

# Object Oriented Programming

## Final Project Report



### Student Information:

**Surname:** Basuki

**Given Name:** Adrian Nugroho

**Student ID:** 2702298210

**Course Code :** COMP6699001

**Course Name :** Object Oriented Programming

**Class** : L2AC

**Lecturer** : Jude Joseph Lamug Martinez, MCS

**Type of Assignment:** Final Project Report

### Submission Pattern:

**Due Date** : 20 June 2024

**Submission Date** : 20 June 2024

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

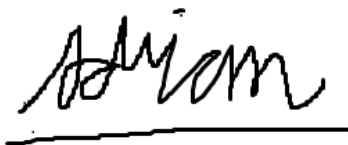
### **Plagiarism/Cheating**

Binus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

### **Declaration of Originality**

By signing this assignment, I understand, accept, and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

**Student Signature:**

A handwritten signature in black ink, appearing to read 'Adrian', is written over a solid black horizontal line.

**Adrian Nugroho Basuki**

## **Table of Contents**

<b>Background</b>	<b>4</b>
<b>Project Specifications</b>	<b>4</b>
Description	4
Controls	4
<b>Solution Design</b>	<b>5</b>
Class Diagram	5
Important Sections of The Code	6
Other Classes	18
<b>Screenshots</b>	<b>19</b>
<b>Resources</b>	<b>22</b>

## **Background**

When thinking of a project to make, I contemplated whether to make some kind of management system, a 2D game, or another discord bot (if I did not find any other options). In the end, I finally built up my courage to try something more difficult than what I made for my algorithm and programming project and decided to make a 2D video game. This time, I had to work with much more files than I did back then, causing it to be more confusing, as I had to switch files often to edit methods and attributes here and there. However, this is a good thing, as I am teaching myself to be more organized with my code, organizing everything into classes.

## **Project Specifications**

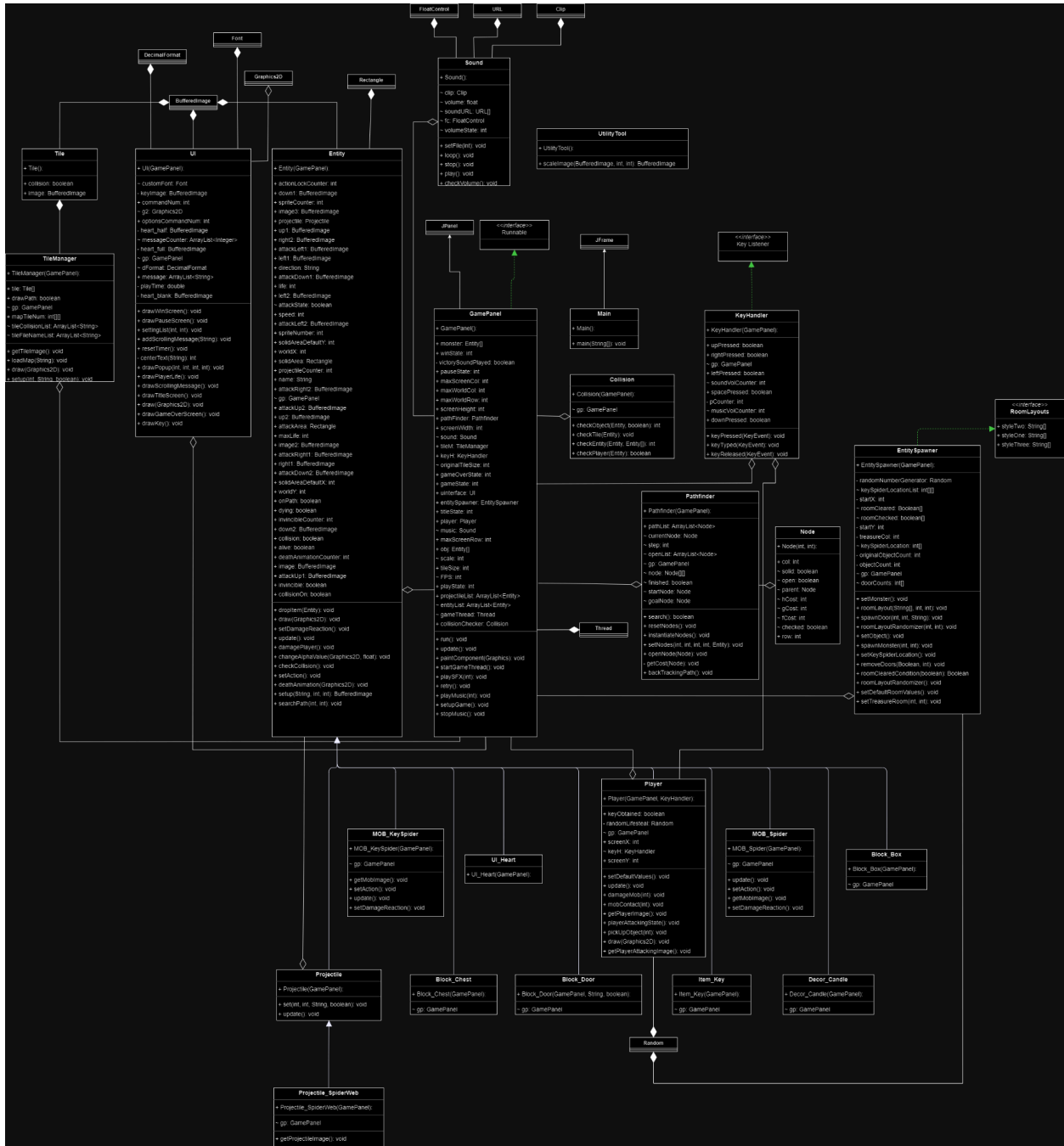
### **Description**

The game I created is a roguelike game where the player has to go around the map to find a mob that is holding a key that will unlock the door to the room where the treasure lies. Once the player touches the treasure, the game ends and the time the player spent finding the treasure will be shown. This game is inspired by the mobile game “Soul Knight”, which is a roguelike game that lets players go through several randomly generated stages with sub-stages within those stages.

### **Controls**

The game contains a few keybinds that will let the players do different actions. When the player first opens the game, they will see a title screen. This title screen, along with the pause screen, game over screen and win screen, could be navigated by using the up and down arrow keys on the keyboard. Next, once the player is in the game, they will enter the play state. To move the characters, the player is required to use their W, A, S, and D keys. Once the player enters the first room, they will encounter mobs spawning around the room. To fight them, players could press the spacebar to attack. If the players want to pause in the middle of the game, they could press escape to enter the pause state and open the pause menu where the players could also adjust several settings.

## Class Diagram



## Important Sections of The Code

There are several key features that were implemented in the code so that the game could work and others that could make the game better. These features include collision detection, camera movement, and many more.

Firstly, the game runs using a loop to update the contents of the game and draw them afterwards, which is initiated by the startGameThread method. However, different computers may handle the game differently, causing inconsistency in the frames per second. Some computers will run the loop too quickly that the content of the game would not load properly. To fix this, I used a concept called delta time which controls the speed of the game based on elapsed time, rather than the speed of individual frames. This calculation requires recording the timestamp at the start of each frame, finding the interval between updates in nanoseconds (1000000000/FPS), and the current timestamp in the loop. The delta value will keep increasing in the loop as it is added by the quotient of the difference between current time and last time and the draw interval.

```
public void run(){
    double drawInterval = 1000000000/FPS; // Time interval between updates
    double delta = 0; // Delta value
    long lastTime = System.nanoTime(); // Timestamp when frame starts
    long currentTime;

    // Updating and drawing the game information based on FPS
    while(gameThread != null) {

        currentTime = System.nanoTime();

        delta += (currentTime - lastTime) / drawInterval;

        lastTime = currentTime;

        if(delta >= 1){
            // Call update method to update information
            update();

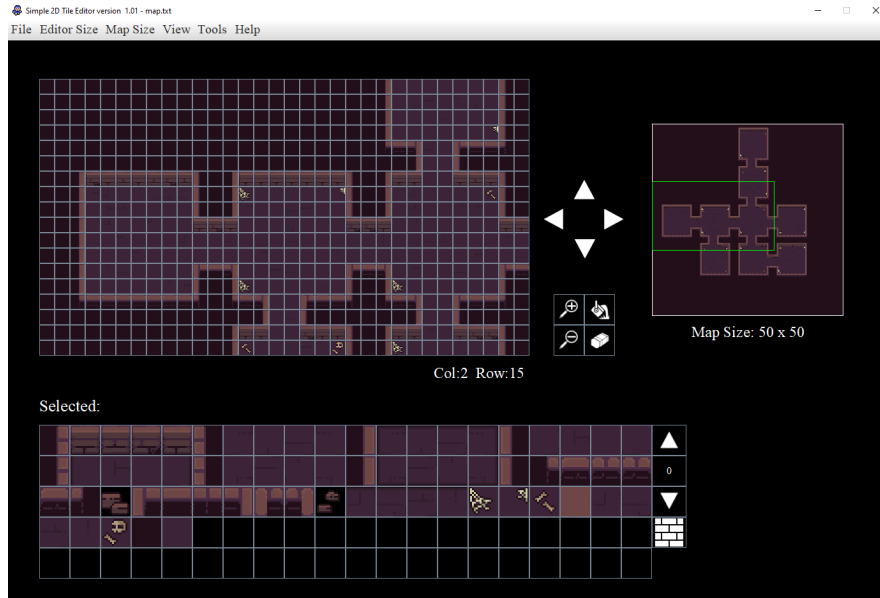
            // Call paintComponent method to redraw
            repaint();

            delta--;
        }
    }
}
```

Once the loop is initiated, the game needs to load the map. This is where the Tile and TileManager class is used. The Tile class is a class for every unique tile in the game. Each tile has a unique image and a collision attribute which could either be true or false (this will be explained later in the collision detection section). I used RyiSnow's 2D tile editor when creating this game which allowed me to save each tile's data, including its image title and collision status, from the tileset I imported to the editor into a txt file. When the TileManager class is instantiated in the GamePanel class, it activates a loop that reads every line of the txt file until it reaches a

null value, meaning that it has reached the end of the file. The loop will save all the tile file names and the collision status of each tile into an arraylist.

## Tile Editor



After the map is created, the map is exported from the tile editor in the form of a txt file which contains 50 rows and 50 columns of numbers corresponding to the index of the tiles in the Tile array instantiated in the TileManager class. To load the map, the TileManager class's loadMap method will take in the path of the map txt file as an argument and read the contents using InputStream to read the txt file, InputStreamReader to convert bytes to characters and BufferedReader to buffer the input for efficient reading. While the code has not reached 50 columns and 50 rows (maximum world columns and rows defined in the GamePanel class), the code will keep adding the numbers from the txt file to the mapTileNum 2D array, creating a 2D array of the map. This 2D array will then be used in the draw method which renders the map to the game window, but limits it to only rendering the tiles that are visible on the screen to make the program more efficient.

## Draw Method of TileManager

```
int worldCol = 0;
int worldRow = 0;

while(worldCol < gp.maxWorldCol && worldRow < gp.maxWorldRow){

    int tileNum = mapTileNum[worldCol][worldRow];

    // Tile's position on the world map
    int worldX = worldCol * gp.tileSize;
    int worldY = worldRow * gp.tileSize;

    // Tile's screen position
    int screenX = worldX - gp.player.worldX + gp.player.screenX;
    int screenY = worldY - gp.player.worldY + gp.player.screenY;

    // Limiting the tiles rendered to only the tiles that are seen on the screen area
    if(worldX + gp.tileSize > gp.player.worldX - gp.player.screenX && worldX - gp.tileSize < gp.player.worldX + gp.player.screenX &&
        worldY + gp.tileSize > gp.player.worldY - gp.player.screenY && worldY - gp.tileSize < gp.player.worldY + gp.player.screenY){
        g2.drawImage(tile[tileNum].image, screenX, screenY, gp.tileSize, gp.tileSize, observer.null); // Drawing the tile's image
    }

    worldCol++;

    // Moving to another row
    if(worldCol == gp.maxWorldCol){
        worldCol = 0;
        worldRow++;
    }
}
```

Next, the game needs a proper method so that the player character is able to move around the map. To achieve this, I have made the player stay at the center of the screen and shift the map around, making it seem like the character is moving when it actually stays in place. The draw method of the TileManager class is where this ‘camera’ feature is implemented. World X and Y are coordinates of the map, while Screen X and Y are coordinates of the game window. Coordinates (0,0) for Screen X and Y would not always mean that the World coordinates are also (0,0) because (0,0) for Screen X and Y is the screen coordinates for the top left corner of the window. The offset between the screen and world coordinates is calculated by subtracting the current checked world coordinates in the loop with the player’s world coordinates and adding the result with the player’s screen coordinates (coordinates for the center of the screen).



## Player Movement in Player Class

```
else if(keyH.downPressed || keyH.upPressed || keyH.leftPressed || keyH.rightPressed) {
    if(keyH.upPressed){
        direction = "up";
    }
    if(keyH.downPressed){
        direction = "down";
    }
    if(keyH.leftPressed){
        direction = "left";
    }
    if(keyH.rightPressed){
        direction = "right";
    }
}
```

## Player Movement in KeyHandler Class

```
else if(gp.gameState == gp.playState){
    if(code == KeyEvent.VK_W){
        upPressed = true;
    }
    else if(code == KeyEvent.VK_A){
        leftPressed = true;
    }
    else if(code == KeyEvent.VK_S){
        downPressed = true;
    }
    else if(code == KeyEvent.VK_D){
        rightPressed = true;
    }
}
```

```
public void keyReleased(KeyEvent e) {
    int code = e.getKeyCode();
    if(code == KeyEvent.VK_W){
        upPressed = false;
    }
    else if(code == KeyEvent.VK_A){
        leftPressed = false;
    }
    else if(code == KeyEvent.VK_S){
        downPressed = false;
    }
    else if(code == KeyEvent.VK_D){
        rightPressed = false;
    }
    else if(code == KeyEvent.VK_SPACE){
        spacePressed = false;
    }
}
```

These are the player movement key settings from the Player and KeyHandler class. If the keys are pressed, players will move towards a certain direction at a specified speed and once the key is released, the player will stop moving.

To check if a player or mob, an entity, is hitting a solid tile, an object, or another entity, a collision checker class is required. Therefore, the Collision class was created. This class has a few methods to check for solid tiles, check for objects, check for other entities, and check for the player.

Firstly, there is the checkTile method which basically predicts where the player will be in the next step, based on the direction, and checks the tiles on the left and the right of wherever the player is facing. If at least one of those tiles are solid tiles, then the player will stop moving.

## checkTile method

```
public void checkTile(Entity entity){

    // Finding position of the player's solid area (hit-box) with respect to the world map
    int entityLeftWorldX = entity.worldX + entity.solidArea.x;
    int entityRightWorldX = entity.worldX + entity.solidArea.x + entity.solidArea.width;
    int entityTopWorldY = entity.worldY + entity.solidArea.y;
    int entityBottomWorldY = entity.worldY + entity.solidArea.y + entity.solidArea.height;

    // Finding column and row numbers of the coordinates
    int entityLeftCol = entityLeftWorldX/gp.tileSize;
    int entityRightCol = entityRightWorldX/gp.tileSize;
    int entityTopRow = entityTopWorldY/gp.tileSize;
    int entityBottomRow = entityBottomWorldY/gp.tileSize;

    // Integer variables to check two tiles for each direction
    int tileNum1, tileNum2;
```

## Checking collision for up direction

```
// Predict where player will be after moving
switch(entity.direction) {
    case "up":
        entityTopRow = (entityTopWorldY - entity.speed)/gp.tileSize;
        tileNum1 = gp.tileM.mapTileNum[entityLeftCol][entityTopRow]; // Checking collision on left side
        tileNum2 = gp.tileM.mapTileNum[entityRightCol][entityTopRow]; // Checking collision on right side
        if(gp.tileM.tile[tileNum1].collision || gp.tileM.tile[tileNum2].collision){
            entity.collisionOn = true;
        }
        break;
```

The next three methods work in a similar way with one another. The first method among the three would be the checkObject method. This method checks whether a player or an entity (a mob) is colliding with an object. First, the code loops through the GamePanel class's list of objects. Then, it calculates the position of the object's and entity's solid areas (hit-boxes). The code will then predict where the entity's solid area will be after taking a step towards the direction they are facing and check if the solid areas of the entity and the object intersect. If they do, the entity will stop moving in that direction, and if the entity is a player, the method will return the index of the object, which will be used in the pickUpObject method from the player class. The other two methods use the same way of using the intersection of the solid areas to check whether an entity comes in contact with another entity or the player.

## checkObject method

```
public int checkObject(Entity entity, boolean player){
    int index = 999;

    for(int i = 0; i < gp.obj.length; i++){
        if(gp.obj[i] != null){
            // Entity's solid area (hit-box)
            entity.solidArea.x = entity.worldX + entity.solidArea.x;
            entity.solidArea.y = entity.worldY + entity.solidArea.y;

            // Object's solid area (hit-box)
            gp.obj[i].solidArea.x = gp.obj[i].worldX + gp.obj[i].solidArea.x;
            gp.obj[i].solidArea.y = gp.obj[i].worldY + gp.obj[i].solidArea.y;

            switch(entity.direction){
                case "up":
                    entity.solidArea.y -= entity.speed;
                    break;
                case "down":
                    entity.solidArea.y += entity.speed;
                    break;
                case "left":
                    entity.solidArea.x -= entity.speed;
                    break;
                case "right":
                    entity.solidArea.x += entity.speed;
                    break;
            }

            if(entity.solidArea.intersects(gp.obj[i].solidArea)){
                if(gp.obj[i].collision){
                    entity.collisionOn = true;
                }
                if(player){
                    index = i;
                }
            }

            entity.solidArea.x = entity.solidAreaDefaultX;
            entity.solidArea.y = entity.solidAreaDefaultY;
            gp.obj[i].solidArea.x = gp.obj[i].solidAreaDefaultX;
            gp.obj[i].solidArea.y = gp.obj[i].solidAreaDefaultY;
        }
    }
}
```

Even though collision is an important feature that makes the game work, there is another step that gives a bit of unpredictability to the game, which is the entity spawning. This step is mainly done in the EntitySpawner class's setObject and setMonster methods, which are called in the GamePanel class. Other than just spawning objects and mobs, this class is also responsible for the random arrangement of the blocks in each room, assigning the location of the treasure room, and assigning the location of the spider that drops the key.

In the constructor of the EntitySpawner class, there are arrays for how many doors each room has, the possible key spider locations and the coordinates for the top left corner of the first room. This constructor is then followed by a method that also sets several other default values, such as if a room is already checked by the player or if a room is already cleared. I chose to make a separate method for this because I want it to be used to reset those values when the player retries.

```

public EntitySpawner(GamePanel gp){
    this.gp = gp;
    this.doorCounts = new int[]{6, 2, 8, 2, 2, 2, 2}; // How many doors each room has
    this.keySpiderLocationList = new int[][]{{13,32}, {33,22}, {33,32}}; // Possible key spider locations

    // First room top left coordinates
    startX = 14*gp.tileSize;
    startY = 23*gp.tileSize;
}

// Default values for if a room is already checked and cleared
2 usages
public void setDefaultRoomValues(){
    keySpiderLocationList = new int[][]{{13,32}, {33,22}, {33,32}};
    for (int i = 0; i < roomChecked.length; i++) {
        roomChecked[i] = false;
        roomCleared[i] = false;
    }
}
}

```

After assigning the default values, the setObject method is called, assigning the treasure room, and generating the room layouts. The treasure room is assigned using a random number generator which produces an output of either 1 or 2. If the output is 1, the treasure room will be at the top of the map, while if the output is 2, the treasure room will be at the bottom right of the map. Next, the setObject method will call the roomLayoutRandomizer method and pass the coordinate offset from the first room as the arguments. This will deploy the box objects in each room based on a random choice, done by another random number generator, that selects one of three room layout options provided in the RoomLayouts interface that I created. Before reaching the end of the method, the code makes sure that objects other than the treasure will not be spawned inside of the treasure room. So, it checks if the treasure room's column is either 23 or 33, then it generates the random layout in the other possible treasure room that is currently empty. Lastly, this method will call the setKeySpiderLocation method which chooses the room where the key spider will spawn.

```

// TREASURE LOCATION
int randomTreasureLocation = randomNumberGenerator.nextInt( bound: 2) +1;
switch(randomTreasureLocation){
    case 1:
        setTreasureRoom( leftCol: 23, bottomRow: 8);
        treasureCol = 23;
        break;
    case 2:
        setTreasureRoom( leftCol: 33, bottomRow: 38);
        treasureCol = 33;
        break;
}
}

```

```

// LOADING MAP OBJECTS
roomLayoutRandomizer(); // Room 1
roomLayoutRandomizer( xOffset: 0, yOffset: 10); // Room 2
roomLayoutRandomizer( xOffset: 10, yOffset: 0); // Room 3
roomLayoutRandomizer( xOffset: 10, yOffset: 10); // Room 4
roomLayoutRandomizer( xOffset: 20, yOffset: 0); // Room 5
roomLayoutRandomizer( xOffset: 10, yOffset: -10); // Room 6
if(treasureCol == 23){
    roomLayoutRandomizer( xOffset: 20, yOffset: 10); // Room 7
}
else if(treasureCol == 33){
    roomLayoutRandomizer( xOffset: 10, yOffset: -20); // Room 7
}

// KEY SPIDER LOCATION
setKeySpiderLocation();

```

Now that the objects are all spawned, the `setMonster` method will be called repeatedly in the `GamePanel`'s update method to check for changes in every frame. The first section of the code, the largest section, checks if a room is already checked and if a player is entering it. If both conditions are met, the spiders will be spawned along with doors that keep the player in the room until they have cleared it. The room's status will be set to be checked once a player enters it, and once they enter it the second time, mobs will not spawn anymore. The code will then iterate over each room to check if they have killed all the spiders. If they have, then the room's cleared status will be true. The code then iterates over the `roomCleared` array again and calls the `removeDoors` method for that room. This would remove the doors based on how many doors that spawned in the room, allowing the player to move to other rooms again.

### Checking first room

```
// See if room 1 is checked and if player enters the room
if(!roomChecked[0] && gp.player.worldX== 12*gp.tileSize){

    // Spawn spiders
    spawnMonster( xOffset: 0, yOffset: 0);

    originalObjectCount = objectCount; // Keeping the previous object count without the doors

    // Spawn the door objects
    spawnDoor( col: 11, row: 25, doorDirection: "side");
    spawnDoor( col: 15, row: 30, doorDirection: "front");
    spawnDoor( col: 21, row: 25, doorDirection: "side");

    roomChecked[0] = true;
}
```

### Checking for room clears and removing doors if cleared

```
// Checking if room is cleared
for (int i = 0; i < roomCleared.length; i++) {
    if(roomCleared[i] != null && !roomCleared[i]){
        roomCleared[i] = roomClearedCondition(roomChecked[i]);
    }
}

// Removing doors when all entities in that room are killed (room is cleared)
for (int i = 0; i < roomCleared.length; i++) {
    if(roomCleared[i] != null && roomCleared[i]){
        removeDoors(roomCleared[i], doorCounts[i]);
        roomCleared[i] = null;
    }
}
```

Once the mobs are spawned, they either move randomly, or follow a pathfinding algorithm, A\* algorithm, to chase the player when the player gets close. However, before explaining the pathfinding algorithm, I would like to explain more about the `Entity` class and how the mobs behave.

The `Entity` class is the parent class for every entity in the game, which are the player, objects, mobs and the projectile class. This provides a template for what attributes an entity

should have, including health, maximum health, speed, name, collision status, and a solid area. It has several important methods that are responsible for the way monsters behave, including `setAction`, which decides what the mobs do, `setDamageReaction`, which decides what the mobs do when being attacked, and `checkCollision`, which checks if the mob is colliding with another entity. The `searchPath` method that runs the pathfinding algorithm is also defined in this class.

## Normal Spider

```
name = "Spider";
speed = 2;
maxLife = 2;
life = maxLife;
```

```
// Monster being aggro towards player
if(onPath){

    // Setting up the pathfinding to target the player
    int goalCol = (gp.player.worldX + gp.player.solidArea.x)/gp.tileSize;
    int goalRow = (gp.player.worldY + gp.player.solidArea.y)/gp.tileSize;
    searchPath(goalCol, goalRow);

    // Start shooting projectiles
    if(!projectile.alive && projectileCounter == 60){
        projectile.set(worldX, worldY, direction, alive: true); // Creating a new projectile
        gp.projectileList.add(projectile); // Adding it to the list in GamePanel to be updated
        projectileCounter = 0; // Projectile counter to limit the fire-rate
    }
}
```

## Key Spider

```
name = "KeySpider";
speed = 2;
maxLife = 5;
life = maxLife;
```

```
// Monster being aggro towards player
if(onPath){

    // Setting up the pathfinding to target the player
    int goalCol = (gp.player.worldX + gp.player.solidArea.x)/gp.tileSize;
    int goalRow = (gp.player.worldY + gp.player.solidArea.y)/gp.tileSize;
    searchPath(goalCol, goalRow);

    // Increase speed instead of shooting projectiles
    speed = 3;

}
```

The two mobs that I created in this game are the normal spider and the spider that drops the key: the key spider. At first glance, they may look very similar to one another, however the key spider behaves differently from the normal spider. A normal spider would shoot projectiles and chase the player down using A\* pathfinding if a player gets close, while the key spider gains

speed and tries to come in contact with the player to damage them. These actions are all set in the `setAction()` method inherited from the entity class. Other than that, the key spider has a higher hitpoint compared to the normal spider.

Now that the entity class and mobs are explained, I will explain about the pathfinding algorithm I chose for this game. For this game, I chose to use the A\* pathfinding algorithm. It relies on values called g, h, and f costs, which are the distance between start node and current node, distance between current node and goal node, and the total cost of  $G + H$  respectively, where nodes are tiles in the game. By prioritizing the lowest f-costs, the algorithm efficiently finds the optimal path to the destination. If two or more available nodes have the same f-cost, their g-costs will be compared and the lower one will be more efficient.

The classes involved in implementing this algorithm in my code are the Node class, Pathfinder class, and Entity class. Firstly, the Node class is a class that defines a node, giving it the required attributes such as the g, h, and f costs, solid status, open, and checked. This node class is implemented in the Pathfinder class. After instantiating the Pathfinder class in the GamePanel, the nodes will be instantiated as well, based on the world map's dimensions. In the Entity class's `searchPath` method, the `setNodes` method of the Pathfinder class is called, giving each node their solid status and their costs. After that, the main pathfinding method will be called, which is the `search` method.

## Instantiating Nodes

```
public void instantiateNodes(){
    // Initialize the 2D array of Node objects with dimensions based on the world columns and rows
    node = new Node[gp.maxWorldCol][gp.maxWorldRow];

    // Variables to keep track of the current column and row
    int col = 0;
    int row = 0;

    // Loop through all columns and rows to instantiate each Node object
    while(col < gp.maxWorldCol && row < gp.maxWorldRow){
        // Create a new Node at the current column and row
        node[col][row] = new Node(col, row);

        // Move to the next column
        col++;

        // If the end of the current row is reached, reset column to 0 and move to the next row
        if(col == gp.maxWorldCol){
            col = 0;
            row++;
        }
    }
}
```

## Getting the cost of a node

```
// Finding the G, H, and F costs of the nodes
// usage  Adrian
private void getCost(Node node) {

    // Calculate the gCost (distance from the start node to the current node)
    int x = Math.abs(node.col - startNode.col); // Horizontal distance to start node
    int y = Math.abs(node.row - startNode.row); // Vertical distance to start node
    node.gCost = x + y;

    // Calculate the hCost (heuristic estimate from the current node to the goal node)
    x = Math.abs(node.col - goalNode.col); // Horizontal distance to goal node
    y = Math.abs(node.row - goalNode.row); // Vertical distance to goal node
    node.hCost = x + y;

    // Calculate the fCost which is the sum of G and H costs
    node.fCost = node.gCost + node.hCost;
}
```

## Setting Nodes

```
// Looping through all columns and rows
while(col < gp.maxWorldCol && row < gp.maxWorldRow){

    // Get the tile number at the current position
    int tileNum = gp.tileM.mapTileNum[col][row];

    // Checking for solid tiles
    if(gp.tileM.tile[tileNum].collision){
        node[col][row].solid = true;
    }

    // Calculate the cost for each node
    getCost(node[col][row]);

    // Move to the next column
    col++;

    // If the end of the current row is reached, reset column to 0 and move to the next row
    if(col == gp.maxWorldCol){
        col = 0;
        row++;
    }
}
```

When the search method is called, it will continue to search for the goal node until that node is reached or until a maximum number of steps is reached by using a while loop. In this loop, it will set the checked value for the current node, starting from the start node, to true and remove that node from the list of open nodes. It will then open the nodes adjacent to it and loop through the list of open nodes to compare their fCost or gCost values. The current node will be set to the node from the list of open nodes with the lowest fCost or gCost. Once the current node is the goal node, the search will be finished and backtrack the nodes with the lowest cost from the goal node all the way to the start node. This method will then return the finished value (true).



## Marking current node as checked and opening surrounding nodes

```
// Initialize the column and row variables
int col = currentNode.col;
int row = currentNode.row;

// Mark the current node as checked and remove it from the open list
currentNode.checked = true;
openList.remove(currentNode);

// Open adjacent nodes (up, left, down, right) if within bounds

if (row-1>=0){openNode(node[col][row-1]);} // Open the node above

if (col-1 >= 0){ openNode(node[col-1][row]);} // Open the node to the left

if (row+1<gp.maxWorldRow){ openNode(node[col][row+1]);} // Open the node below

if (col+1 <gp.maxWorldCol){ openNode(node[col+1][row]);} // Open the node to the right
```

## Finding the next node to go to (lowest fCost)

```
// Iterate through the open list to find the node with the lowest fCost
for(int i = 0; i < openList.size(); i++){

    if(openList.get(i).fCost < bestNodefCost){
        bestNodeIndex = i;
        bestNodefCost = openList.get(i).fCost;
    }

    // If all fCost are the same, compare the gCost
    else if(openList.get(i).fCost == bestNodefCost){
        if(openList.get(i).gCost < openList.get(bestNodeIndex).gCost){
            bestNodeIndex = i;
        }
    }
}
}
```

## Setting current node as node with lowest cost and if it is the goal node, activate backtracking all the way to the start node

```
// Set the current node to the best node found
currentNode = openList.get(bestNodeIndex);

// If the goal node is reached, mark as finished and backtrack to create the path
if(currentNode == goalNode){
    finished = true;
    backTrackingPath();
}
```

The value returned by the Search method is used by the SearchPath method from the Entity class. If the value is true, the mob will follow the path provided by the search algorithm and move towards the player. Once the mob finally reaches the goal node (the player), it will deactivate the pathfinding. That is how it would work by default, however I programmed the

mob to always chase the player when the player gets within a 5 block radius from the mob, so the pathfinding will never stop unless the player moves around 10 blocks away from the mob.

### A portion of the searchPath method

```
if(gp.pathFinder.search()){
    // The coordinates it is going to move to
    int newX = gp.pathFinder.pathList.get(0).col * gp.tileSize;
    int newY = gp.pathFinder.pathList.get(0).row * gp.tileSize;

    // Entity's hit-box position
    int entityLeftX = worldX + solidArea.x;
    int entityRightX = worldX + solidArea.x + solidArea.width;
    int entityTopY = worldY + solidArea.y;
    int entityBottomY = worldY + solidArea.y + solidArea.height;

    // Based on Entity position, find the relative direction of next tile

    if(entityTopY > newY && entityLeftX >= newX && entityRightX < newX + gp.tileSize){
        // Entity is above the newY coordinate and horizontally aligned with newX.
        direction = "up";
    }

    else if(entityTopY < newY && entityLeftX >= newX && entityRightX < newX + gp.tileSize){
        // Entity is below the newY coordinate and horizontally aligned with newX.
        direction = "down";
    }
}
```

### Other Classes

Aside from the aforementioned classes and methods used to implement the more important features, there are also some other small classes that are also essential for the game.

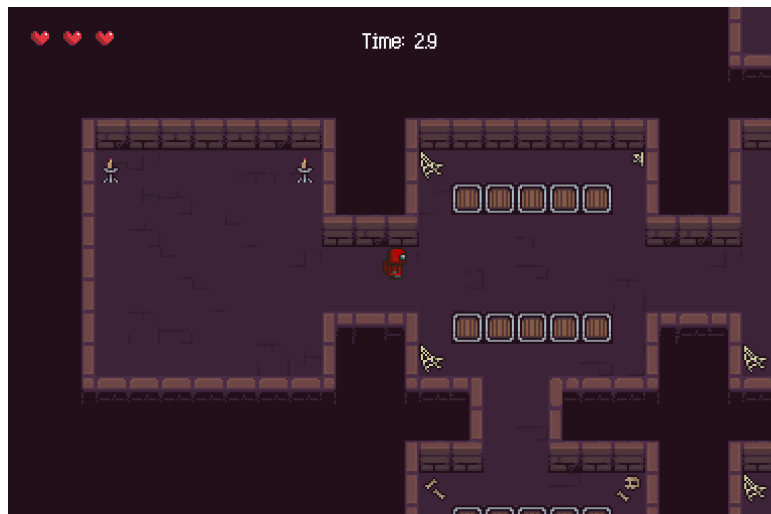
Firstly, there is the UtilityTool class which is responsible for scaling the images based on a specified width and height, making it either larger or smaller than the original dimensions. Other than that, there is the Sound class which is responsible for storing the audio, playing audio files, stopping the audio files, and adjusting the volume (muting and unmuting). Next, the KeyHandler class is a class that is used by the Player and GamePanel class to set certain keys to do certain actions, such as movement or menu navigation.

## Screenshots

Title Screen



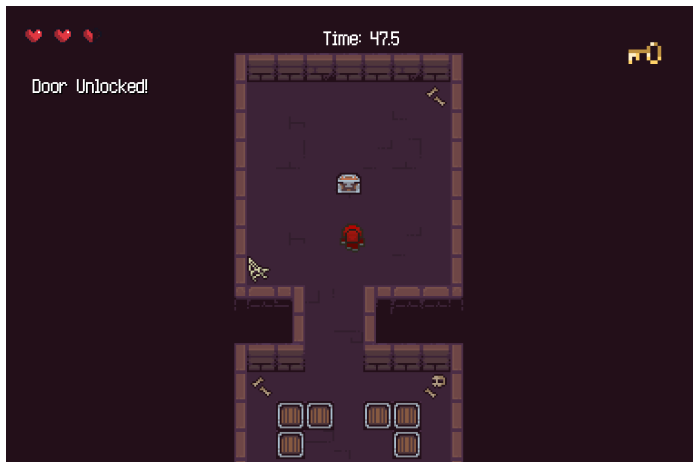
Spawn Area



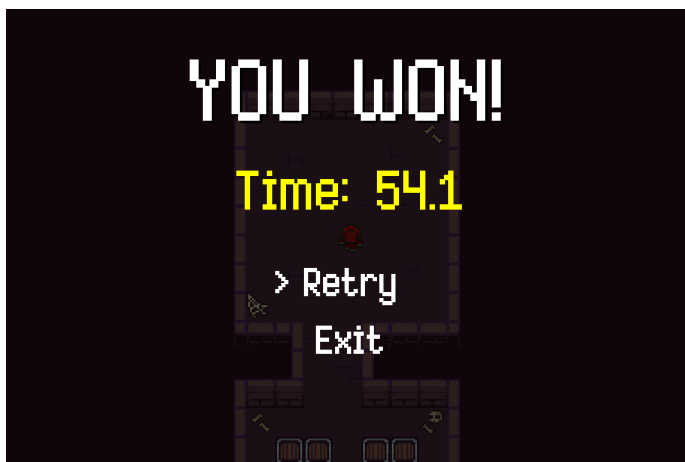
## Key Room



## Treasure Room



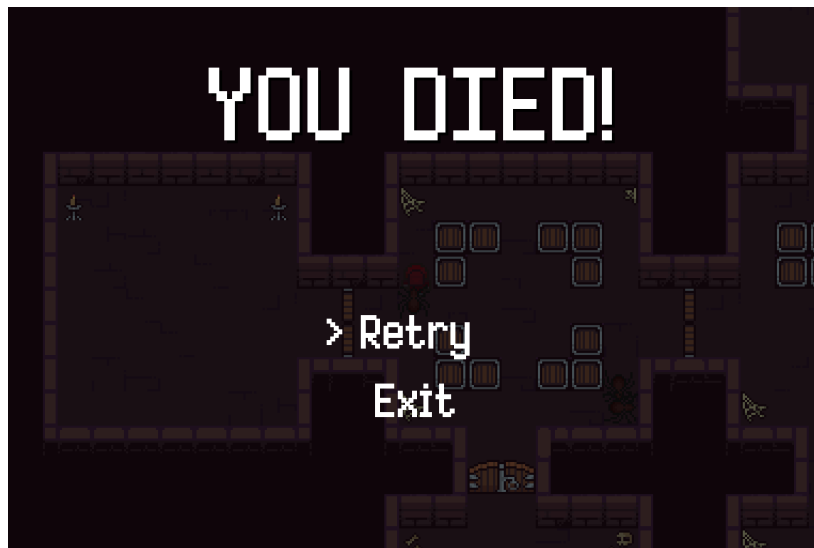
## Win Screen



## Pause Screen



## Death (Game Over) Screen



## Resources

Guide for creating the game: <https://www.youtube.com/@RyiSnow>

Tileset for the map: <https://pixel-poem.itch.io/dungeon-assetpuck>

UML Class Diagram:

- IntelliJ Ultimate
- <https://app.diagrams.net/>