Paul Younger

CMPU 203

game1 Justification

**Introduction:**

The assignment was largely open, with 6 requirements:

1. A field of play where blocks move
2. A set of "live" blocks that are player controlled. (This "set" may contain one block.)
3. A set of "dead" blocks that are no longer player controlled.
4. A scoring system
5. A win or fail state
6. A control mechanism

So for my game the field of play where the blocks move is a 7x7 grid, the player controls one block that can only move horizontally in the bottom row of the field of play. The dead blocks are obstacles that spawn randomly in the top row and slowly fall straight down. The scoring system is such that whenever an enemy block would move off the grid it instead is destroyed and the score increments by 1. The fail state is when an enemy block and the player block would collide. The player can control their object using the left and right arrow keys to move left and right. The project uses the GameWorlds library that handles graphics and player input.

**Broad View of Code Architecture:**

The first building block of the game is the grid like architecture, which is represented in data by an array of data structures called DataStructs which each contain 4 ints: x, y, key, & delay. X and Y represent the DataStruct’s x and y coordinates on the world’s grid, the key defines what type of object occupies the space, either a 0, representing a blank space, a 1, representing the obstacle objects, or a 2, the player object.

The GameWorlds system determines much of game1’s code since it is the engine which handles graphics, time, and player input. The GameWorld model is based on the World class which has several abstract methods which must be overwritten.

**Methods and Implementation:***Game1(constructor):*

Game1 implements the GameWorlds interface, World. This means that essentially the game functions by generating a new Game1 every tick, and then the bigBang method from GameWorld draws and animates the world based on that new Game1 every tick. Game1 has three fields, a DataStruct[] (array of DataStructs, another class that has an int x, int y, int key, int delay), int score, and int ticker. The DataStruct[] that is used throughout the program is called worldArray and contains essentially the entire gamestate. It is visualized as a 7x7 grid (x can be 0-6, y can be 0-6), and each space on the grid can be one of three objects, as determined by its key. If the key is 0 it means that it is a blank space, if the key is 1 it means there is an enemy object, and if the key is 2 it means that space is occupied by the player object. Most of the calculations of the game happen within the worldArray, it changes every frame based on both player input and input from other functions called at certain time intervals based on ticker. This is explained in further detail in the onTick methods explanation.

*calcPin*

calcPin takes in a DataStruct and outputs a Posn, which is a data structure that represents a pair of coordinates (int x, int y). calcPin does a simple calculation which is used to calculate the pinhole for each index of worldArray. This is necessary because in GameWorlds, every image is drawn from a “pinhole” point at its center, so for each of the 7x7 indices in worldArray each has a specific pinhole which is calculated through this method. calcPin uses the x and y cords of the DataStruct it takes in, and multiplies each of them by sqSide, a constant representing the size of each space on the grid, and then adds half of that (halfSide) to represent each coordinate.

*composeWorld*

composeWorld takes in an array of DataStructs, hereby referred to as array, and outputs a WorldImage, a class from GameWorlds. composeWorld is only ever called on worldArray, so could be optimized by changing it to take no parameters and referring directly to worldArray in its implementation instead. composeWorld functions by first creating the blank white rectangle, called scene, on top of which every individual image representing one space in the grid is draw onto. This is done using a for loop which iterates through all of array, drawing a different object depending on the key at each index location, all of which have their pinholes calculated by calcPin. For key==0 a black frame is drawn, making it so the empty spaces in the array make up a grid-like pattern. For key==1 a menacing red rectangle is drawn. For key==2 a blue rectangle is drawn that represents the player object. In each of these instances in the for loop composeWorld calls OverlayImages, a method from GameWorlds, that takes in multiple images and merges them into one layer. This implementation means that by the end of the for loop the whole array has been merged into one big image which is the output of this function.

*enemyLocations*

enemyLocations takes in an array of DataStructs, but like composeWorld, is only ever called on worldArray, so there is an opportunity to simplify the code. The ouput of enemyLocations is an ArrayList of Data Structs that represents every DataStruct for which the key==1 (there is an enemy in that space). This is implemented by iterating through the worldArray and checking if the key==1, if so the item is added to the output ArrayList, if not, the loop moves on.

*Initialize*

Initialize takes no parameters and returns void, it is only ever called on the worldArray at the start of the main method to set up the initial world state. It iterates through the list and fills in the x, y, and key values for each index of the worldArray. This works simply by using two for loops, one to represent x values and one to represent y values, filling in key=0 for every value except the players starting location which is at x==(gridSize-1)/2 and y==gridSize-1, that being the very middle column of the bottom row, in this case (3,6), and the player value, 2, is inserted into the key here. The world array is initialized through this method.

*makeImage*

This is a method of GameWorlds, which is called by bigBang to draw the world. In my current implementation is essentially a wrapper for composeWorld. It simply calls composeWorld on worldArray. This could be simplified as hinted at in composeWorlds explanation. composeWorld could be written inside of makeImage and simply directly reference worldArray and then return a WorldImage directly rather than using this wrapper function.

*moveEnemies*

moveEnemies takes in no inputs and returns void. moveEnemies mutates the worldArray and score. moveEnemies workds by calling enemyLocations to get the locations of all enemies as an arrayList<DataStruct>, and then iterates through this arrayList using a for loop. Depending on the enemies current position as given in enemyLocations different outcomes can occur. moveEnemies first checks to make sure the enemy is not in the bottom row, if this is found to be true then it checks to make sure that the space directly below the enemy is not occupied (currentEnemyWorldIndex <((gridSize-1)\*gridSize)), and if not, it moves the enemy into that location by mutating the key of the previous location to be zero, and the new locations key to be 1. If the player is found to be occupying the space which the enemy would normally move into then a collision occurs and the game ends. If the enemy object is found to be in the bottom row, then the enemy key is mutated to zero, effectively disappearing the enemy, and the score increases by one, since the enemy has successfully been avoided by the player at this point.

*onKeyEvent*

onKeyEvent is a method of GameWorlds which takes in a string generated by key input and returns a World [in this case, a Game1]. The method starts off by locating the player using the playerLocation method, and determines the player index using some simple arithmetic based on the players x and y coordinates. The index is found by multiplying the Y coord by the gridSize and then adding the X coord.

*onTick*

*playerLocation*

*spawnEnemies*

**Testing:***calcPin*

*composeWorld*

*enemyLocatins*

*initialize*

*makeImage*

*moveEnemies*

*onKeyEvent*

*onTick*

*playerLocation*

*spawnEnemies*