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 3 ints: x, y, and key. 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 calls makeImage to draw the world based on that new Game1 every tick. Game1 has three fields, a DataStruct[] (array of DataStructs, which consist of int x, int y, int key), 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.

*enemyLocations*

enemyLocations takes no inputs and ouputs 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. When the loop finishes the output ArrayList is returned.

*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 first declared and then initialized through this method, this method could also be used to restart the game.

*makeImage*

This is a method of GameWorlds, which is called by bigBang to draw the world. makeImage takes in no inputs and returns a World. makeImage 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 within the for loop makeImage 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.

*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. The rest of the calculations within onKeyEvent use this playerIndex value. If the player inputted the right arrow key the game first checks to see if the player is at the rightmost edge of the grid, and if so it checks to see if the leftmost space on the grid is open, and if so moves the player into that leftmost space, allowing the player to “wrap around” with their movement. If the leftmost space is found to be occupied in this case then there is a collision and the game ends and the score returns. Otherwise the game checks to see if the index one to the right is open, and if so moves the player there by mutating the key to represent the player object in its new position and overwriting the previous position with a blank space, represented by a key of 2 and 0 respectively. If the space to the right is found to be occupied then there is a collision and the game ends and the score is returned. The same is implemented for the left key input but mirrored, so the wrapping movement now checks when the player is currently occupying the leftmost index, and checks the vacancy of the rightmost index. Left movement checks the index one to the left for vacancy, and the same conditions apply to left collisions. Additionally the “x” key can be used to quit the game, simply implemented for debugging purposes.

*onTick*

onTick is a method of GameWorlds which takes in no input and returns a World, in this case a new Game1. onTick is called repeatedly every certain interval depending on the parameter passed into bigBang. onTick can call the moveEnemies function or spawnEnemies which both mutate worldArray. onTick returns a new Game1 with its parameters as worldArray, score, and ticker+1. This means that every game tick, the speed of which is determined by bigBang’s speed parameter increments the variable ticker. To determine when to call moveEnemies or spawnEnemies onTick uses the modulus function with ticker and the functions interval, meaning that the rate at which the functions are called every 1/interval ticks.

*playerLocation*

playerLocation takes no inputs and returns a DataStruct representing the players location in worldArray. The function iterates through worldArray and returns whichever Struct it finds that has a key==2, that meaning it represents the players location.

*spawnEnemies*

spawnEnemies generates a random integer between 0 and gridSize, in this case 0-6, and then inserts an enemy object at that index location by mutating the key to 1.

**Testing:**

The general form of the testers is based on a Boolean variable called tP or “tests passed”, every test will check for the result of the function it is testing against the desired value, if the values are not equal then tP will become false, when tP is false it will push a message saying that the test failed and then reset itself to be true. If all the tests pass there should be no fail messages.

*calcPin*

To test calcPin random Structs will be generated within the games normal range (that being x and y are both between 0 and gridSize-1., key is irrelevant for these tests.) these structs will have calcPin called on them, and compare the result to the raw arithmetic that determines the pinhole. If calcPin(randomStruct) equals the raw arithmetic on the same Struct then the function is working properly.

*enemyLocations*

To test enemyLocations a random enemy is generated and enemyLocations is called, which should return the random enemies that were just generated and nothing else. If enemyLocations generates the same arraylist<DataStruct> that we can prove is meant to be generated(targetArrayList), then the function is working properly.

*Initialize*

To test initialize there is one specific DataStruct[] it should generate every time so we can just compare its output to that desired output. This is complicated by initialize only mutating worldArray which, as a non-static, cannot be referenced in this way within the main method, making testing for it harder.

*makeImage*

To test makeImage a random DataStruct[](with all parameters within normal game ranges, 0-gridSize, 0-gridSize, 0-2, only 1 player) is generated and the appropriate logic for drawing the world is applied and saved as a targetWorldImage, this value is then compared to the result of makeImage for the same random DataStruct[], if they are equal then the function is working properly.

*moveEnemies*

To test moveEnemies a random DataStruct[](with all parameters within normal game ranges, 0-gridSize, 0-gridSize, 0-2, only 1 player) should be generated and then have the result of calling moveEnemies on it compared to the desired result as dictated by the logic contained within moveEnemies. This is complicated by moveEnemies only mutating worldArray which, as a non-static, cannot be referenced in this way within the main method making testing for it harder.

*onKeyEvent*

To test onKeyEvent a random DataStruct[](with all parameters within normal game ranges, 0-gridSize, 0-gridSize, 0-2, only 1 player) populated by a player and enemies should be generated and then have the resulting output world of calling onKeyEvent(left or right) in each different possible player position compared to the desired result as dictated by the logic contained within moveEnemies.

*onTick*

To test onTick a random world populated by a random DataStruct[](with all parameters within normal game ranges, 0-gridSize, 0-gridSize, 0-2), and compare the output of onTick to the expected output of onTick as dictated by its implementation.

*playerLocation*

To test playerLocation a random DataStuct[](with all parameters within normal game ranges, 0-gridSize, 0-gridSize, 0-2, only 1 player) should be generated and then have the result of calling playerLocation on this random DataStruct[] compared to the expected result, which in this case could be easily found by iterating through the list and returning the Struct with key==2

*spawnEnemies*

To test spawnEnemies, we know that the desired result is that every time it should flip one key in the top row to be a 1. By starting with a DataStruct[] consisting entirely of normal x&y, but all key = 0, we can easily test if it is working by calling spawnEnemies, then calling enemyLocations which should return an ArrayList<DataStruct> with only 1 member, which should be an enemy in the top row (DataStruct(0-6, 0, 1)).