Vietnamese National University HCM

HCMC International University

School of Computer Science and Engineering

PROJECT REPORT

TOPIC: 2D GAMES

ADVENTURES OF PLAYER

Course: OBJECT – ORIENTED PROGRAMMING

ID: IT069IU

Instructor: *Dr. Tran Thanh Tung*

MSc. Nguyen Trung Nghia

Table of Contents

CONTRIBUTION TABLE

ID	NAME	CONTRIBUTION
ITDSIU23030	Trần Nam Anh	100%
ITDSIU23005	Ngô Thị Ánh Dương	100%
ITDSIU23006	Nguyễn Đức Hải	100%
ITDSIU23017	Đặng Minh Phát	90%

Github – Code: link

CHAPTER 1: INTRODUCTION

1. Abstract

This project proposes the development of an adventure-based video game designed to position the player as a discoverer, thereby fostering an environment conducive to exploration and intellectual engagement. The game will feature a series of interactive scenarios and challenges that entertain and encourage the development of creative thinking, problem-solving abilities, and cognitive flexibility. By integrating elements of narrative-driven gameplay with opportunities for autonomous discovery, the project seeks to create a balanced experience that is both mentally stimulating and emotionally relaxing. Ultimately, this game aims to contribute to the growing body of research supporting the use of interactive digital media to enhance cognitive and emotional well-being in an accessible and engaging format.

Furthermore, this project allows team members to deepen their understanding of core object-oriented programming (OOP) concepts, including inheritance, encapsulation, and polymorphism. In addition, participants will explore essential software design principles, particularly the SOLID principles. Through the practical application of these concepts, team members will enhance

their theoretical knowledge and ability to develop robust, maintainable, and scalable software systems.

2. Project objectives:

This project follow these main objectives:

Creating a creative, thought-provoking and engaging game for players:

By experience, different game maps allow users to use their creativity and logic to win the map.

Apply the knowledge of the Object-Oriented Programming course to the real-life scenarios:

Apply the basic concepts of OOP like inheritance, polymorphism, encapsulation and abstraction.

Apply the SOLID principles in real-world cases to gain more experience applying them reasonably.

Learn to collaborate and discover the popular tool for program collaboration, such as GitHub.

The team needs to manage the tasks in a balanced way for each member, who can learn how to express their opinions to improve the team's results.

By implementing this project, each member learn how to use git and GitHub, which is the most popular tool for collaboration.

3. Techniques and tools used

Programming Language: Java

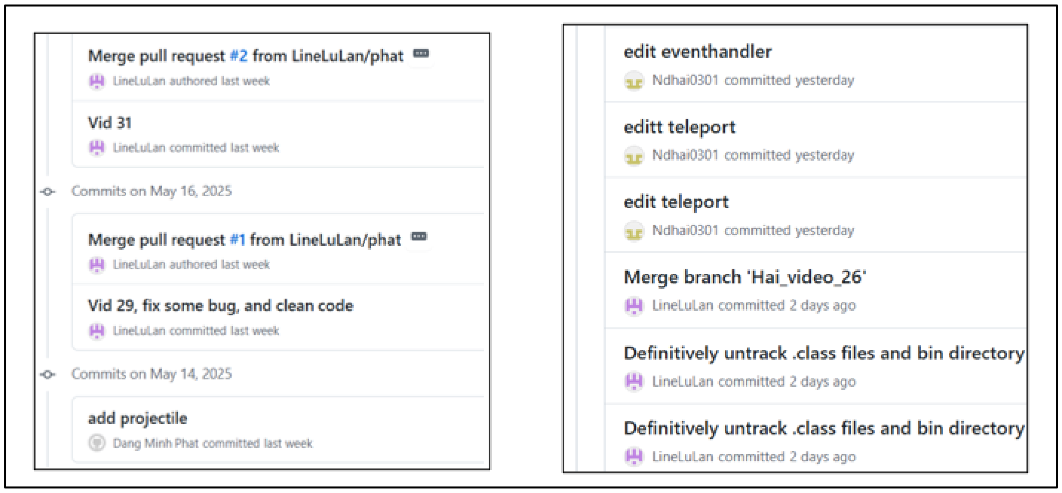
The robust language for supporting the application of OOP.

IDE - Integrated Development Environment: Eclipse, IntelliJ, VSCode.

Version control and collaboration: Git and GitHub

Git: used to track all versions of change.

GitHub: used for creating, storing, managing and sharing code.


Merge pull request #2 from LineLuLan/phat
LineLuLan authored last week
Vid 31
LineLuLan committed last week
Commits on May 16, 2025
Merge pull request #1 from LineLuLan/phat
LineLuLan authored last week
Vid 29, fix some bug, and clean code
LineLuLan committed last week
Commits on May 14, 2025
add projectile
Dang Minh Phat committed last week

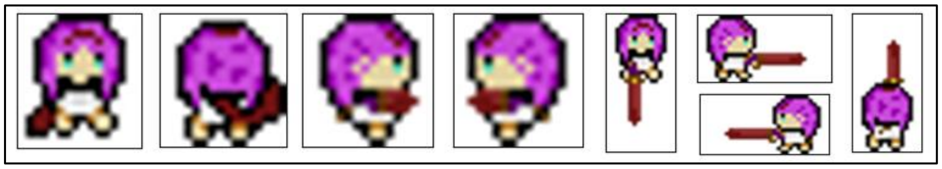
edit eventhandler
Ndhai0301 committed yesterday
editt teleport
Ndhai0301 committed yesterday
edit teleport
Ndhai0301 committed yesterday
Merge branch 'Hai_video_26'
LineLuLan committed 2 days ago
Definitively untrack .class files and bin directory
LineLuLan committed 2 days ago
Definitively untrack .class files and bin directory
LineLuLan committed 2 days ago

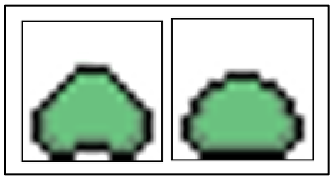
GitHub

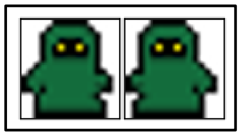
Chapter 2: Game Description

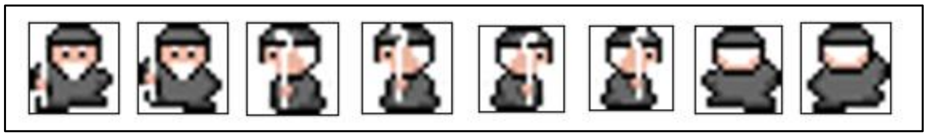
2.1 Design

2.1.1 Character

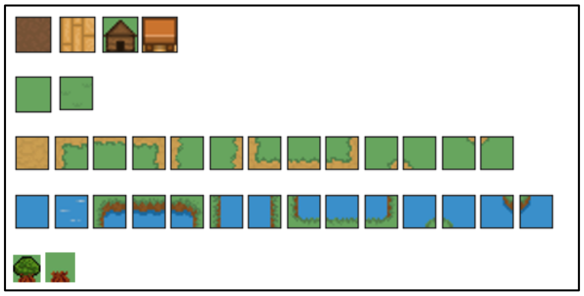

Player

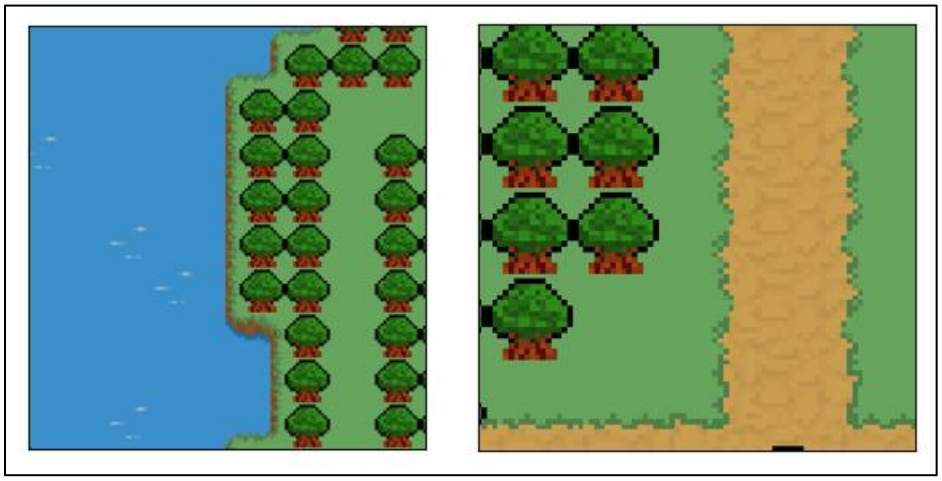

Monster


Merchant


Old man

2.1.2 Background design

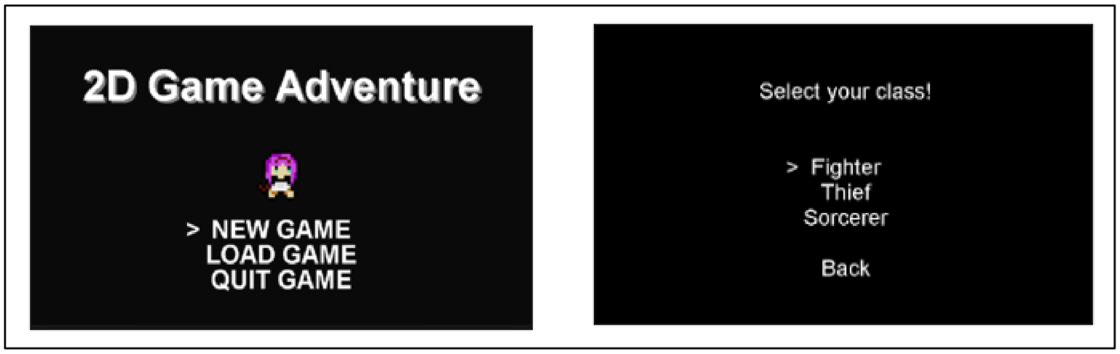

Tiles


Maps

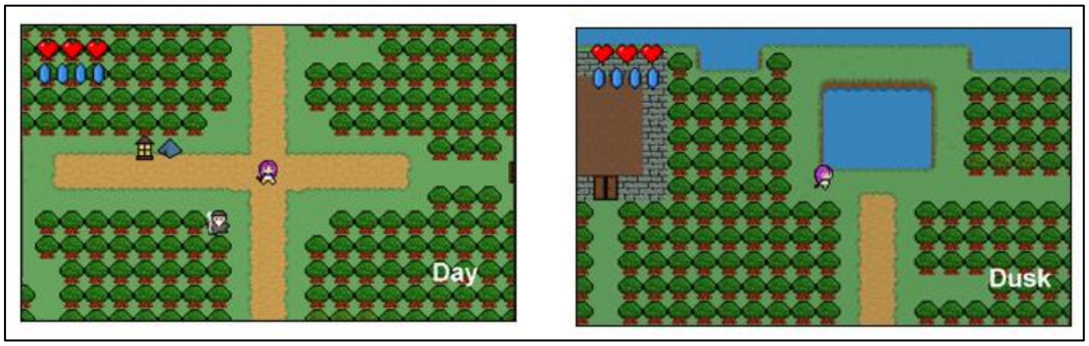
Objects

Objects interaction

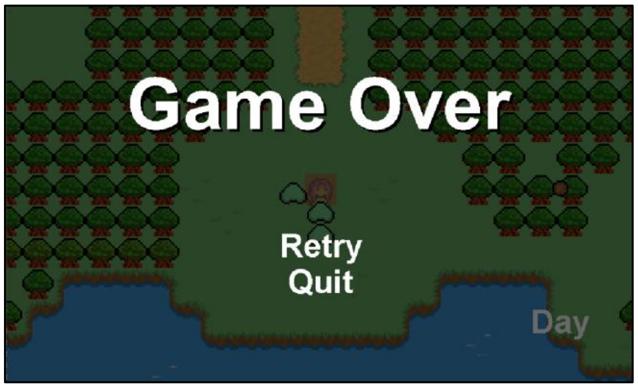
2.1.3 Window designs



Menu



Play status

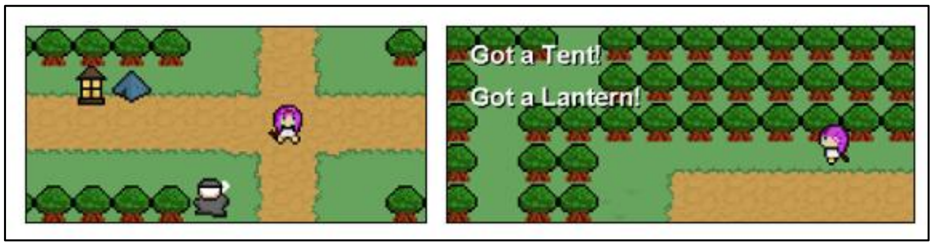


End game

2.2 Game rule

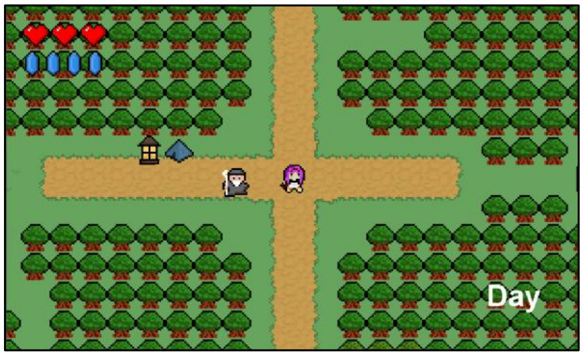
Here is all of the basic guide and rules of the game .

In the hunting game, players will take the role of an adorable girl to start a journey of training and upgrading herself by taking some valuable items and hitting the monsters to gain experience.



Take useful items

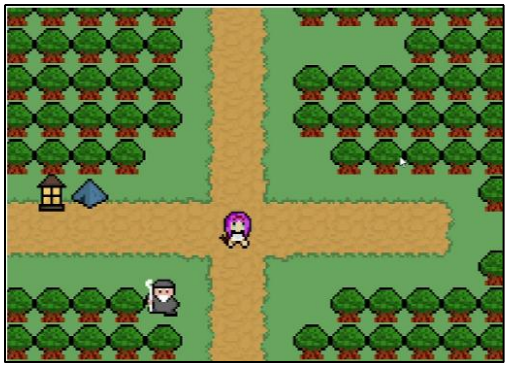
The game starts at the centre of the play map. Then, players have to find monsters to level up and finally beat the boss to end the game. Before that, you have to choose your character's role, including Fighter, Thief, or Sorcerer.



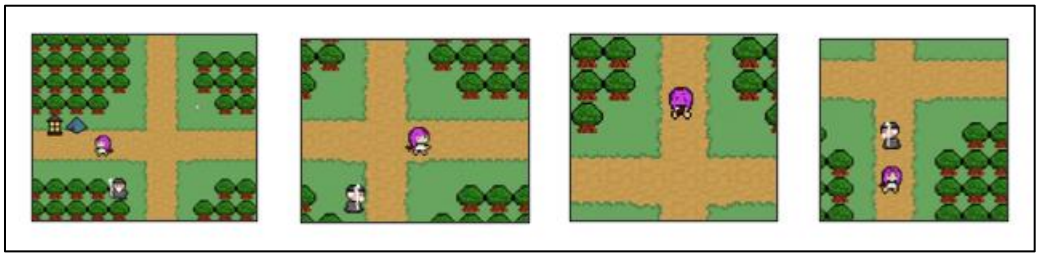
Starting game

Combat and Monster Hunting system:

Players controls character by using keyboara W – right, A – up, S – down, D - left to move around the map.

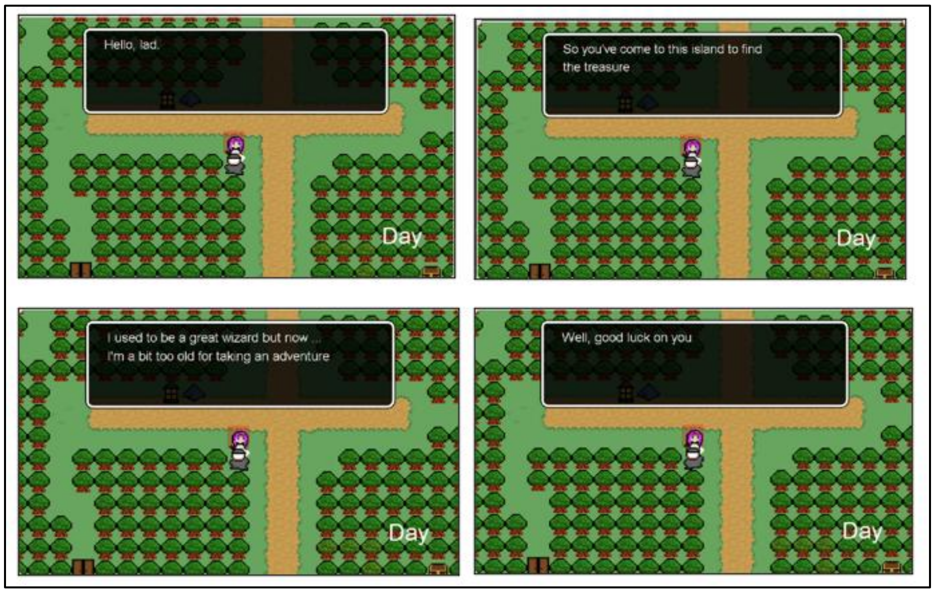


Original direction



Movement

When encountering monsters, or any type of objects in Game, players can use Enter to interact with these objects like hit the monsters, talk with NPC, or use Healing Pool.



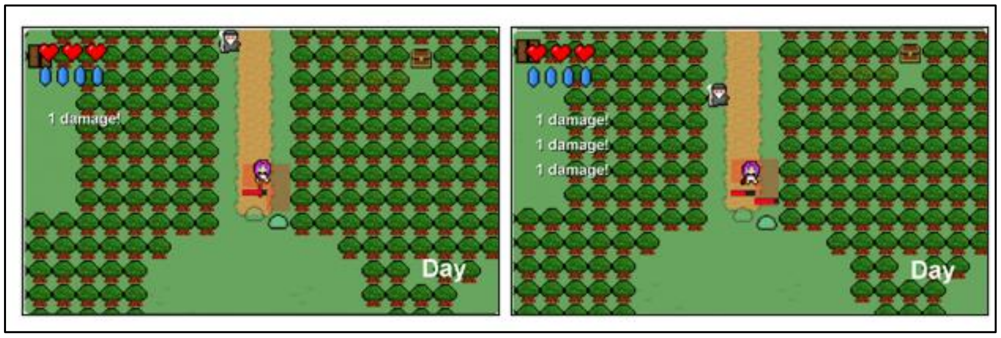
Hello, lad.

So you've come to this island to find the treasure

I used to be a great wizard but now ...
I'm a bit too old for taking an adventure

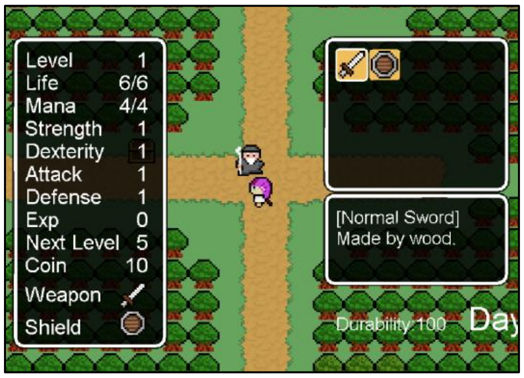
Well, good luck on you

Talk with NPC – Old man



Hit the monster

Moreover, players can press some buttons to use some game function like the character storage, setting game, or pausing game.

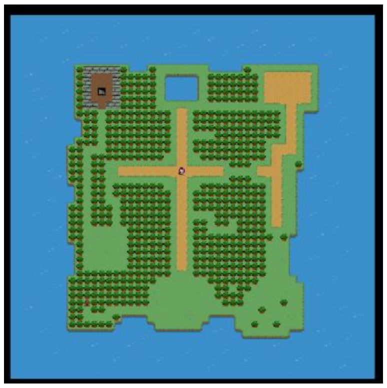


Level	1
Life	6/6
Mana	4/4
Strength	1
Dexterity	1
Attack	1
Defense	1
Exp	0
Next Level	5
Coin	10
Weapon	
Shield	

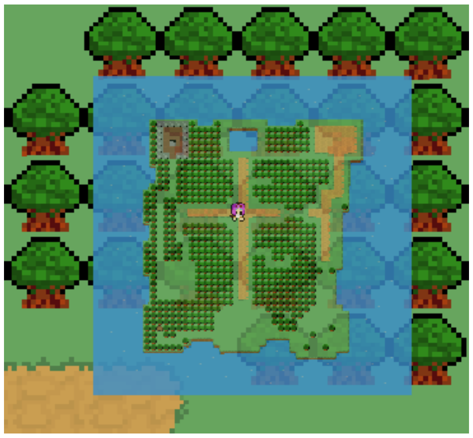
[Normal Sword]
Made by wood.

Durability 100

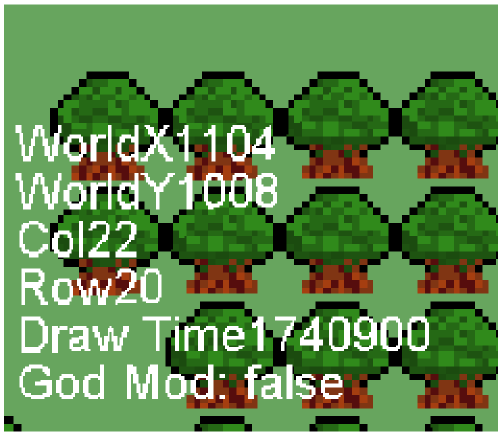
C - character's status



M – whole map



X – mini map

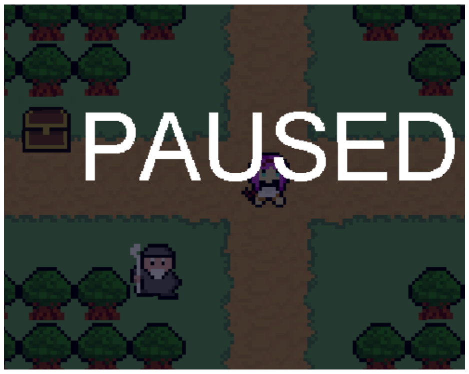


```
WorldX1104
WorldY1008
Col22
Row20
Draw Time1740900
God Mod: false
```

T - location

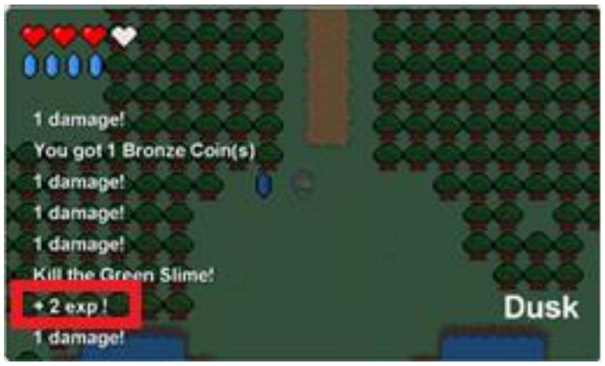
Space – use shield

Enter – use sword



P - pause

- Defeat monster, player will gain experience (EXP) and a random item,

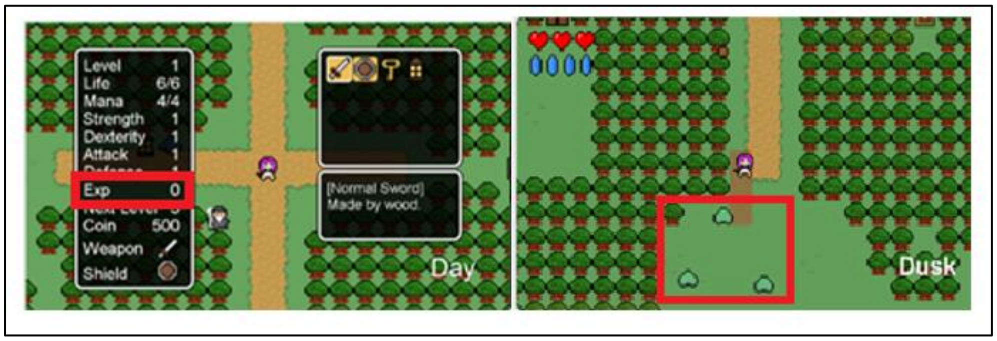


```
1 damage!
You got 1 Bronze Coin(s)
1 damage!
1 damage!
1 damage!
Kill the Green Slime!
+ 2 exp !
1 damage!
```

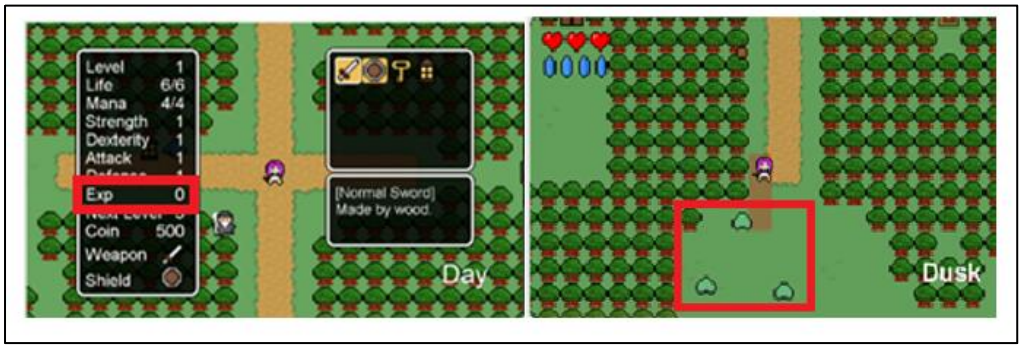
Gain experiences

- Levelling and EXP system:
  - Once the exp by defeating monsters, the character's level will level up automatically.

When levelling up, the player will have higher character stats (HP, attack, defence, etc.)



Exp before defeating the monster

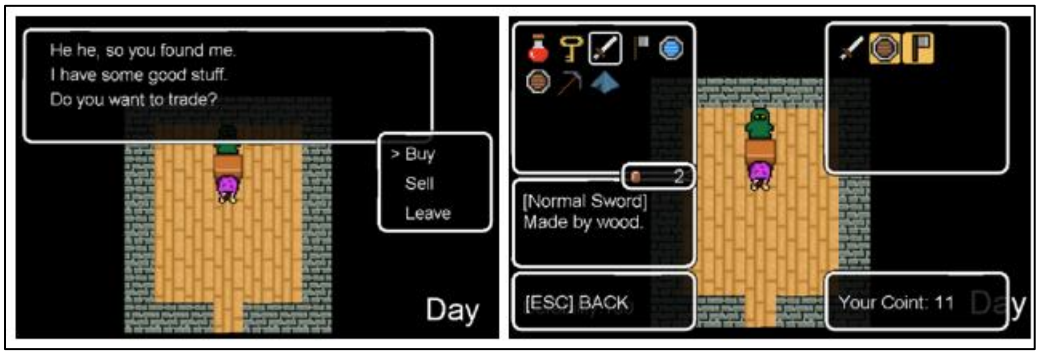


Exp after defeating the monster

Item exchange system:

Players can sell received items to NPCs to get coins and buy valuable items at a list price in the system.

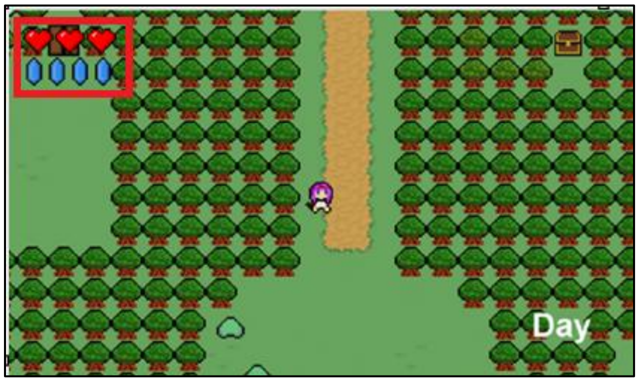
The price of the item the player sells will be lower than that of the item the player buys from the NPC.


He he, so you found me.
I have some good stuff.
Do you want to trade?

Item exchange system

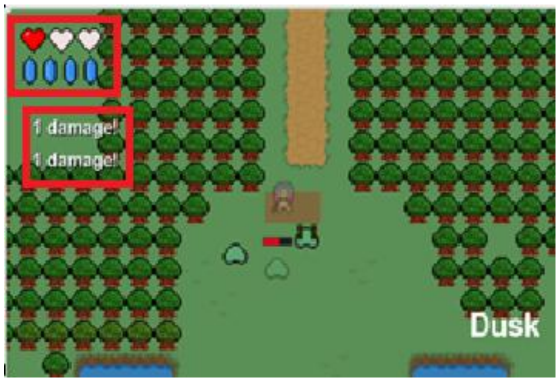
Health System and Game Over Condition

The character has a visible Health Bar (HP) displayed on the screen.



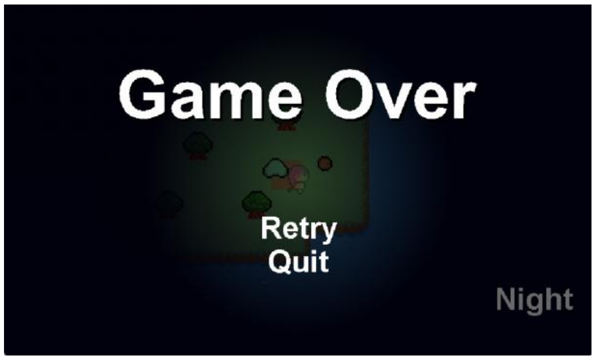
Visible health bar

When the character takes damage from monsters, their HP will decrease.



HB when getting damage

If the HP reaches zero, the player will die and the game will end immediately (Game Over).Upon death, players must restart the game from the beginning.



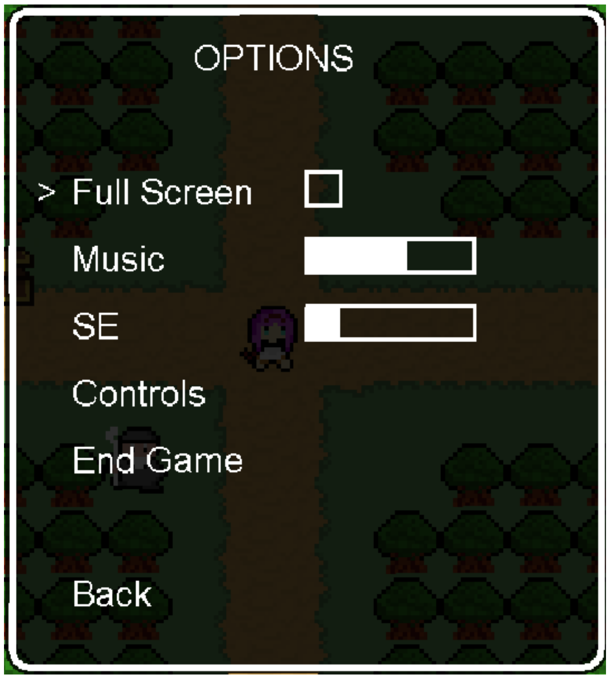
Game over

Setting the game system

The player can adjust the game's volume and music to fix the individual's interest.

The side of the game's window can be full screen or not

Players can stop the game and save it. One index, and everything will be saved at the time the player saves the game.

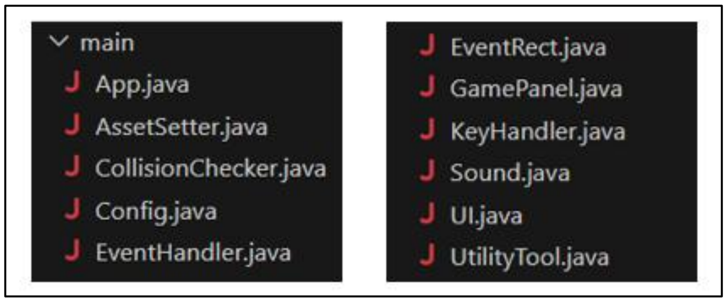


Game system

2.3 How the core game work

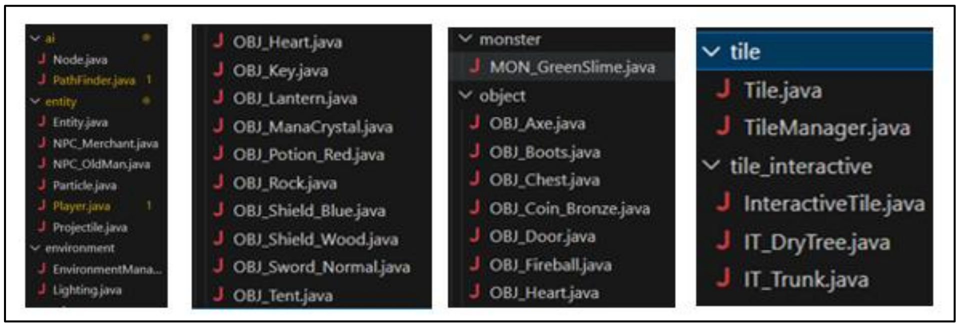
2.3.1 Packages

- Several packages are contributing to the final result. However, we will focus mainly on the main package to understand how the project works.



```
main
  App.java
  AssetSetter.java
  CollisionChecker.java
  Config.java
  EventHandler.java
  EventRect.java
  GamePanel.java
  KeyHandler.java
  Sound.java
  UI.java
  UtilityTool.java
```

Main package



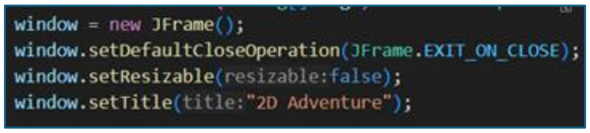
```
ai
  Node.java
  PathFinder.java  1
entity
  Entity.java
  NPC_Merchant.java
  NPC_OldMan.java
  Particle.java
  Player.java  1
  Projectile.java
environment
  EnvironmentMana...
  Lighting.java

  OBJ_Heart.java
  OBJ_Key.java
  OBJ_Lantern.java
  OBJ_ManaCrystal.java
  OBJ_Potion_Red.java
  OBJ_Rock.java
  OBJ_Shield_Blue.java
  OBJ_Shield_Wood.java
  OBJ_Sword_Normal.java
  OBJ_Tent.java

monster
  MON_GreenSlime.java
object
  OBJ_Axe.java
  OBJ_Boots.java
  OBJ_Chest.java
  OBJ_Coin_Bronze.java
  OBJ_Door.java
  OBJ_Fireball.java
  OBJ_Heart.java

tile
  Tile.java
  TileManager.java
tile_interactive
  InteractiveTile.java
  IT_DryTree.java
  IT_Trunk.java
```

Other packages

2.3.2 App class and Gamepanel class

2.3.2.1 Set up the window

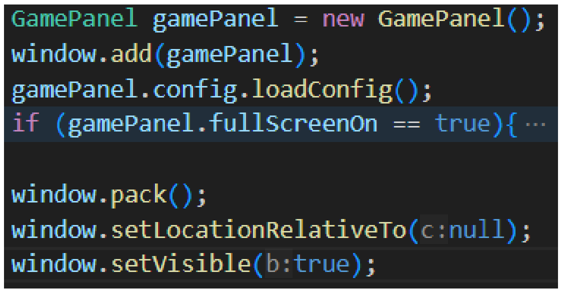
- First, set up the window.



```java
window = new JFrame();
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
window.setResizable(resizable:false);
window.setTitle(title:"2D Adventure");
```

Window set up

- Second, add the GamePanel into the window, and set up some mode(full screen, screen centre in the window, visible screen)

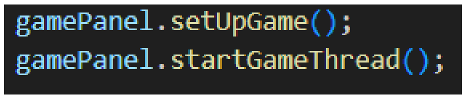


```
GamePanel gamePanel = new GamePanel();
window.add(gamePanel);
gamePanel.config.loadConfig();
if (gamePanel.fullScreenOn == true){ ···

window.pack();
window.setLocationRelativeTo(c:null);
window.setVisible(b:true);
```

Add game panel

2.3.2.2 GamePanel

- First, inside the app class, we will call two functions of GamePanel.



```
gamePanel.setUpGame();
gamePanel.startGameThread();
```

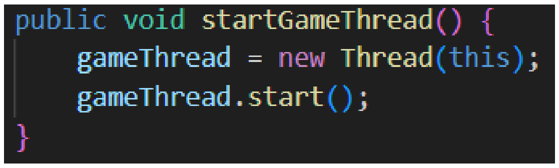
App class

The setUpGame will set the screen when we turn the game on.

The startGameThread will update all the changes over time.

- Second, discover the startGameThread, as well as the GamePanel class.

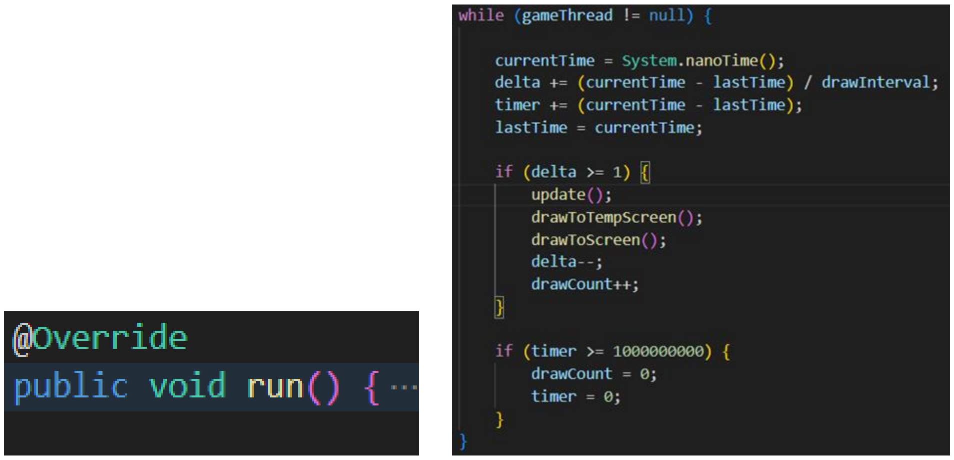
Inside the startGameThread, we implement the method start.
This method will call the run method.



```
public void startGameThread() {
    gameThread = new Thread(this);
    gameThread.start();
}
```

startGameThread method

- Then, we will decorate the run method using the Override method so the game loop is executed at a fixed frame rate—60 FPS or 60 times per second.



```
@Override
public void run() {···
```

```
while (gameThread != null) {

    currentTime = System.nanoTime();
    delta += (currentTime - lastTime) / drawInterval;
    timer += (currentTime - lastTime);
    lastTime = currentTime;

    if (delta >= 1) {
        update();
        drawToTempScreen();
        drawToScreen();
        delta--;
        drawCount++;
    }

    if (timer >= 1000000000) {
        drawCount = 0;
        timer = 0;
    }
}
```

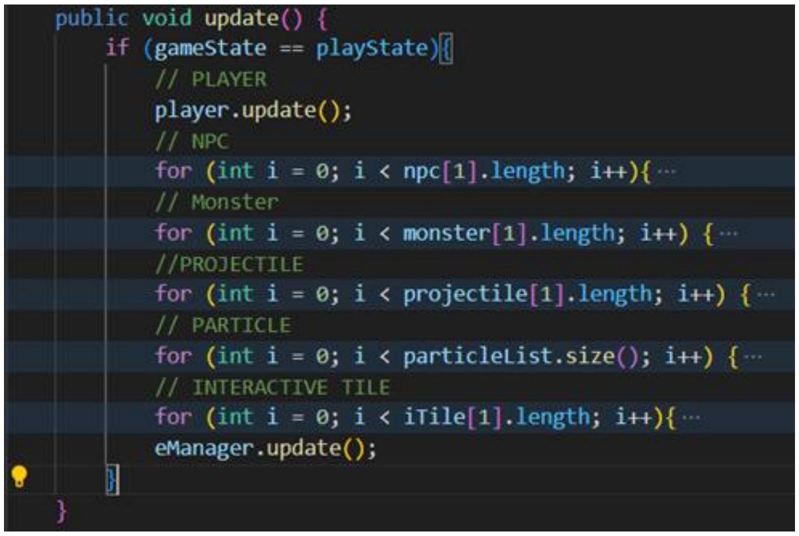
Run method

Inside the game loop, we call the update, drawToTempScreen, and drawToScreen methods when delta >= 1. This means that the game will execute these methods once every 60 FPS.

Discovering the update method:

Inside the update method, we will update all the objects that we want to exist in our games.

For example, the main character, like a player, or another character, like an NPC or monster, or updating the objects like a projectile, particle, or interactive tile.

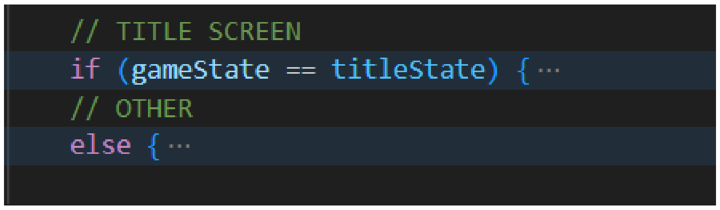


```
public void update() {
    if (gameState == playState){
        // PLAYER
        player.update();
        // NPC
        for (int i = 0; i < npc[1].length; i++){···
        // Monster
        for (int i = 0; i < monster[1].length; i++) {···
        //PROJECTILE
        for (int i = 0; i < projectile[1].length; i++) {···
        // PARTICLE
        for (int i = 0; i < particleList.size(); i++) {···
        // INTERACTIVE TILE
        for (int i = 0; i < iTile[1].length; i++){···
        eManager.update();
    }
}
```

update method

Discovering the drawToTempScreen method

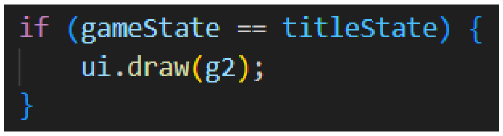
Inside this method, we have two modes: titleState or menu, else or play state.



```
// TITLE SCREEN
if (gameState == titleState) {···
// OTHER
else {···
```

drawToTempScreen method

If the gameState is in the titleState, then we will take the ui object to draw the menu

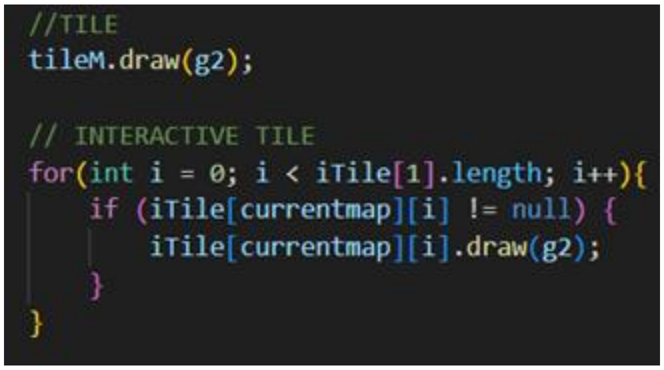


```
if (gameState == titleState) {
    ui.draw(g2);
}
```

titleState mode

If the gameState is in the playState, then we will draw in the following order:

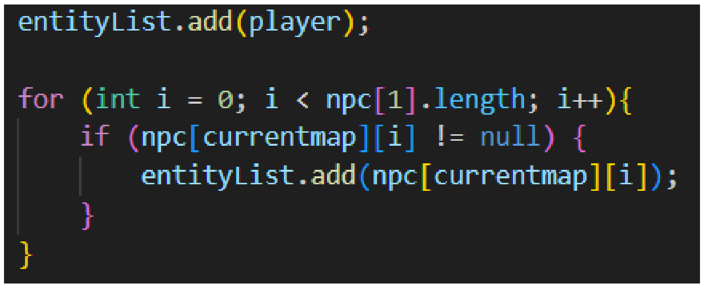
Tiles or maps of the game:



```
//TILE
tileM.draw(g2);

// INTERACTIVE TILE
for(int i = 0; i < iTile[1].length; i++){
    if (iTile[currentmap][i] != null) {
        iTile[currentmap][i].draw(g2);
    }
}
```

Drawing tiles

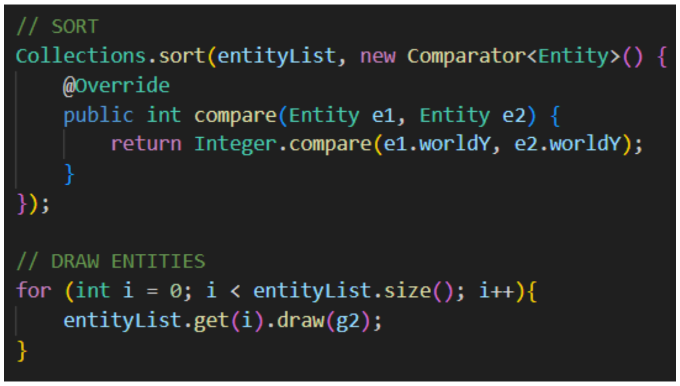
Then we will take all the updated objects, such as player or NPC, that are updated in the update method, to store in the entityList:



```
entityList.add(player);

for (int i = 0; i < npc[1].length; i++){
    if (npc[currentmap][i] != null) {
        entityList.add(npc[currentmap][i]);
    }
}
```

Store entities

Then we will sort the list in the order that we want to draw and then draw them:



```
// SORT
Collections.sort(entityList, new Comparator<Entity>() {
    @Override
    public int compare(Entity e1, Entity e2) {
        return Integer.compare(e1.worldY, e2.worldY);
    }
});

// DRAW ENTITIES
for (int i = 0; i < entityList.size(); i++){
    entityList.get(i).draw(g2);
}
```

Sort and draw entites

2.3.3 Entity and Player

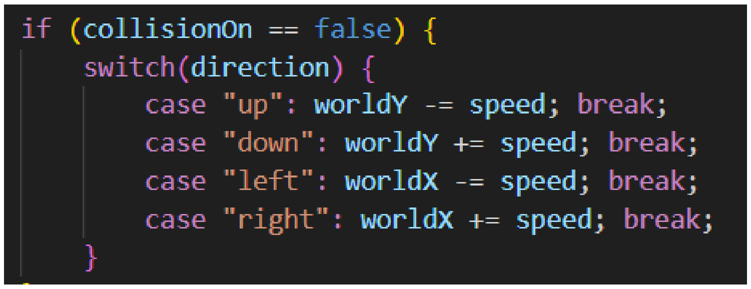
The entity is the superclass that all objects in the game state will inherit, such as player, oldman—NPC, monster, etc. The player is the most representative class.

2.3.3.1 Entity

Even though there are multiple methods inside the entity function. There are two main core functions – update and draw, which are also called in the GamePanel.

A) Update

The main update:

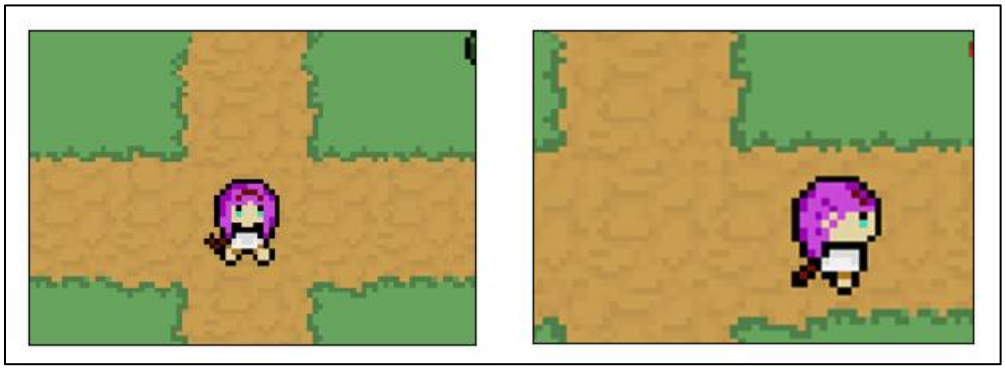


```
if (collisionOn == false) {
    switch(direction) {
        case "up": worldY -= speed; break;
        case "down": worldY += speed; break;
        case "left": worldX -= speed; break;
        case "right": worldX += speed; break;
    }
```

Entites update

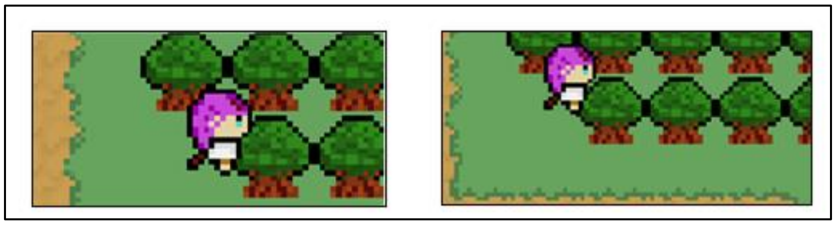
Explanation:

collisionOn is false, which means that the player can go through. For example, the brown object is a road, which is the object that the player can pass through, so collisionOn is off. And in this case, we also update the new place of the player by worldX and worldY.



Collision is off

However, in this case, collisionOn is true, so the player can not pass through this object.

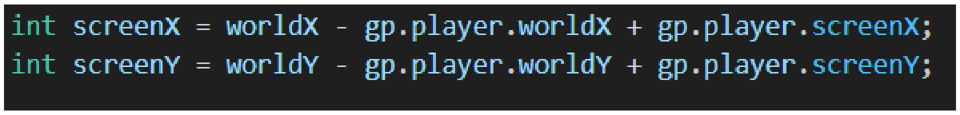


Collision is on

B) Draw

This method has two main functions. The first concerns where to display the object, and the second concerns what the image should be.

Firstly, choosing the place to display the object.



```java
int screenX = worldX - gp.player.worldX + gp.player.screenX;
int screenY = worldY - gp.player.worldY + gp.player.screenY;
```

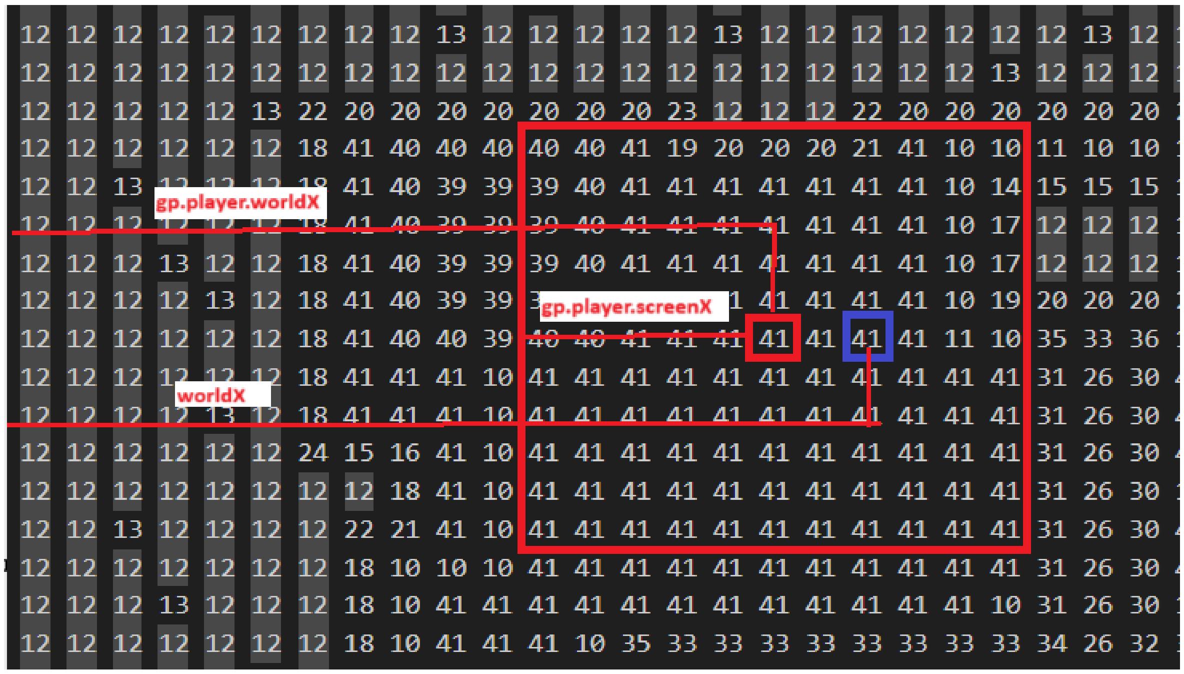
Mapping formulars

Explanation:

The whole map is big, and we do not want to show it on the screen; we just show the big red square.

Then we will take the leading player appearing in the centre of the screen.

So, we will take the player as a milestone when any new object is drawn.



```
12 12 12 12 12 12 12 12 12 13 12 12 12 12 12 13 12 12 12 12 12 12 12 13 12
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 13 12 12 12
12 12 12 12 12 13 22 20 20 20 20 20 20 20 23 12 12 12 22 20 20 20 20 20 20
12 12 12 12 12 12 18 41 40 40 40 40 40 41 19 20 20 20 21 41 10 10 11 10 10
12 12 13 12 12 12 18 41 40 39 39 39 40 41 41 41 41 41 41 41 10 14 15 15 15
12 12 12 12 12 12 18 41 40 39 39 39 40 41 41 41 41 41 41 41 10 17 12 12 12
12 12 12 13 12 12 18 41 40 39 39 39 40 41 41 41 41 41 41 41 10 17 12 12 12
12 12 12 12 13 12 18 41 40 39 39 3[illegible] 1 41 41 41 41 10 19 20 20 20
12 12 12 12 12 12 18 41 40 40 39 40 40 41 41 41 41 41 41 41 11 10 35 33 36
12 12 12 12 12 12 18 41 41 41 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 12 12 13 12 18 41 41 41 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 12 12 12 12 24 15 16 41 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 12 12 12 12 12 12 18 41 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 13 12 12 12 12 22 21 41 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 12 12 12 12 12 18 10 10 10 41 41 41 41 41 41 41 41 41 41 41 31 26 30
12 12 12 13 12 12 12 18 10 41 41 41 41 41 41 41 41 41 41 41 41 10 31 26 30
12 12 12 12 12 12 12 18 10 41 41 41 10 35 33 33 33 33 33 33 33 33 34 26 32
```

Mapping illustration

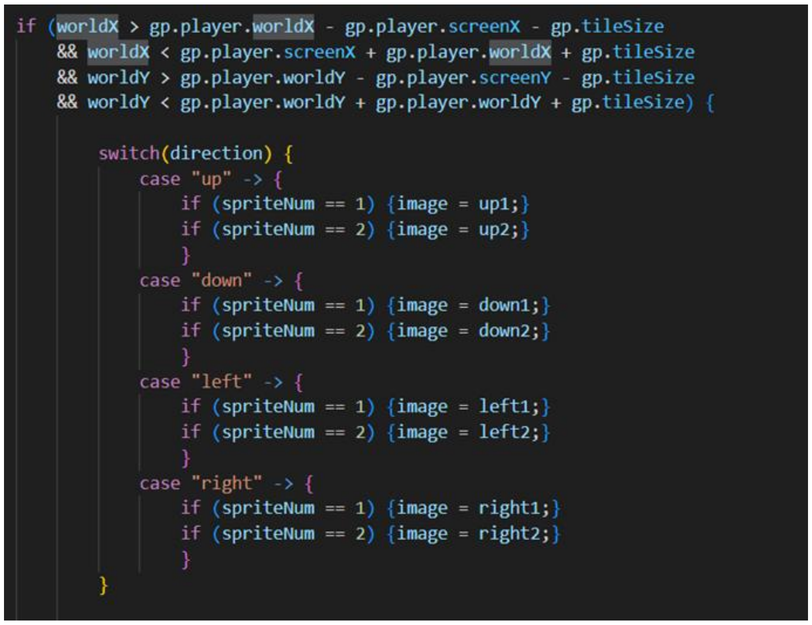
- Secondly, choosing the picture to draw:

The condition in the if statement means that if the object is inside the red square (picture mappings), we will update the image.

Then, we will assign the corresponding image for suitable cases.

For example, the Image of left1 and right1 of the player.

Updated images



```
if (worldX > gp.player.worldX - gp.player.screenX - gp.tileSize
    && worldX < gp.player.screenX + gp.player.worldX + gp.tileSize
    && worldY > gp.player.worldY - gp.player.screenY - gp.tileSize
    && worldY < gp.player.worldY + gp.player.worldY + gp.tileSize) {

        switch(direction) {
            case "up" -> {
                if (spriteNum == 1) {image = up1;}
                if (spriteNum == 2) {image = up2;}
                }
            case "down" -> {
                if (spriteNum == 1) {image = down1;}
                if (spriteNum == 2) {image = down2;}
                }
            case "left" -> {
                if (spriteNum == 1) {image = left1;}
                if (spriteNum == 2) {image = left2;}
                }
            case "right" -> {
                if (spriteNum == 1) {image = right1;}
                if (spriteNum == 2) {image = right2;}
                }
        }
```

Updated codes

2.3.3.2 Player

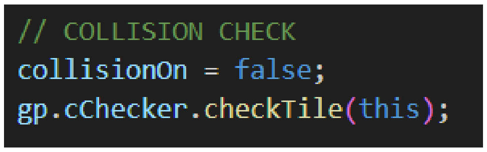
- By discovering the player class, we can understand how players interact with other objects.

- Dividing the interaction into two modes - attacking and unattacking

A) Unattacking mode

Main Steps of unattacking interaction mode:

1. Check if this object the player can pass through

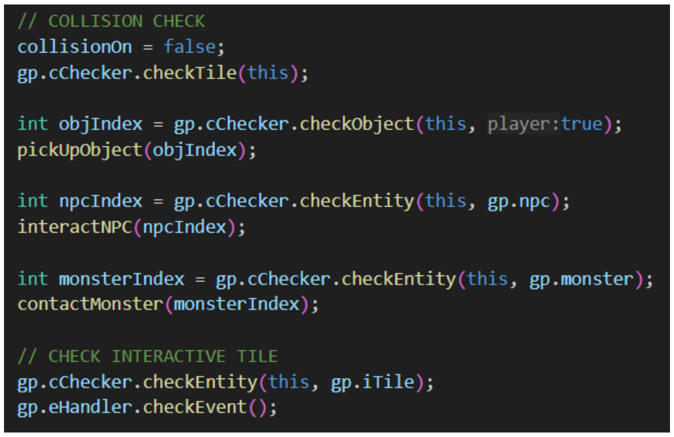


```
// COLLISION CHECK
collisionOn = false;
gp.cChecker.checkTile(this);
```

Collision checking

2. Check whether the player interacts with a specific object(character).

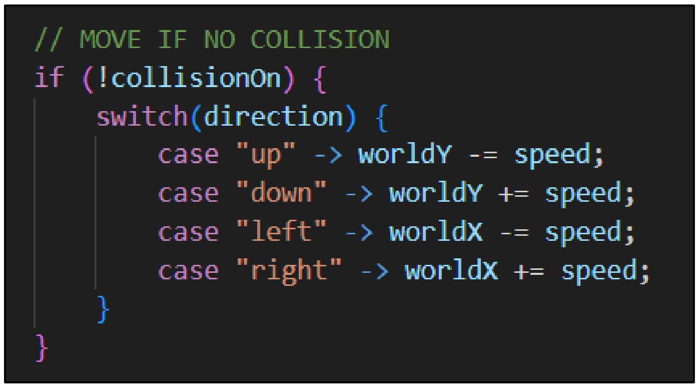
Call the corresponding method with each object(character).



```
// COLLISION CHECK
collisionOn = false;
gp.cChecker.checkTile(this);

int objIndex = gp.cChecker.checkObject(this, player:true);
pickUpObject(objIndex);

int npcIndex = gp.cChecker.checkEntity(this, gp.npc);
interactNPC(npcIndex);

int monsterIndex = gp.cChecker.checkEntity(this, gp.monster);
contactMonster(monsterIndex);

// CHECK INTERACTIVE TILE
gp.cChecker.checkEntity(this, gp.iTile);
gp.eHandler.checkEvent();
```

Interactions code

3. If the interacting object is passed through, then we will move:



```
// MOVE IF NO COLLISION
if (!collisionOn) {
    switch(direction) {
        case "up" -> worldY -= speed;
        case "down" -> worldY += speed;
        case "left" -> worldX -= speed;
        case "right" -> worldX += speed;
    }
}
```

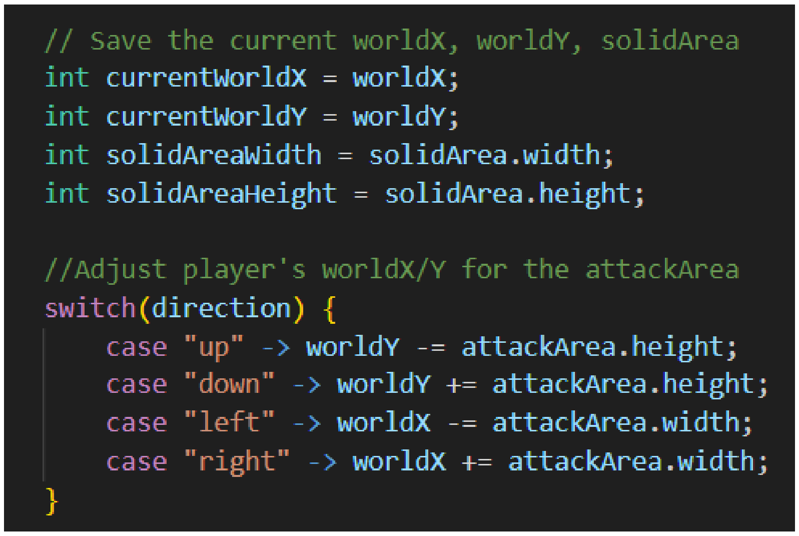
After interactions

B) Attacking mode

The key difference between unattacking mode and attacking mode is the place of the interacted object(character) before and after the interaction.

In the unattacking mode, we only change the place after interaction; meanwhile, in the attacking mode, we make the changes before and after interaction.

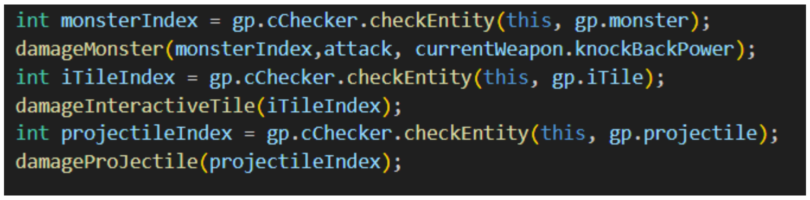
Before interaction



```
// Save the current worldX, worldY, solidArea
int currentWorldX = worldX;
int currentWorldY = worldY;
int solidAreaWidth = solidArea.width;
int solidAreaHeight = solidArea.height;

//Adjust player's worldX/Y for the attackArea
switch(direction) {
    case "up" -> worldY -= attackArea.height;
    case "down" -> worldY += attackArea.height;
    case "left" -> worldX -= attackArea.width;
    case "right" -> worldX += attackArea.width;
}
```

Before interactions(attacking mode)

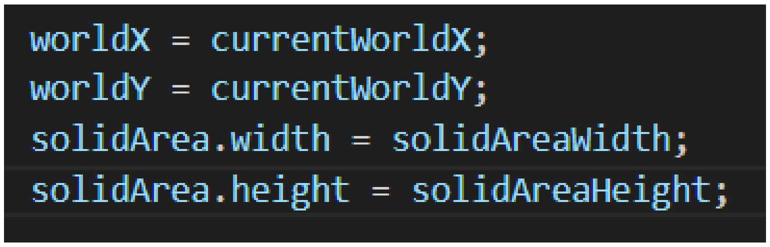
Interaction



```
int monsterIndex = gp.cChecker.checkEntity(this, gp.monster);
damageMonster(monsterIndex,attack, currentWeapon.knockBackPower);
int iTileIndex = gp.cChecker.checkEntity(this, gp.iTile);
damageInteractiveTile(iTileIndex);
int projectileIndex = gp.cChecker.checkEntity(this, gp.projectile);
damageProJectile(projectileIndex);
```

While interactions (attacking mode)

After interaction

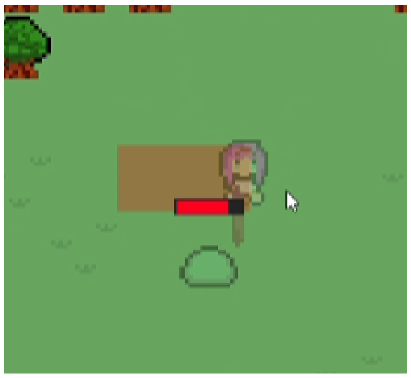


```
worldX = currentWorldX;
worldY = currentWorldY;
solidArea.width = solidAreaWidth;
solidArea.height = solidAreaHeight;
```

After interactions (attacking mode)

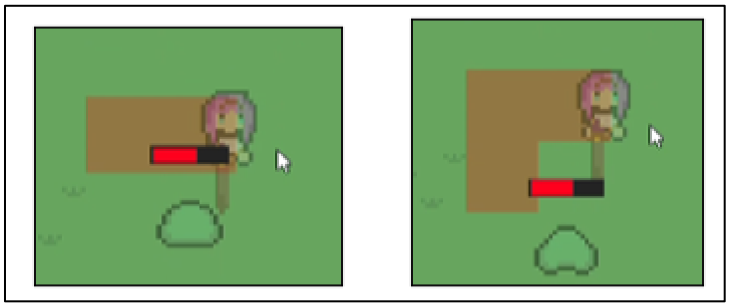
Illustration

Before starting the loop of interaction



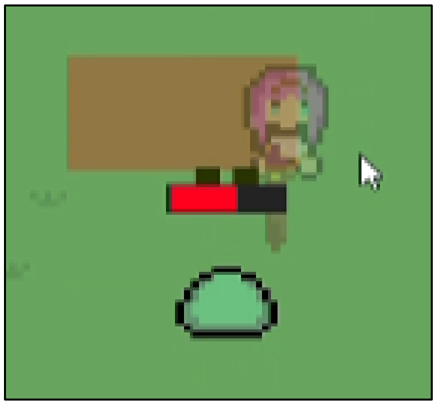
Before interactions (attacking illustration)

Then, we will combine steps 1 and 2 because they occur too fast, causing them to appear to overlap.



While interactions (attacking illustration)

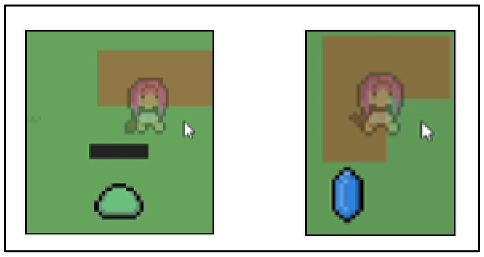
Step 3: The monster goes back to the original place.



After interactions (atacking illustration)

This iteration will continue until the attacking mode is off. There are three scenarios this mode is off:

The monster is dying



Monster is dying

Our player is dying.

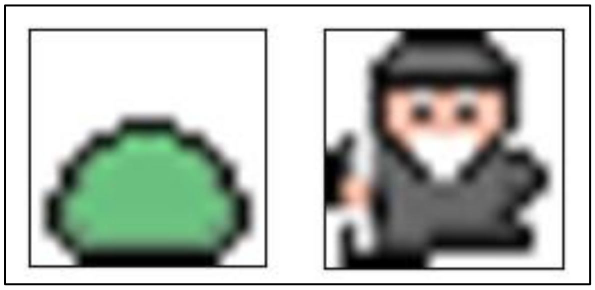
Player chooses not to fight anymore.

2.3.3.3 Features

We will stop to discuss another object (character) because almost all other objects (characters) are born for the player's interaction. However, inside each object (character), including some special features that we want to mention here.

A* algorithm – pathFinding features

Applied objects: NPC old man and monster

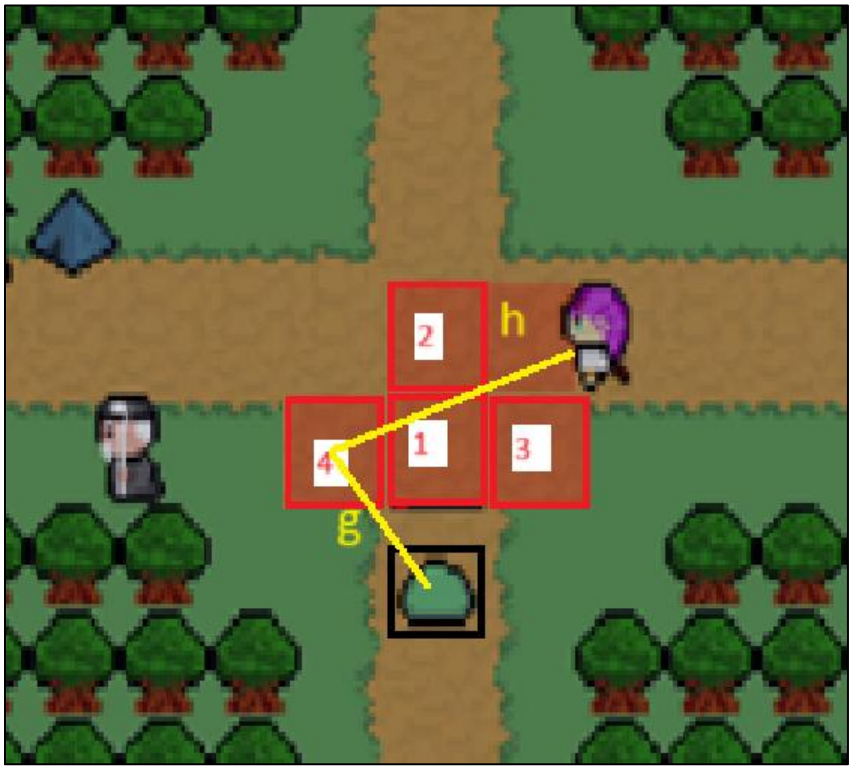


A applied objects*

The application's purpose is to make the interaction more real. For example, before the application, even though the player was very close to the monster, the monster may have avoided the player.

Applying the algorithm of heuristic gold makes the monster likely to hunt the player.

Explanation:



A illustration*

Denote cell 1 as current node.

Then, we will store four cells around the first cell, including the second, third, fourth, and black cells.

However, the black cell is the starting node so we will exclude it from the checking list.

Next, we will compute h cost and g cost:

H cost: distance from the gold node to the checking cell

G cost: distance from the starting node to the checking cell

Then set F cost = H cost + G cost

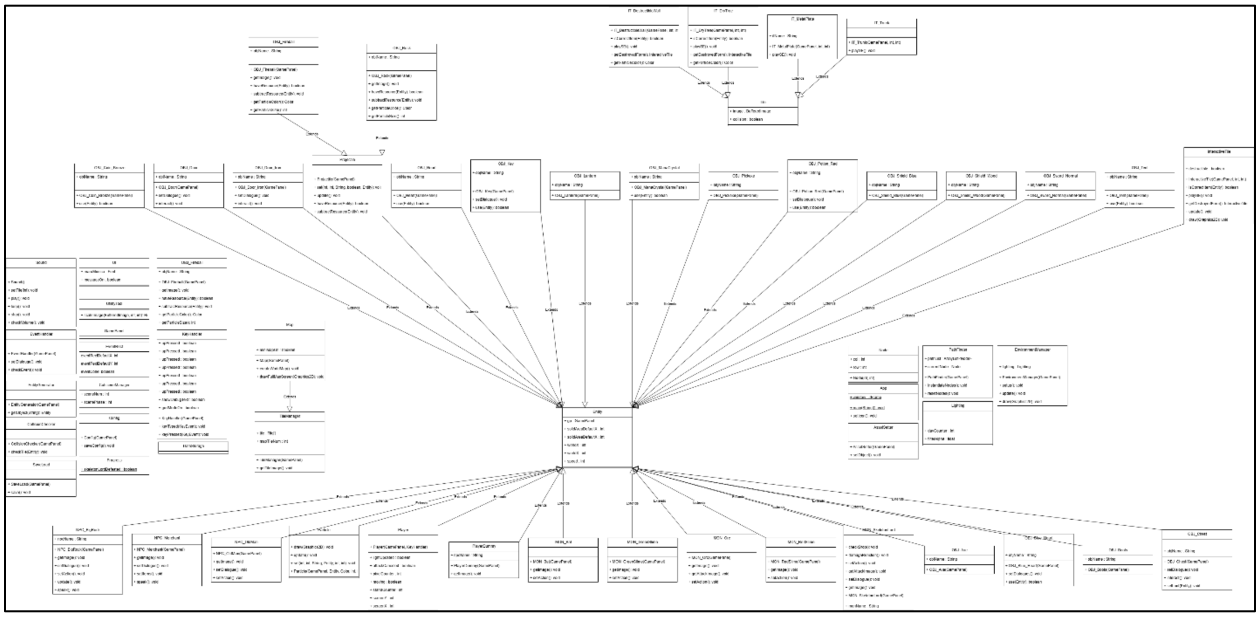
Compare F cost among three checking nodes 2nd node, 3rd node, 4th node:

If this node has the smallest F cost, we will put it in the tracking path; in this case, it is the second node.

If the F cost is equal, then we will compare the h cost.

Because the 2nd is not the gold node, we will continue the loop of finding a path until we reach the gold node.

2.4 UML



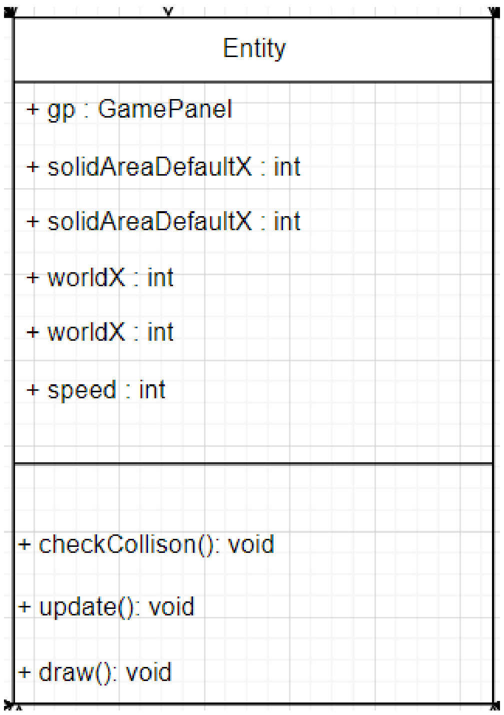
This UML class diagram lays out the architecture of a 2D game system, something in the action-adventure or survival genre. It gives us a bird's-eye view of the game's core components: players, non-player characters (NPCs), monsters, items, environment, and various utility systems. This system was designed with object-oriented principles, focusing on inheritance, modularity, and expansion.

The Entity Class

At the very core of this setup is the Entity class. Think of it as the blueprint for every movable and interactive object in the game world. It holds all the standard stuff like:

- worldX, worldY: Their coordinates in the game world.
- speed: How fast they can move.
- solidArea: This defines their collision box.
- gp: A reference to the GamePanel, the main game window.

Key methods here include update() for all the behind-the-scenes logic and draw(Graphics2D g2) to put the entity on the screen.

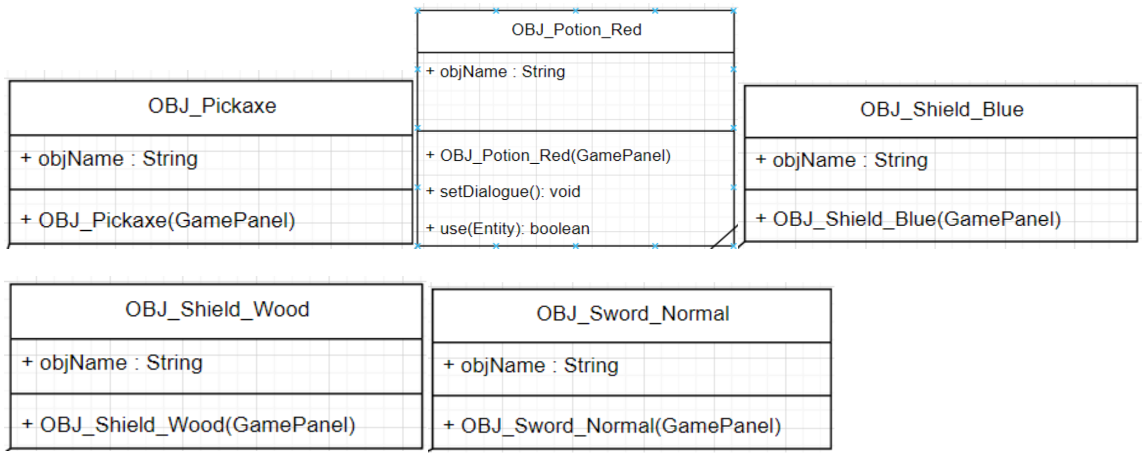


Entity's Subclasses

A whole bunch of classes inherit from Entity, and we can group them loosely into a few categories:

<u>a. Weapons and Shields</u>

You'll see things like OBJ_Sword_Normal, OBJ_Shield_Wood, and OBJ_Shield_Blue. Each represents a specific weapon or shield with its unique quirks and how it's used, often through a use(Entity e) method.



<u>b. Interactive and Collectable Items</u>

Good examples here are OBJ_Key, OBJ_Door, OBJ_Lantern, OBJ_Potion_Red, OBJ_Tent, and OBJ_Jewel_Blue. These aren't just for show; they have distinct effects like healing, unlocking things, or kicking off in-game events.

c. Players and NPCs

The Player class extends Entity but gets extra bells and whistles for managing inventory, direction, interactions, and game status.

Then there are the NPC classes, like NPC_Merchant, NPC_OldMan, and NPC_Child. Each one plays a specific role in dialogue, quests, or trading.

d. Monsters and Bosses

Think MON_Bat, MON_GreenSlime, or even the big bad MON_BossLord. These classes come packed with their own AI behaviours, like how they move (setAction()), how they react when they take damage (damageReaction()), and other enemy-specific logic.

The Environment and Game World

a. Tile and Map System

The TileManager is in charge of rendering the tile-based map. Each Tile stores information, such as its image and whether something can collide with it. The Map handles the overall map logic and those slick transitions between different areas (mapSwitch).

b. Sound Management

The Sound class handles all the background music and sound effects, setting the mood. And AssetSetter? We use that to place all the initial game entities on the map when the game starts.

c. Event Handling

EventHandler and EventRect work together to manage trigger zones for in-game events. Meanwhile, Config is there for saving and loading player settings, keeping things personalized.

d. Pathfinding System

PathFinder is where the magic happens for enemy and NPC movement; it implements the pathfinding algorithm. Each Node represents a grid position and holds all the essential pathfinding properties (like gCost, hCost, and whether it's solid).

Game Interface and Rendering

The UI class is the maestro behind rendering menus, inventory screens, status displays, and all that good stuff. And the GamePanel? That's the main canvas, combining game logic, rendering, and input in one place.

Classes like OBJ_Heart (your health!), OBJ_Boots (speed!), and various panels (MON_Death, MON_Ending, MON_GameOver) all contribute to how you interact with the game and see its status.

Chapter 3: Future Work

While the current version of "Adventures of Player" establishes a solid foundation, several avenues exist for future enhancements and expansion:

Content Expansion:

More Diverse Maps & Levels: Introduce new environments with unique themes, challenges, and hidden areas.

Expanded Roster of Enemies & NPCs: Add a wider variety of monsters with different attack patterns and behaviors, and more NPCs offering quests, lore, or services.

Richer Storyline & Quests: Develop a more intricate main storyline and introduce side quests to deepen player engagement and world-building.

Additional Items & Equipment: Implement a broader range of weapons, armor, consumables, and special items with unique effects.

Gameplay Mechanics & AI:

Advanced Player Abilities & Skill Trees: Introduce class-specific skills, spells, and a progression system allowing for character customization.

Improved AI: Enhance monster and NPC AI with more sophisticated pathfinding, decision-making, and group behaviors.

Refined Combat System: Add more depth to combat, such as special attacks, combos, status effects, or defensive maneuvers.

Boss Battles: Design challenging and memorable boss encounters with unique mechanics.

Technical & UX Improvements:

Enhanced Graphics & Animations: Improve visual fidelity with more detailed sprites, smoother animations, and particle effects.

Sound Design & Music: Integrate a richer soundscape with ambient sounds, sound effects for actions, and an original soundtrack.

UI/UX Refinements: Optimize the user interface for better clarity, intuitiveness, and aesthetic appeal.

Performance Optimization: Profile and optimize code for smoother performance, especially as more content is added.

Robust Save/Load System: Further develop the save/load functionality to ensure reliability and handle more complex game states.

Expanded Settings: Offer more customization options for controls, graphics, and audio.

Advanced Features (Long-term):

Multiplayer Functionality: Explore possibilities for cooperative or competitive multiplayer modes.

Procedural Content Generation: Investigate procedural generation for maps or dungeons to increase replayability.

Modding Support: Consider designing the game to be extensible by the community.

Chapter 4: Conclusion

The *Adventures of Player* project successfully demonstrates the practical application of Object-Oriented Programming (OOP) principles through the development of an interactive 2D adventure game. By implementing core OOP concepts such as inheritance, polymorphism, encapsulation, and abstraction, the team translated theoretical knowledge into a fully functional software product.

The game not only offers engaging gameplay and creative design but also integrates a structured architecture that supports maintainability and scalability. Features such as character interaction, real-time combat, inventory and exchange systems, and pathfinding AI using the A* algorithm reflect thoughtful design and implementation.

In addition to enhancing programming proficiency, the project fostered essential collaboration skills. Tools like Git and GitHub enabled effective version control and team coordination, aligning with real-world development practices. This experience provided valuable insights into software engineering workflows and solidified a foundation for future development in game design and beyond.

Ultimately, *Adventures of Player* stands as a testament to the team's ability to blend creativity with technical skill, producing an educational and entertaining software solution grounded in modern programming methodologies.

References

1. *Piskel tool: link*
2. *SOLID Theory: link*
3. *Design Pattern learning: link*
4. *Idea game from RyiSnow: link*
5. *OOP Lecture Slide (International University, Course: Object - Oriented Programming, Instructor: Nguyen Thanh Tung)*
6. *OOP Lab Assignment (International University, Course: Object - Oriented Programming, Instructor: Nguyen Trung Nghia)*