input and the command-line interface, in addition to the makefile (Hanyang)
- Add graphics (both)
- Memory management (both)

A DISCUSSION OF THE ASSIGNMENT SPECIFICATIONS

Question 1: 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?

We decided to make a Block base class that served as a model upon which we could build the 7 different types of blocks. We used polymorphism to customize each of the 7 children class' constructors to a predefined configuration. That way, no new blocks other than the predefined 7 could be initialized, since no such constructor would be able to support their configuration. This proved to be a good safeguard against non-standard configurations, such as, for example, blocks where the four pieces are not adjacent.

Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

To handle this, we decided to have one abstract base class called Level, and all the different levels would build upon this base class. The wisdom behind this was twofold: first, that it would neatly organize all of our levels to branch out from one abstract base class, so that if we ever wanted to add a level, we could just add it in later. The second, and perhaps more relevant to the question, wisdom is that whenever we want to add a new level, we do not have to recompile all of our code. All we have to do is simply add another implementation of the base class. The remainder of our code would not complain because it only deals with the level base class, and so adding another implementation of this pure virtual class would not cause any errors in any other parts of the code.

Question 3: 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`)?

Our program was quite organized in this regard because all of our command were tied up neatly in a single command array. Adding new commands would be as simple as adding a string to our `cmdary` array of strings. A function such as rename would be very simple, because it would just require us to overwrite the existing entries of commands that are stored with their updated values. We designed our main function specifically for this purpose – to be able to easily add, remove or update commands upon request, in a neat and organized way – a task as simple as adding, removing, or updating an element to or from our `cmdary`.

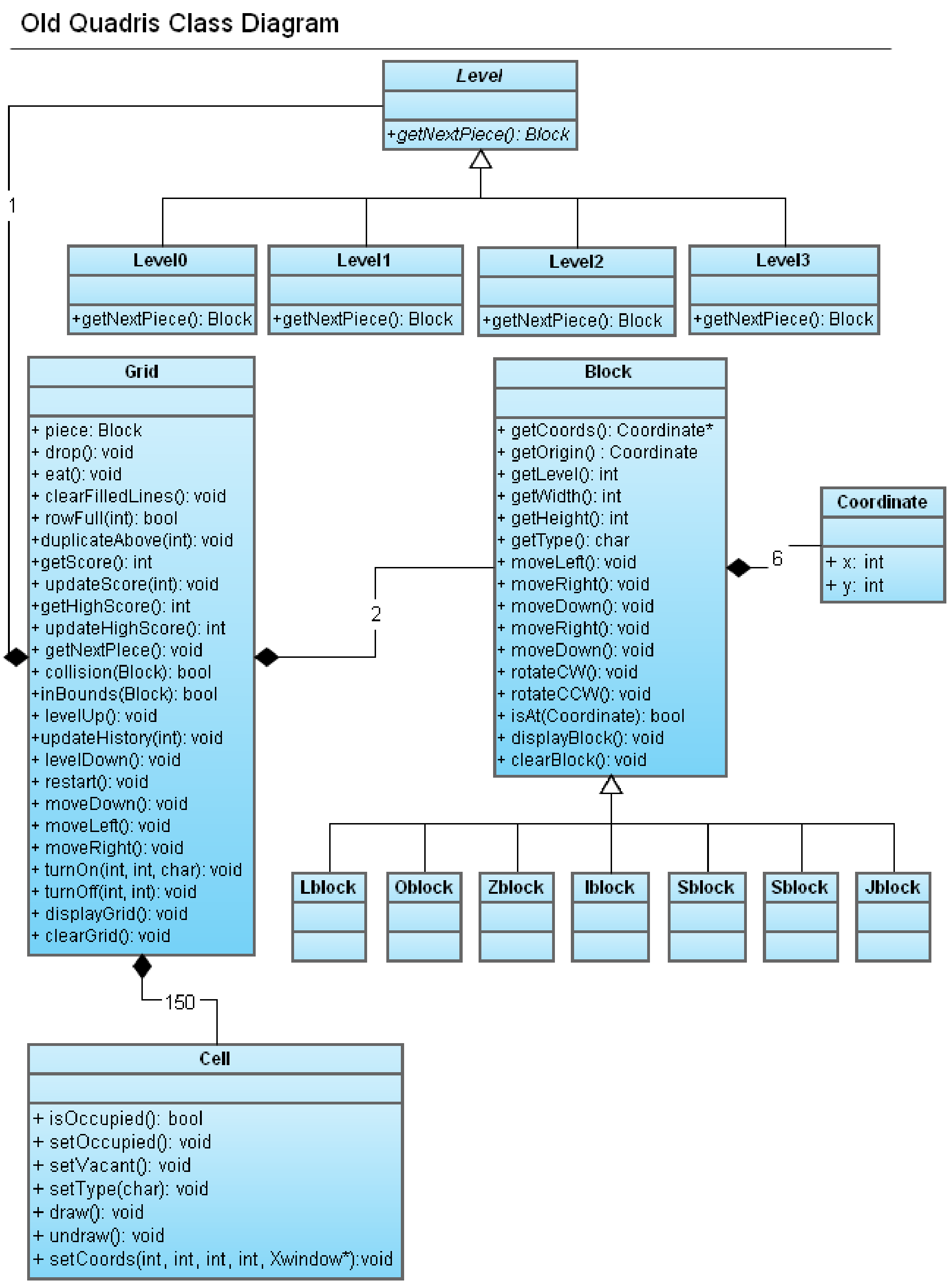
CHANGES TO OUR DESIGN

In the next section, we compare our initial UML to our final one. There are a number of noticeable differences in the two. Perhaps the most strikingly obvious one is that we added a History class to our design. This was because initially, we believed that scoring would be handled internally by the Grid class. Soon after, we thought of a cleaner way of handling scoring than dealing with everything within the Grid class. That is when we decided to add to our Grid class a vector of History objects. Each History object represented a block that had been placed on the board. The History object would need to keep track of the level that the block was added as well as the coordinates of its individual parts.

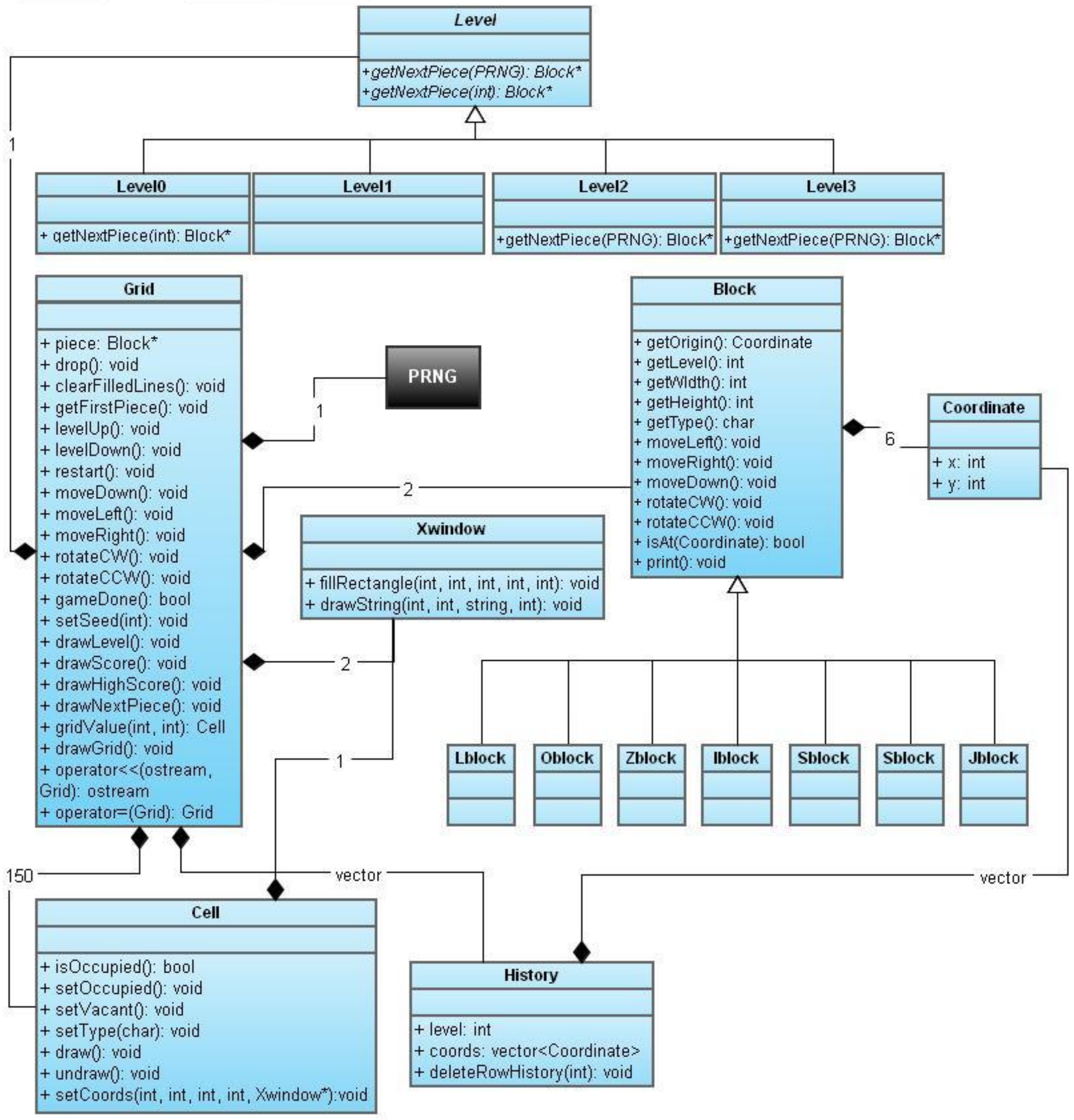
We also decided to add a PRNG object to our Grid class. Since the Grid was decidedly our control centre and largely the brain behind the entire program, we decided to give it its own PRNG object, rather than sticking with our original plan of doing the random number generation computations in our main function. This proved to be a much more logical organizational structure, because the Grid class already had autonomy over generating its own next piece, so rather than having to communicate this to the main function each time to update the generator, if given its own generator, the process would be a lot more streamlined.

A final change was adding two Xwindow objects (for game and next piece) to our grid class. This way, for graphics, the grid class could draw and erase to or from its Xwindows as required.

COMPARISON OF UMLs



New Quadris Class Diagram



DEVIATIONS FROM OUR PLAN OF ATTACK

Based on our calendar from our plan of attack, we were expecting to be done everything by Friday. We ended up actually finishing on Saturday (and Sunday for the report).

In retrospect, our deadlines were quite optimistic. The reason being that much of the tasks we had were scheduled for completion throughout the course of the week. We planned for the workload to be spread out evenly throughout the week. However, as it turned out, it ended up being something more like the bulk of the work being completed over weekends. This was because we did not have a clear idea of the workload that we would have throughout the week. Things such as stats tests and CS245 assignments got in the way during the week. We were quickly able to make amends, however. To make up for this lost time, we worked extra hard over the weekends.

One thing we heavily underestimated was the graphics component. After having completed our text based component of the project, we were so overjoyed that we got somewhat complacent, and kept pushing off progress on the graphics part. On Friday night, however, we pulled things together and worked hard to get it done.

Lastly, and this was perhaps what caused us to deviate from our plan of attack the most, was that we underestimated the scoring component of the game. We did not fully account for how much more of a dynamic the scoring component added to the project as a whole. In our initial model, we were not keeping track of what pieces had been placed on the board, nor were we able to distinguish what level that block had been placed in. We later realized that we could not just slap some bandaids on our model and get it to work, as that would result in messy code, and perhaps more complications down the road. That is why we decided to sit down and think about our options. Thankfully it was not such a big deal as to have us redo the *entire* program from scratch. It merely meant that we had to add another class and find a way to implement this class into our scoring. The task, after we designed the class to handle it, turned out to be a lot simpler than we imagined. Obviously this change took its toll on our UML, as shown above.

As far as roles were concerned, the front-end back-end division of tasks turned out to be really effective. With this system, we were able to minimize task dependencies. We could each work on our things concurrently rather than consecutively, and that way, our progress was not stalled. We also made it a habit to perpetually backup our work on a shared folder, and this also proved to be an effective tool for syncing our project.

MEETING THE PROJECT SPECIFICATIONS

We are very pleased to be able to say that we completed every single requirement of this project. We are even adding bonus features, and will submit them with our project if we are able to complete them in time. We were very careful to meticulously check each of our functions with great care and detail to ensure that the program ran as required.

Perhaps the most painstaking and time consuming precautions we took in order to make sure that our program was working was to test a good 5 full games of Quadris, running them simultaneously on the sample solution and our solution. While it was difficult to keep going (especially because each game was not like a typical game of Tetris where you could move things with the press of a key, but rather, one where we had to repeatedly type a command for each action), we looked at the end goal, and decided it was worth it to test.

We also ensured that the program did not leak any memory whatsoever. One of our concerns was whether or not we would be able to code the graphics component without leaking memory, as we were not entirely sure of the Xwindow class, and whether it would work or not. Luckily things turned out in our favour and tracking memory leaks was not much of an issue at all.

CONCLUSION

In conclusion, we hope that you have enjoyed reading our program and design document as much as we have enjoyed making them. We hope that this design document, coupled with our plan of attack, will help you get a good idea of our problem solving process, and will accurately help document our progress on this project.