**2D DUNGEON GAME**

**By**

**Group 3**



**For**

**Lecturer Tran Thanh Tung**

|  |  |
| --- | --- |
|  |  |

Table of Contents

[**I.** **Introduction** 3](#_Toc169339899)

[**1.** **Abstract** 3](#_Toc169339900)

[**2.** **Game Rules** 3](#_Toc169339901)

[**3.** **Player Mechanics** 3](#_Toc169339902)

[**4.** **References** 4](#_Toc169339903)

[**5.** **Developer team** 4](#_Toc169339904)

[**II.** **SOFTWARE REQUIREMENTS** 4](#_Toc169339905)

[**1.** **What we have** 4](#_Toc169339906)

[**2.** **What we want** 4](#_Toc169339907)

[**3.** **Working tools, platform** 5](#_Toc169339908)

[**III.** **DESIGN & IMPLEMENTATION** 5](#_Toc169339909)

[**1.** **Object-Oriented Principles in Game Design** 5](#_Toc169339910)

[1.1 Inheritance 5](#_Toc169339911)

[1.2 Encapsulation 6](#_Toc169339912)

[1.3 Polymorphism 7](#_Toc169339913)

[1.4 Abstraction 7](#_Toc169339914)

[**2.** **Game Panel** 8](#_Toc169339915)

[2.1 UI Design 8](#_Toc169339916)

[2.2 Main Methods 13](#_Toc169339917)

[2.3 UML 23](#_Toc169339918)

[**IV.** **EXPERIENCE** 27](#_Toc169339919)

# **Introduction**

## **Abstract**

In the rapidly expanding field of software engineering and development, video game creation stands out as both unique and similar to other software projects. Its uniqueness lies in the collaboration of teams from various disciplines, such as art, and programming. Additionally, achieving engaging gameplay often involves extensive prototyping and iterative development.

Our current project involves developing a 2D platformer game for the “Object-Oriented Programming” course, which is a four-credit component of our degree. This project provides us with an excellent opportunity to enhance our development cycle, manage development periods effectively, and improve our graphical skills. Moreover, it allows us to refine our Java programming abilities and apply theoretical concepts learned in class, such as object-oriented programming, design patterns like the singleton.

## **Game Rules**

The primary goal is to reach the end of each level, defeating all enemies and avoiding death.

## **Player Mechanics**

Movement: The player can move left, right, and jump. The controls are typically mapped to the W-A-S-D keys.

Attacking: The player can attack enemies using a specified key, which reduces the enemy's health.

Death: If the player's health reaches zero, they lose a life or the game ends.

Enemies: Various types of enemies are present, each with distinct behaviors and attack patterns.

Health System: Both the player and enemies have health points that decrease when hit.

## **References**

Images from <https://pixelfrog-assets.itch.io/kings-and-pigs>

Tutorial 2d platformer game: <https://www.youtube.com/@KaarinGaming>

## **Developer team**

|  |  |  |
| --- | --- | --- |
| Full Name | Student ID | Declaration of contribution |
| Đỗ Gia Thụy | ITCSIU21237 | Handle enemy configuration |
| Nguyễn Thanh Tú | ITITIU21338 | Create overlay screen, make presentation |
| Đỗ Huỳnh Duy Tiến | ITCSIU22291 | Write report and handle the control of player |
| Lý Khải Minh | ITITSB22008 | Config canon and projectiles and traps, make presentation |
| Nguyễn Khánh Vinh | ITDSIU21129 | Create levels for the game |

# **SOFTWARE REQUIREMENTS**

## **What we have**

1. User friendly efficient and lucrative system.

2. Minimum maintenance cost (graphics).

4. Easy to operate.

## **What we want**

1. Develop system within limited cost.

2. Maximum high definition.

3. Design the whole system with efficient manner.

5. Easy to update.

## **Working tools, platform**

1. Eclipse with proper libraries.

2. GIMP 2.10.38 for editing levels.

# **DESIGN & IMPLEMENTATION**

## **Object-Oriented Principles in Game Design**

Object-oriented programming is fundamental in game development, facilitating organized and maintainable code. The following OOP principles are employed:

### Inheritance

Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class, promoting code reuse and establishing hierarchical relationships between classes. In the game code, inheritance is widely used, with several classes extending from base classes.

|  |
| --- |
|  |
| Pic 1: Example of inheritance |

In the example above, the Entity class serves as a base class, providing common properties and methods for all entities in the game, such as position (x, y), dimensions (width, height), and a hitbox. The Player class and Piggy class (which extends Enemy) inherit from the Entity class, inheriting its properties and methods while adding their own specific attributes and behaviors.

### Encapsulation

Encapsulation is the bundling of data (attributes) and methods (behavior) together within a class. It helps in achieving data abstraction and hiding the internal implementation details of an object from the outside world. In the game code, encapsulation is evident in various classes, such as Player, Enemy, GameObject, and their subclasses.

|  |
| --- |
|  |
| Pic 2: Example of encapsulation |

In the Player class, the attributes such as animations, moving, attacking, left, right, jump, and lvlData are private, encapsulating the internal state of the player object. The class provides public methods like setAttacking() and isLeft() to access and modify these attributes in a controlled manner.

### Polymorphism

|  |
| --- |
|  |
| Pic 3: Example of polymorphism |

In the example above, the Enemy class is an abstract class that defines a common update() method for all enemy types. The Piggy class, which extends Enemy, overrides the update() method with its specific implementation for updating the behavior, animation, and attack box of the "Piggy" enemy. This demonstrates polymorphism through method overriding, where objects of different classes (Piggy, and potentially other enemy types) can be treated as instances of the Enemy class.

### Abstraction

Abstraction involves simplifying complex systems by hiding unnecessary details and exposing only the essential features to the user or other components of the system. In the game code, abstraction is achieved through the use of abstract classes and interfaces.

|  |
| --- |
|  |
| Pic 4: Example of abstraction |

In the Enemy class, various methods are declared as abstract, such as firstUpdateCheck(), updateInAir(), move(), turnTowardsPlayer(), canSeePlayer(), isPlayerInRange(), and isPlayerCloseForAttack(). These methods define the common behavior that all enemy types should implement, but the specific implementation details are left to the concrete subclasses (e.g., Piggy).

By using abstract classes, the game code achieves abstraction by separating the general enemy behavior from the specific implementation details of each enemy type. This approach promotes code reuse, modifiability, and extensibility, as new enemy types can be added by creating new concrete subclasses that inherit from the Enemy class and provide their own implementations of the abstract methods.

## **Game Panel**

### UI Design

|  |
| --- |
|  |
| Pic 5: Flowchart about the proccess of the beginning game |

* + 1. **Pause Screen**

The pause screen implementation is handled primarily by the PauseOverlay class. This class is responsible for rendering the pause menu and managing its interactive components.

|  |
| --- |
|  |
| Pic 6: Pause Overlay |

Key features of the pause screen include:

* Background Image:

A background image is loaded to serve as the visual backdrop for the pause screen. This image is scaled and positioned appropriately.

|  |
| --- |
|  |
| Pic 7: The logic to load background image |

* Sound Buttons:

Two sound buttons are created for controlling music and sound effects. These buttons toggle the muted state when interacted with.

|  |
| --- |
|  |
| Pic 8: The logic to load sound button |

* Utility Buttons:

Three utility buttons (menu, replay, and unpause) are provided for navigating to the main menu, restarting the level, or resuming the game.

|  |
| --- |
|  |
| Pic 9: The logic to load utility buttons |

* Volume Control:

A volume button allows players to adjust the volume through a slider interface.

|  |
| --- |
|  |
| Pic 10: The logic to create volume button |

* Interactivity:

The PauseOverlay class includes methods for handling mouse events to update the state of buttons and volume control based on user input.

|  |
| --- |
|  |
| Pic 11: methods to control the state of buttons and volume |

* + 1. **Death Screen**

The death screen is referenced indirectly in the code snippets through the use of buttons that manage game states.

|  |
| --- |
|  |
| Pic 12: Screen when being dead |

The implementation for rendering and updating the death screen follows similar principles to the pause screen, but with a focus on options relevant after player death, such as retrying or returning to the main menu.

* Rendering:

The death screen uses a dark overlay to signify the end of the game, and buttons are provided to navigate from this screen.

|  |
| --- |
|  |
| Pic 13: The logic to render death screen |

### Main Methods

* + 1. Handle character

|  |
| --- |
| A diagram of a computer program  Description automatically generated |
| Pic 14: Flow chart showing the logic of character |

* Player

The Player class extends the Entity class and manages player-specific attributes and behaviors such as animations, movement, jumping, gravity, health, and UI rendering.

* Movement

The player movement is managed through the ‘updatePosition’ method, which is called within the update ‘method’.

Left/Right Movement: If the left flag is true, the player moves left by decreasing the x-coordinate by the walkSpeed. If the right flag is true, the player moves right by increasing the x-coordinate by the walkSpeed.

Jumping: If the jump flag is true and the player is not already in the air (!inAir), the player jumps by setting inAir to true and setting the y-velocity to jumpSpeed.

* Attacking

The player's attack functionality is managed through the update method, which checks for attacks and updates the attack box position.

|  |
| --- |
|  |
| Pic15: Check attack logic |

Update Attack Box: The updateAttackBox method updates the position of the attack box to match the player's current position.

Check Attack: If the attacking flag is true, the checkAttack method checks for collisions between the attack box and enemies or objects by calling checkEnemyHitand checkObjectHit.

* Being Hit

When the player gets hit, their health is reduced. This is typically managed within the update method based on various game interactions.

|  |
| --- |
|  |
|  |
| Pic 16: Health checking when being hit |

Check Health: if the player's health (currentHealth) is less than or equal to 0. If it is, it sets the player to a dying state by calling playing.setPlayerDying(true). If the player is dead, it sets the game over state by calling playing.setGameOver(true).

Change Health: The changeHealth method adjusts the player's health by the specified value. It ensures that the player's health does not exceed the maximum health (maxHealth).

* Piggy
* Action

The 'Piggy' class represents a specialized enemy type within the game, inheriting properties and methods from the base 'Enemy' class. This class incorporates unique behaviors and attributes specific to the Piggy enemy type, such as distinctive attack mechanics and movement patterns. Pig entities are seamlessly integrated into the game map through the use of constant values and adaptive state parameters within the 'Constant' class. This design facilitates scalability and adaptability, accommodating changes in enemy size and varying environmental conditions.

|  |
| --- |
|  |
| Pic 17: The logic to generate piggy |

* Behavior

A diagram of a game

Description automatically generated

Pic 18: Behavior of piggy

The flowchart illustrates a state machine for controlling enemy behavior in a platformer game. It outlines how an enemy transition between different states based on various conditions, primarily related to the player's presence and proximity:

**States**

* **Game Starts:** The initial state when the game begins.
* **In Air/Fall:** The enemy is not grounded and is either falling or jumping.
* **Idle:** The enemy is not moving or attacking.
* **Running:** The enemy is moving horizontally.
* **Attacking:** The enemy is executing an attack.
* **Move Towards Player:** The enemy is actively pursuing the player.

**Transitions and Conditions**

* **Game Starts -> In Air/Fall:** The game begins, and the enemy's initial state is determined by whether it's in the air or not.
* **In Air/Fall -> On Floor (Decision):** If the enemy is on the floor, it transitions to the Idle state. If not, it remains in the In Air/Fall state.

|  |
| --- |
|  |
|  |
| Pic 19: Code logic handling piggy when it on air |

* **Idle -> Running:** If the player is in sight, the enemy transitions to the Running state.
* **Running -> Player in Sight (Decision):** If the player is no longer in sight, the enemy returns to the Idle state. Otherwise, it proceeds to check if the player is in range.

|  |
| --- |
|  |
| Pic 20: Checking whether player in range of piggy can see |

* **Running -> Player in Range (Decision):** If the player is in range, the enemy transitions to the Attacking state. If not, it continues to move towards the player.
* **Move Towards Player:** The enemy moves towards the player's position.

|  |  |
| --- | --- |
|  |  |
| Pic 21: Piggy runs toward player if in range of attacking | |

* **Attacking:** The enemy executes its attack animation and logic.
* **Attacking -> On Floor (Decision):** After the attack, the enemy checks if it's on the floor. If yes, it goes back to Idle; otherwise, it goes to InAir/Fall.

|  |
| --- |
|  |
| Pic 22: Attack player |

* + 1. Handle Image

BufferedImage serves as a crucial tool for storing and manipulating image data for each character in the game. It allows for efficient management of its visual representation, ensuring smooth integration into the game environment. This applies for player and enemy class.

|  |  |
| --- | --- |
| A black screen with pixelated objects  Description automatically generated | A screenshot of a computer code  Description automatically generated |
| Pic 23: Image of Player and the way to store image | |

Take Player for example:

* + 1. Handle level

The system leverages the RGB color model, where each pixel represents a single tile within the game world. Each color component (red, green, blue) holds distinct information:

* Red: set the tile
* Green: set the enemy
* Blue: set the object

The level data is store as below:

|  |
| --- |
|  |
| A screenshot of a computer code  Description automatically generated |
| Pic 24: Image of the map and the way to store image |

The image below illustrates a sample design for Level 1 using the RGB system:

|  |
| --- |
| A screenshot of a computer  Description automatically generatedA screenshot of a computer  Description automatically generatedA screenshot of a video game  Description automatically generated |
| Pic 25: The way to initialize player and piggy, as well as other obstacles |

The green value for the player has been set to 100, and the green value for the enemy (Piggy) has been set to 0, indicating their respective starting positions.

### UML

A diagram of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated

Pic 26: UML about Player and Enemy class, which extends Entities class

The Enemy class represents game antagonists with attributes like health, damage, speed, and position. It manages behaviors such as patrolling, attacking, and moving, responding to player interactions. The class serves as a base for specific enemy types, allowing for polymorphism and code reuse.

The Player class represents the user-controlled character with attributes like health, damage, speed, and position. It processes keyboard and mouse inputs to control movements and actions like moving, jumping, and attacking. Inheriting from the Entity class, it adds specific functionalities for player interactions.

A computer screen shot of a computer program

Description automatically generated

Pic 27: UML about Main, inputs, and ui package

Menu Class

The Menu class is responsible for executing UI-related functionalities within the game. It acts as the interface between the user and the game system, providing various options and settings for the player to interact with.

Game Class

The Game class is the core of the game system, performing operations for multiple other classes. It serves as the controller for the game's logic, managing instances of the Player, Enemy, and Level classes, as well as handling overall game flow and state.

GamePanel Class

The GamePanel class is a graphical component created within the GameWindow class. It is integrated into a Frame object, where the combination of Frame and Panel constitutes the complete game window.

A diagram of a company

Description automatically generated with medium confidence

Pic 28: UML about game state class

The GameState class is a pivotal component in managing the different states of a game. This class typically handles transitions and the logic associated with each distinct phase or mode within the game.

A screenshot of a computer program

Description automatically generated

Pic 29: UML about Utilz package

The utilz package is a utility package designed to provide common functionality and helper methods that can be used across various parts of the game. This package contains classes and methods that offer generic and reusable functionalities, making the overall codebase more modular and easier to maintain.

# **EXPERIENCE**

In a game project, the product is a game. And after almost two months working on it, we all come to the point:

A game is much more than just its software. It has to provide content to become enjoyable. Just like a web server: Without content the server is useless, and the quality cannot be measured.

During the time working on the project, our team realize that the software part of the game is not the only one, and it must be considered in connection to all other parts: the environment of the game, the story, game plays, the artwork, animation, and so on.

We have chance to practice a lot of lesson which we have learnt on class, and deals with plenty of bugs. Besides, it encourages students study more outside the class. Therefore, we not only firm the knowledge from lesson, but also update information technology and have new experience.

Finally, our team realizes that Computer Science major need its student self-study most of the time. IT world is too big to wait and just learn from school.