Task 1 - Adding a player

Task 2 - Player movement

Task 3 - Level generation

Task 4 – Better Player movement

Task 5 – Better Level generation

Task 6 – Even Better Player movement

Task 7 – Mining

Task 8 – Adding moles

Task 9 – Mole movement

Task 10 – Improved level generation

Task 11 – Better mole movement

Task 14 – Player energy

Task 15 – New levels

Task 17 – Power up

Task 18 – Scoring

Task X – Start screen & Custom options

In the CustomGUI2 class, a number of private integer variables are declared. Also a few SpinnerNumberModel classes are instantiated. An integer array is also instantiated called gameAttributes. This is used to pass the customized attributes to the GameEngine class when its instantiated and after the user confirms their custom options. In the constructor method, the initComponents() method is called. This instantiates all the labels, spinners, buttons and other GUI elements and places them on the JFrame window. In the method for the confirm button, a Boolean flag variable is initialised, some Strings are stored as the inputs from the spinbox classes (used to choose the customisable attribute values). These are then converted to integers and then the JTextArea is cleared. The playerX and playerY variables are checked against the levelHeight and levelWidth variables and if they are bigger, then an error message is inserted into the JTextArea and attributesValid is set to false. Otherwise, a message is inserted into the JTextArea that says “All fields valid” and attributesValid is set to true. The final part of this method checks if attributesvalid is true, if it is, stores the integer variables into the gameAttributes array, instantiates the GameGUI and GameEngine classes, sets the Gui to visible, instantiates the InputHandler, registers the key handler and starts the game with difficulty “Custom”.

The last few editions are in the GameEngine class. A few extra attributes are declared for the “Custom” mode and a gameDifficulty attribute is also declared. The constructor is modified to set the gameAttributes to the necessary attributes of the class if they exist (“Custom” mode is selected), otherwise default values are used for the class attributes. In generateLevel(), instead of the levelNumber switch, a gameDifficulty switch is used with the same values for the tile chances. In addMoles() a switch is created for the gameDifficulty as well, setting different numbers of moles depending on the difficulty selected. In createPlayer() an if statement is used to check if it’s “Custom” mode and uses the attributes that have been set by the user on the custom screen to instantiate the Player. Otherwise, it uses the default values to instantiate the Player. The final addition is passing in the gameDifficulty to startGame() and setting it.

|  |
| --- |
| The method (createPlayer()) will first generate a random number in the range [2,11] (not including 11) and then multiply it by 50 to get a multiple of 50 between 100 and 500 and assigned to the maxEnergy variable that will be used to pass into the constructor of the Player class when it is instantiated. Next the scanner class is instantiated, and variables (x and y) are declared. Next user inputs are taken for the x and y co-ordinates until the user enters a value that it on the grid (0-34 for x and 0-17 for y). A do-while loop is used for this and the Scanner.nextInt() method for the inputs. NOTE: x and y will be entered by the user on the default input/output screen.  Finally, the Player class is instantiated with all 3 values (maxEnergy, x and y). |

|  |
| --- |
| The method (movePlayer()) will first store the current values of the Player x and y co-ordinates in new integer variables to be used in the actual movement of the Player. Next, a series of switch-case statements are used to check the value of the direction variable (passed in as a parameter) and implement the movement accordingly. NOTE: All cases make use of the setPosition() method of the Player and include a break. The first case is ‘N’ which means the player is moving up. For this case, the parameters passed to the setPosition() method of Player are playerX and (playerY-1). This will move the player up from the perspective of the user. The next case is ‘S’ which means the Player is moving down. For this case, the parameters passed to the setPosition() method of Player are playerX and (playerY+1). This will move the player down from the perspective of the user. The next case is ‘W’ which means the Player is moving left. For this case, the parameters passed to the setPosition() method of Player are (playerX-1) and playerY. This will move the player left from the perspective of the user. The final case is ‘E’ which means the Player is moving right. For this case, the parameters passed to the setPosition() method of Player are (playerX+1) and playerY. This will move the player right from the perspective of the user. The break statement is used to make sure the Player doesn’t move multiple directions at once. |

|  |
| --- |
| The method (generateLevel()) will first instantiate a 2D array using the LEVEL\_WIDTH and LEVEL\_HEIGHT constants as the dimensions of the level array. Next, a nested for loop will be used to iterate over the level array. The first for loop will finish when it reaches the level.length value (same as LEVEL\_WIDTH) and the second for loop will finish when it reaches the level[i].length value (same as LEVEL\_HEIGHT). Finally the method will continue instantiating new Tile objects that have the TileType EMPTY as long as the nested for loop runs and store it in level[i][j] which will be the current element of the 2D array called level. |

|  |
| --- |
| The method (movePlayer()) will now implement a series of if statements within the different cases, one for each case. The first if statement (in the ‘N’ case) will check to see if the new Y co-ordinate of the Player is not less than 0 and if it isn’t then will move the player. Likewise, in the ‘S’ case the if statement will check if the new Y co-ordinate is not more than the (LEVEL\_HEIGHT-1) and if it isn’t then will move the player. The same will apply for the ‘W’ and ‘E’ cases but instead of using the Y co-ordinate, it will use the x co-ordinate and LEVEL\_WIDTH instead of LEVEL\_HEIGHT for case ‘E’. The rest of the code is the same as in Task 2 which was the initial setup of this method. |

|  |
| --- |
| The method (generateLevel()) will now initialise 2 variables that will indicate the chance that a tile could be DIRT or COPPER. The DIRT chance is set to 65% and the COPPER chance to 25%. This is after the instantiation of the level array but before the nested for loop. Inside the nested for loops there is a call to the nextInt() method of the random number generator and the result is stored as randomChance (a new integer variable). Then there is an if-elseif-else structure where the randomChance variable is used to determine which tile type should be instantiated and stored in the level array. If the chance is between 0 and 24 then the COPPER tile type will be instantiated, if it is more than 24 but less than 65 then the DIRT tile type will be instantiated. Otherwise, an EMPTY tile type will be instantiated. |

|  |
| --- |
| The method (movePlayer()) will now implement a nested if statement with each of the if statements within each case. Before the nested if statement, in each case a new variable currentTile will be assigned to the tile that the Player will move to depending on the case which corresponds to the direction the Player is going. The if statements will check if currentTile is a TileType EMPTY and only move the Player if it is. |

|  |
| --- |
| The method (movePlayer()) will now start by using the getEnergy() method of the Player and storing the energy value in a variable called mineEnergy. The only other addition will be an else statement that will go with the nested if statement and execute if the currentTile is not a TileType EMPTY. Within the else statement there will be a call to the mine() method of currentTile taking mineEnergy as a parameter. This will change the tile to be of TileType EMPTY if the Player moves towards it after which point the Player will be able to move into the now empty tile. |

|  |
| --- |
| The method (addMoles()) will start by initialising the moles array as a size 10 standard array. Next 5 Mole objects are instantiated and stored in the moles array, each with different values for the x and y co-ordinates and with the same maxFullness value. |

|  |
| --- |
| The first method (moveAllMoles()) starts by using an enhanced for loop to iterate through the moles array. Inside the for loop, an if statement checks if the current element of the array is a Mole or if it’s null. If it is a mole, then it will call the moveMole() method with the current mole as the parameter. The second method (moveMole()) firstly retrieves the x and y coordinate of the current mole and then generates a random number for the direction variable. Next, a switch-case statement is used, similar to the one used in movePlayer() initially. This has cases 1,2,3,4 as per the random numbers generated and each case corresponds to a direction. The setPosition() method of Mole is used in each case and the new coordinates are passed in as parameters. Then at the end of each case (except case 4) a break is used. |

|  |
| --- |
| The method (generateLevel()) now has additional fixed probabilities for more tile types. These tiles are being instantiated using if statements in a similar way to in task 5. Then in each case in movePlayer() the tile type is checked. If it is SILVER then the tile “exploded”, calling a new explode method I made for this specific scenario. If it is URANIUM, then the level height and width are decreased. Code in the GUI class has been modified to account for the level dimensions changing as well. In the explode() method, 3 parameters are passed to it, playerX, playerY and mineEnergy. A nested for loop is used with playerX and playerY as the loop controller variables. Inside the nested for loop, is the call to the mine() method of the tile in the level array using the mineEnergy variable as the parameter of the method. |

|  |
| --- |
| The method (moveMole()) now has verification for moving off the map and into other moles and the player. Firstly, the player x and y co-ordinates are taken. Then a few variables, movementValid (boolean), thisMoleY (int), thisMoleX (int) are declared to be assigned later on. In each case, a flag variable to verify the movement (movementValid) is assigned to true. Then a for loop is used to iterate over the moles array. Inside this for loop an if statement is used to check if the Mole in the moles array is not null. If it isn’t null, then its coordinates are stored (depending on which case, x or y is taken) and the coordinates are compared to where the Mole m is moving (again depending on this case). If the coordinates are the same, then movementValid is assigned to false. Otherwise, it is not changed. After this for loop, an if statement is used to check if the Player is where the Mole m is moving. Again, if the coordinates match, then movementValid is assigned to false. The penultimate addition (in each case), is incorporating the movementValid variable into the setPosition if statement. The final addition is to add an else to connect to above mentioned if statement and recursively call the method so the Mole m can still move.              Task 12 – Mining moles                 Task 13 – EXPLODING MOLES |

|  |
| --- |
| The method (doTurn()) now adds 1 line to increase the Player energy by 2 each turn. In the generateLevel() method, an extra variable is added for the base chance and another else if statement is used to generate this tile. Also in the movePlayer() method, in each case, in the mining else statement, the Player energy is decreased by 5. Also, an else if is added within this if-else code block that compares the TileType of currentTile to BASE and if it is a BASE Tile, uses the refillEnergy() method of the Player to replenish their energy and also moves the Player onto that Tile. |

|  |
| --- |
| The method (allOreMined()) is used to count all the ores in the level and check if the oreCount is greater than 0 or not. It does this by firstly assigned a new variable oreCount to 0. Next, it uses a nested for loop to iterate over the level array. Inside these for loops, it stores the currentTile as level[i][j]. Next it uses an if statement to check if currentTile is not null and inside this a switch case statement to check the type of currentTile. If the type is COPPER, SILVER or URANIUM, it increments oreCount by 1. Below these for loops, it then checks if oreCount is greater than 0. If it is, returns false indicating that not all the ore have been mined by the Player. If it isn’t, then returns true, indicating that there are no remaining ores left on the map. The doTurn() method now includes a variable storing the result of the above mentioned method, variables to store the Player’s x and y coordinates, and a variable to store the Tile where the Player currently is. Then an if statement is used to check if all ores have been mined and that the Player is currently on a Tile of TileType BASE. The if statement then calls the nextLevel() method if it is executed. The nextLevel() method firstly increments the levelNumber, then calls the generateLevel(), addMoles(), placePlayer() and giu.updateDisplay() methods to progress to a new level. The placePlayer() method, sets the player position with 10 as x and 5 as y. In generateLevel() a switch case statement is used which uses the levelNumber as the variable and sets different chances for tile types depending on the levelNumber. In addMoles(), a rule switch is used to set different values for the maxFullness of each Mole, depending on the levelNumber. Finally, if statements are used to check the levelNumber and spawn extra moles with increments of the levelNumber up to level 4. Those are all of the new additions required to generate new levels dynamically. |

|  |
| --- |
| The method (movePlayer()) now includes an increase to the mining strength when a URANIUM Tile is mined. Firstly, a new variable called minedTileType is declared. In the mining else statement of each case, this variable is used to store whatever is returned from the mine() method of currentTile. Thereafter, this variable is checked against TileType.URANIUM and if it is (Player has fully mined a URANIUM Tile) then the miningStrength is incremented by 5. |

|  |
| --- |
| The method (movePlayer()) now increments the Player score depending on the ore they mined. In the mining else statement, in each case, if the Tile is of type SILVER, score is incremented by 2, 1 for COPPER and 5 for URANIUM. Then in the doTurn() method, score is printed after each turn of the Player. |