Paul Younger

CMPU 203

Game2 Justification

**Introduction:**

The assignment for game2 was to develop a game about “adventure” that satisfies the following requirements:

1. Different modes with transitions between them
2. Independent non-player actors, or “mobs”
3. Persistent player attributes

*Game Manual*

The game takes place on an *n*x*n* grid where the player controls the one blue block. The player can move up, down, left and right. The black squares are obstacles which are impassable terrain, the red squares are enemies, which if they touch the player result in the fail state of the game but can be destroyed by attacking using the spacebar. The blue circle is the key/unlocker for the golden chest(yellow square). The objective of the game is to navigate the game’s world, avoiding or defeating enemies and collect the unlocker and use it to open the golden chest to win the game. The arrow keys are used to navigate up, down, left and right, and the spacebar is used to attack all spaces surrounding the player, but not that the attack cannot be used in doorways!

**Overview of Code Structure:**

The basic structure behind the game is a grid 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 objects occupies the space as follows:

* 0: blank space, draw only gridlines
* 1: player object, blue square
* 2: impassable object, black square
* 3: enemy object, red square
* 4: attack effect, green square
* 5: key/unlocker, blue circle
* 6: golden chest, yellow square

The game operates using the GameWorlds library 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:**

*Game2(constructor):*

Game2 implemenets the GameWorlds interface, World. This means the games functions are essentially implemented by generating a new Game2 every tick, and calling the makeImage function every tick to draw the world based on the newly generated Game2 each tick. Game2 has a parameter DataStruct[], which always has worldArray passed into it, this worldArray essentially contains the entire gamestate, and can be visualized as an *n*x*n* grid, with the object occupying each space of the grid determined by the key in worldArray. The objects generated by the various possible key values are described above in the Overview of Code Structure section.

*spawnPlayerMiddle:*

spawnPlayerMiddle takes no parameters and returns void. The function simply takes the center point of the worldArray as calculated based on *n* and changes the key to contain the player in that space. This function is called at the beginning of the game to initialize the world.

*onTick:*

onTick takes no parameters and returns a World. onTick is an abstract method of JavaWorlds. The JavaWorlds BigBang function calls this function every frame. It checks to see if the current screen contains any enemies, and if not sets that screen’s enemies variable to disable future spawning. Once a screen is cleared of enemies if you travel to another screen and then return the enemies should remain dead and not reappear as they first were. Additionally the onTick function checks every frame to cleanup all keys equal to 4, which is the attack effect, this process is two-fold, one it clears the visual effect of the attack, and two it clears the enemies who are caught in the attack. At the end the function simply constructs a new Game2 by passing in and returning a new Game2 with worldArray.

*onKeyEvent:*

onKeyEvent takes a string and returns a World. onKeyEvent is an abstract method of JavaWorlds. It takes in an input as a string, and based on the input and the player’s current location a new WorldArray is calculated and returned in a Game2 by the function. For example if the key input is the right arrow key, the game first checks to make sure that the player is not at the rightmost edge of the grid to avoid a pointer out of bounds error, then checks to make sure that the space one to the right of the player isn’t occupied by an enemy or a wall, and if these conditions are met then the players current location is cleared and the player is inserted in the space one to the right. These statements are largely the same for left, up and down, but instead of checked the space to the right they check the space to the left, up and down respectively. If the player is in a doorway and walks through the doorway as checked by this function then there is a function call to generate the next screen. At the end a world is always returned.

*makeImage:*

makeImage takes no parameters and returns a WorldImage. makeImage is an abstract method of JavaWorlds. makeImage iterates through every position in worldArray and generates and image based on the key in each space. The keys and their paired images are catalogued in the Overview of Code Structure above. makeImage calls calcPin to determine the coordinates to draw images at based on their index.

*genEmptyArray:*

genEmptyArray takes no parameters and returns void, only modifying worldArray. This method iterates through and fills worldArray with the appropriate x and y coordinates and fills every key with zero.

*genScrZero:*

genScrZero takes in no parameters and returns void, only modifying worldArray. This method iterates through worldArray and determines which value each index should hold in their key, essentially determining the field of play. This sets up the solid edges of the screen, the doorways, the enemies, the keys, the chest, and based on the value of the screen’s enemies/key variable, determines whether or not to spawn enemies or the key.

*genScrOne:*

genScrOne takes in no parameters and returns void, only modifying worldArray. This method iterates through worldArray and determines which value each index should hold in their key, essentially determining the field of play. This sets up the solid edges of the screen, the doorways, the enemies, the keys, the chest, and based on the value of the screen’s enemies/key variable, determines whether or not to spawn enemies or the key.

*calcPin:*

calcPin takes in a DataStruct and returns a Posn. Using a simple calculation based on the dimensions of the side of each space in the array calcPin calculates the x and y coordinates of the center of each index to aid in drawing the world.

*playerLocation:*

playerLocation takes no parameters and returns a DataStruct. This function simply iterates through worldArray and returns the DataStruct which contains the player in its key. If no player is found then an exception is thrown.

*enemiesAliveHuh:*

enemiesAliveHuh takes in void and returns an int. This method is called every frame through onTick, and determines whether or not the enemies in the current screen have been dispatched by iterating through worldArray looking for key’s which represent the enemies. Returns 0 or 1.

*calcIndex:*

calcIndex takes in two ints, x and y, and returns an int representing the index associated with the x and y values passed in.

*DXDYIndex:*

DXDYIndex takes in two ints, x and y, and returns an int representing the index associated with the x and y values passed in. Instead of being based on x and y in worldArray directly, this function calculates based on a 3x3 grid with the player being the center, and dx and dy values from -1 to 1. This is used in the space-bar attack calculation.

*disableCurrentScrSpawn:*

disableCurrentScrSpawn takes no inputs and returns void. Based on which screen the player is currently in, it disables the enemy spawning for that screen. This function is called when enemiesAliveHuh is determined to be 0, meaning that when all the enemies are gone from a screen and enemiesAliveHuh goes to 0, disableCurrentScrSpawn flips that screens spawn variable and then the enemies will be permanently gone instead of spawning again each time the room is re-entered.

**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. When random worldArray’s are referred to in testing it is assumed that they are within the range of normal acceptable worldArray’s in the game. This means they are *nxn* arrays with sides of length *side*, contains only one player index etc.

*spawnPlayerMiddle:*

To test spawnPlayerMiddle a random worldArray with no player object is generated, spawnPlayerMiddle is called, and then the middle index’s key is checked to determine if it contains a player object.

*onTick:*

To test onTick a random worldArray is generated and the output of calling onTick on this random worldArray is compared against the expected output as dictated by onTick’s implementation.

*onKeyEvent:*

To test onKeyEvent a random worldArray is generated and populated by a player, then calling onKeyEvent with each possible key input for the given worldArray and comparing the result with the expected result for worldArray as dictated by the logic for each key input.

*makeImage:*

To test makeImage a random worldArray is generated and the appropriate logic for drawing the world is applied and saved as a targetWorldImage which is compared to the result of calling makeImage for the same random worldArray.

*genEmptyArray:*

To test genEmptyArray a random worldArray is generated and then genEmptyArray is called, worldArray is then checked to make sure that each x and y is correct (which can be checked using a reverse of calcIndex), and that every key contains 0.

*genScrZero:*

To test genScrZero a random worldArray is generated and then genScrZero is called, worldArray is then checked to make sure that each x and y is correct, and that every key contains the appropriate int based on the logic contained in genScrZero.

*genScrOne:*

To test genScrOne a random worldArray is generated and then genScrOne is called, worldArray is then checked to make sure that each x and y is correct, and that every key contains the appropriate int based on the logic contained in genScrOne.

*calcPin:*

To test calcPin random DataStructs will be generated and have calcPin called on them, comparing the result to the raw arithmetic that determines the pinhole for each x and y.

*playerLocation:*

To test playerLocation a random worldArray should be generated and populated by a player object. The result of calling playerLocation should be compared against the players actual location as either passed in during generation or determined by iterating through the array.

*enemiesAliveHuh:*

To test enemiesAliveHuh a random worldArray should be generated and have enemiesAliveHuh called on it, the result of this function call should be compared against iterating through the array determining if and of the keys contain an enemy. A=

*calcIndex:*

To test calcIndex random x and y from 0 to n should be generated and have calcIndex called passing in x and y. The result of this function call is compared against the raw logic which determines the index based on x and y.

*DXDYIndex:*

To test DXDYIndex a random dx and dy from -1 to 1 should be generated and have DXDYIndex called passing in dx and dy. The result of this function call is compared against the raw logic which determines the index based on dx and dy.

*disableCurrentScrSpawn:*

To test disableCurrentScrSpawn a random worldArray should be generated and have enemiesAliveHuh called, then disableCurrentScrSpawn can be called and the current screens enemy spawn variable can be checked, if enemiesAliveHuh was one, the current screen’s spawn variable should also be one, if enemiesAliveHuh was zero, the current screen’s spawn variable should also be zero.