

UNIVERSITY OF HUDDERSFIELD  
DEPARTMENT OF COMPUTER SCIENCE



# PROJECT REPORT

**MODULE: Object-Oriented Systems Development**

**PROJECT TITLE: Monk - RPGGame**

**Student ID: U2259343**

**Name: Lai Minh Thong**

**UNDER THE GUIDANCE OF**

Dr Rabia Jilani

Mr. Diaeddin Alarnaouti

---

# TABLE OF CONTENTS

---

I) Introduction .....	3
1) Short Introduction .....	3
2) Final Product .....	5
II) Code Implementation .....	8
1) UML Diagram .....	8
2) Algorithm and Structures of classes .....	8
3) Extre Features .....	26
III: Test Cases .....	28
IV: Evaluation .....	36
VI: Conclusions .....	37

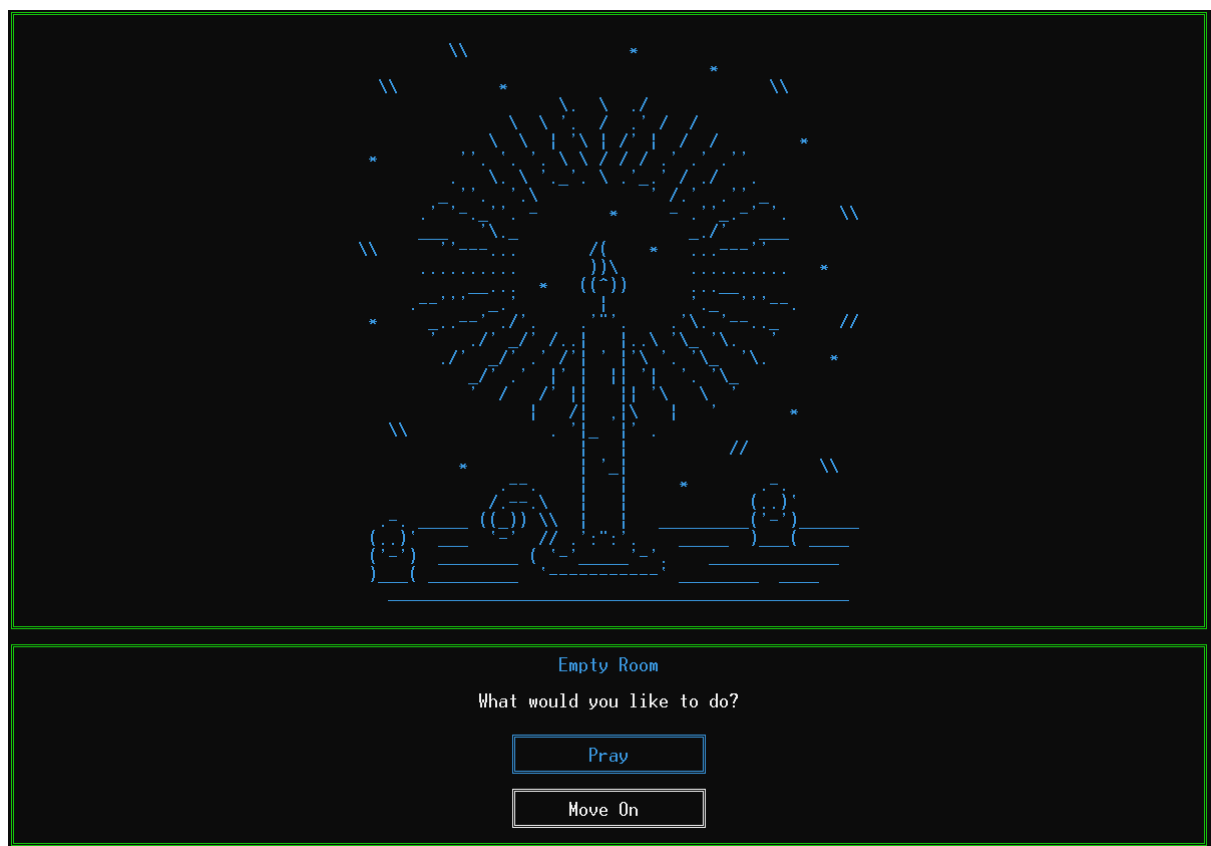
# I) Introduction

## 1) Short Introduction

**Monk – RPG game**, written in C/C++, runs directly on the console window. Use the knowledge learned in the OOP course and some extended knowledge about C/C++ to apply to product implementation

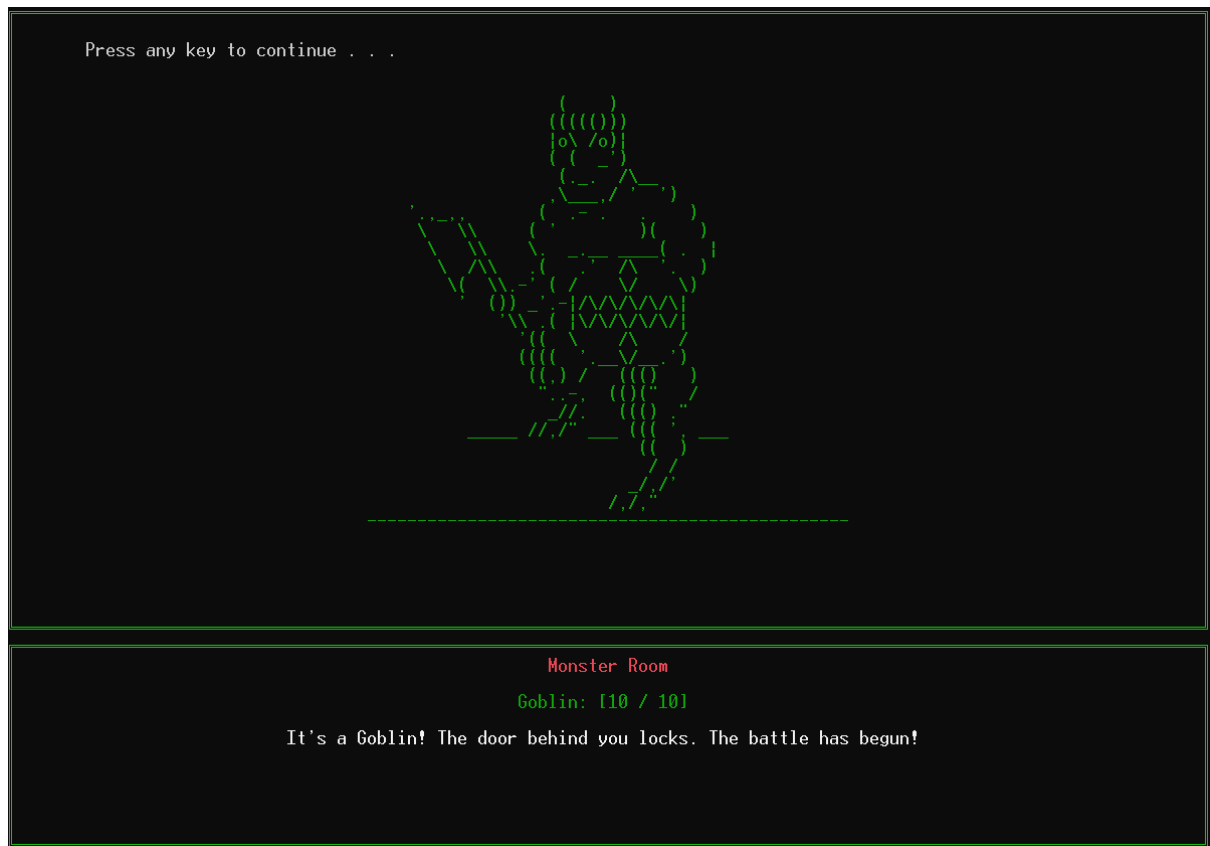
- Game scenario: As the monk, you will explore various rooms, each with its own unique challenges and rewards. There are different types of rooms you may encounter, including monster rooms and treasure rooms. Make strategic decisions on which rooms to enter, as the difficulty increases the deeper you venture:

+ **Empty Room:** the Monk can pray to restore his health points fully and equipment to enhance your monk's abilities.



Pic 1.1: Empty Room

+ **Monster Room:** the Monk will engage in turn-based combat with fearsome creatures. Use your skills and abilities wisely to defeat the monsters and progress further.



Pic 1.2: Monster Room

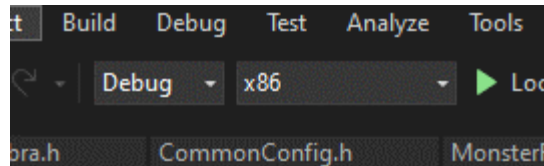
+ **Treasure Room:** the Monk will discover valuable items.



*Pic 1.3: Treasure Room*

## 2) Final Product

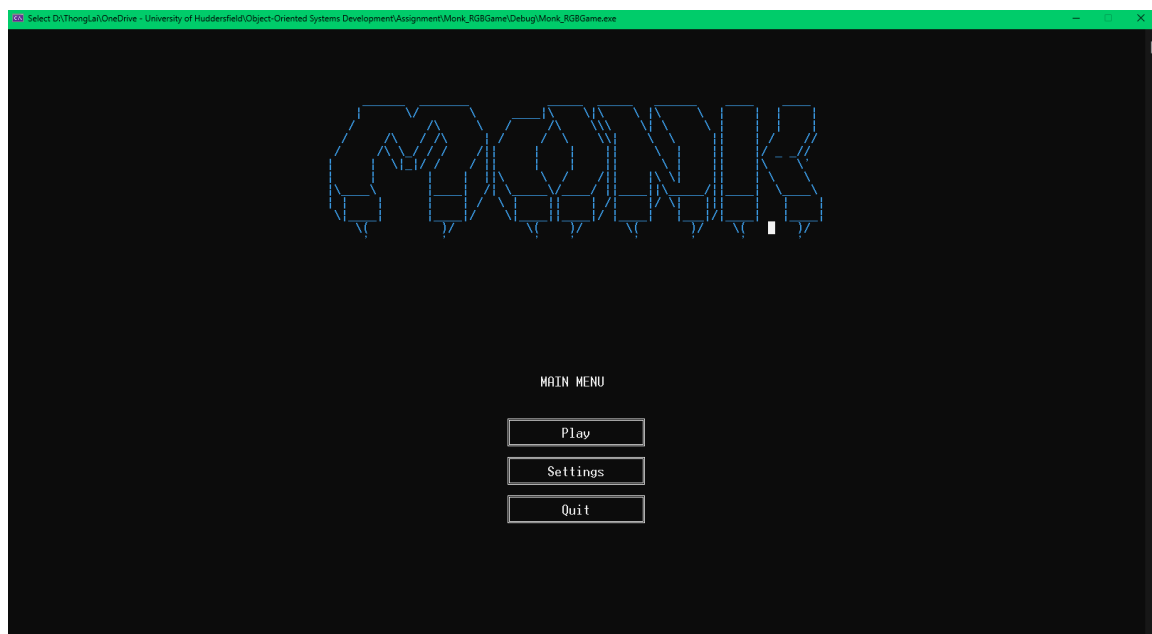
- Video demonstration link: <https://youtu.be/W9HB1wrwJEY>
- Before running the game, it is best to run on **Visual Studio 2022** with the **x86** platform as follows:



- Description of the game's interface

*At the Menu interface, use W-S or Up/Down and Enter keys to select these options:*

- [1] Play:** Start a new game.
- [2] Settings:** Player can set window size.
- [3] Quit:** Exit the game.

*Pic 2.1: Menu Screen*

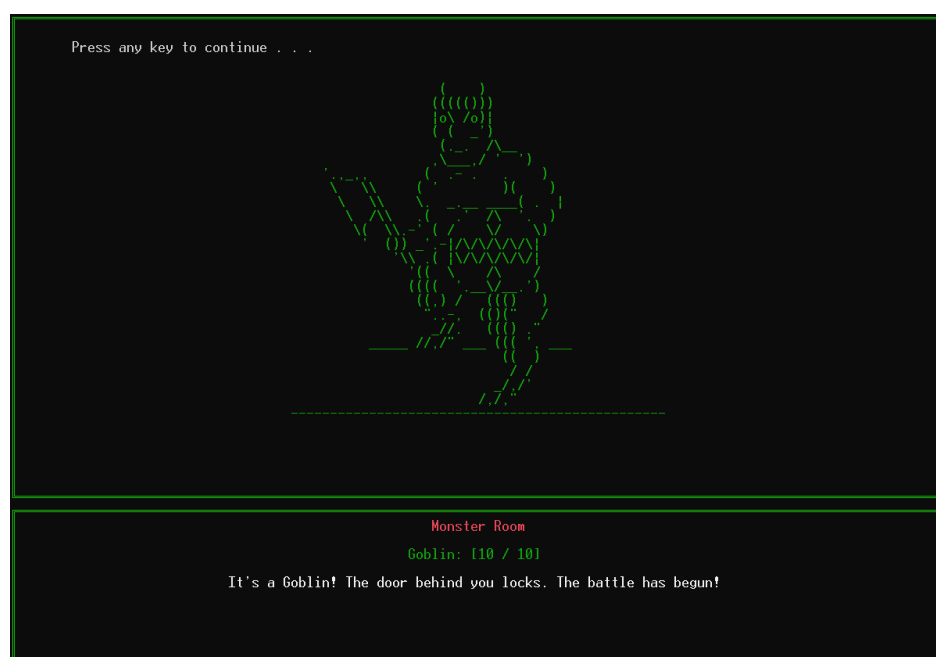
- At the in-game interface:

- On the right side is the **Status Screen**, where player can see information about **Health**, **Attack points**, and the number of **explored rooms**.



*Pic 2.2: Status Screen*

- The left side is the game's main interface. It contains **Gameplay** and a **Description Screen**.



*Pic 2.3: Main Interface*

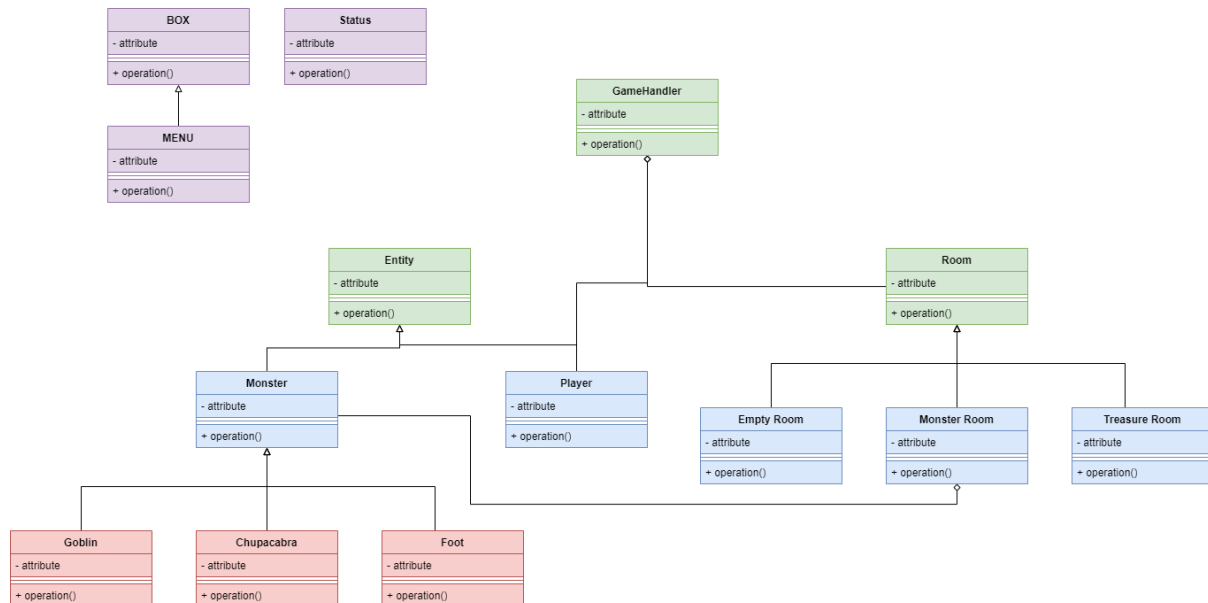
The game starts with the request for the player to enter their name and a short greeting description:

*Pic 2.4: Welcome Screen*

The player will start in the Empty Room, and then continue to explore other rooms with unique features. The game is over when the player finds the treasure or runs out of health points

## II) Code Implementation

### 1) UML Diagram



Picture II - 1.1: UML

- In this program, the **GameHandler** class is the main class that runs the entire game. It acts as the parent class for the **Player** and **Room** classes because it includes the necessary components for gameplay.
- The **Monster** and **Player** classes inherit from the **Entity** class because they share some similar features. The **Goblin**, **Chupacabra**, and **Foot** classes, on the other hand, inherit from the **Monster** class because they represent different types of monsters found in the game's monster rooms.
- To help with game navigation and provide a user interface, the program uses the **BOX** and **MENU** classes along with the **Status** class. The **BOX** and **MENU** classes help with moving through the game and making choices, while the **Status** class is responsible for displaying the user interface.
- By using this inheritance structure and incorporating the **BOX**, **MENU**, and **Status** classes, the program is designed to be organized and user-friendly, allowing for a smoother game experience.

### 2) Algorithm and Structures of classes

#### - Window functions:

Firstly, this program used many Window-handle functions to build a console UI.



```
// Set up when starting up functions  
void setRasterFonts();  
void SetWindowSize(int width, int height);  
void FixConsoleWindow();  
void HideCursor();  
void SetUpParameters();  
  
void FullScreenMode();  
void WindowedMode();  
void GetWindowSize();  
WORD DefineColor(int t_color, int t_background);  
void SetTextColor(WORD wColor);  
void GotoXY(int x, int y);  
bool GetXY(int& x, int& y);
```

Secondly, all of the contents in the gameplay will be displayed using convenient defined functions to draw certain prompts or messages throughout the game's interface.

```
int midWidth(int width, string message);

int midWidth(int width, int content_width);

int midHeight(int height, int content_height);


void printString(string message, int X, int Y, int text_color = WHITE, int
bg_color = BLACK);

void printStringCenter(string message, int text_color = WHITE, int bg_color =
BLACK);

void printStringCenterGameplay(string message, int text_color = WHITE, int
bg_color = BLACK);

void removeString(string message, int X, int Y);

void removeStringCenter(string message);


void printOnDescriptionAndWait(string prompt, int X, int Y = GAMEPLAY_H +
midHeight(DESCRIPTION_H, 1), int text_color = WHITE, int bg_color = BLACK);

void printOnDescriptionCenterAndWait(string prompt, int text_color = WHITE,
int bg_color = BLACK);


void printOnGameplayAndWait(string prompt, int X, int Y =
midHeight(GAMEPLAY_H, 1), int text_color = WHITE, int bg_color = BLACK);

void printOnGameplayCenterAndWait(string prompt, int text_color = WHITE, int
bg_color = BLACK);


void waitForKeyBoard(int X = SCREEN_WIDTH / 20, int Y = SCREEN_HEIGHT / 20);

string waitForInput(string prompt, int X, int Y, int text_color = WHITE, int
```

### **- Status Class:**

The Status class is used every time, the program needs to print out a string with a specific location and color. It will save the current setting of the cursor then after the drawing, the destructor of the Status object will revise the cursor's settings back.

```
//Save status of coordinates and text/background color before drawing
```

```
class Status
{
private:
    int X, Y;
    int COLOR;
public:
    Status();
    ~Status();
    int getX();
    int getY();
    int getColor();

    void ResetToCurrent();
}
```

For the **Game** objects classes: **Entity(Monster, Player)**, **Monster(Goblin, Chupacabra, Foot)**, **Player**:

### **- Entity Class:**

The properties and methods of the base class **Monster** and **Player** are almost the same (both have a *name*, *description*, *health*, *attack point*, ...) so they can inherit the **Entity** class:

```
class Entity {  
private:  
    string name;  
    string description;  
    int color;  
  
    int health;  
    int baseHealth;  
    int damage;  
    int cancel_chance;  
  
public:  
    Entity();  
  
    // Getters  
    string getName();  
    string getDescription();  
    int getColor();  
    int getHealth();  
    int getDamage();  
    int getBaseHealth();  
  
    // Setters  
    void setName(string monsterName);  
    void setDescription(string description);  
    void setColor(int monsterColor);  
    void setHealth(int health);  
    void setDamage(int damage);  
    void setBaseHealth(int baseHealth);  
    void setCancelChance(int cancel_chance);  
  
    bool tryCancelAction();  
    bool isAlive();  
  
    virtual void takeDamage(int amount);  
    virtual void displayHealth();  
    virtual void removeHealth();  
};
```

## Attributes:

- + **color**: Entity Color when drawing name and description (range from 0-15)
- + **cancel\_chance**: Chance that allows the entity to cancel any actions that come from the opponent.

## Methods:

- **tryCancelAction()**: Randomize a chance to ignore a turn of the opponents.

```
bool Entity::tryCancelAction() {
    return (generateRand(0, 100) < cancel_chance) ? true : false;
}
```

- **displayHealth()** and **removeHealth()**: display **Health Point** to the **Status** screen and remove it. These two functions are used to update the health of the player.

```
void Entity::displayHealth()
{
    string monster_health = name + ": [" + to_string(health) + " / " +
to_string(baseHealth) + "]";
    printString(monster_health, midWidth(GAMEPLAY_W, monster_health.size()),
GAMEPLAY_H + DESCRIPTION_H * 1 / 4, color);
}
```

```
void Entity::removeHealth()
{
    string monster_health = name + ": [" + to_string(health) + " / " +
to_string(baseHealth) + "]";
    printString(string(monster_health.size(), ' '), midWidth(GAMEPLAY_W,
monster_health.size()), GAMEPLAY_H + DESCRIPTION_H * 1 / 4);
}
```

**Class Monsters:**

```
class Monster : public Entity {
private:
    string* artModel;
    int artHeight;
    int artWidth;

    int attack_chance;
public:
    Monster();

    string* getArtModel();
    int getArtHeight();
    int getArtWidth();

    void setArtModel(string* artModel);
    void setArtHeight(int artHeight);
    void setArtWidth(int artWidth);
    void setAttackChance(int attack_chance);

    int actionToPerform();
};
```

Attributes:

- **artModel**: Contains the ASCII ART of the monster to display on the gameplay screen
- **artHeight** and **artwidth**: The maximum height and width of the monster's model art. This helps to develop a function to display and remove the art more easily.
- **attack\_chance**: The **Monsters** also can generate a random action (between **Attack** or **Defend**) which depends on this chance.

Methods:

**actionToPerform()**: Used to deduce which move the monster would make by generating random numbers and compare with the `attack_chance` attribute above.

Because the properties and methods of **Goblin**, **Chupacabra** and **Foot** are the same, we just consider **Goblin** for short:

```
Goblin::Goblin() : Monster()
{
    setName("Goblin");
    setDescription("Groan!");
    setColor(GOBLIN_COLOR);

    setBaseHealth(GOBLIN_BASE_HEALTH);
    setHealth(GOBLIN_BASE_HEALTH);
    setDamage(GOBLIN_BASE_DAMAGE);

    setCancelChance(GOBLIN_CANCEL_ACTION_CHANCE);
    setAttackChance(GOBLIN_ATK_DEF_CHANCE);

    setArtModel(GOBLIN_ART);
    setArtHeight(GOBLIN_HEIGHT);
    setArtWidth(GOBLIN_WIDTH);
}
```

The monster classes only have the constructor initialize the characteristics of each monster with the global configuration – which is defined in **CommonConfig.h**:

```
//Entities Parameters

// Player

extern int PLAYER_COLOR;

extern int PLAYER_BASE_HEALTH;

extern int PLAYER_BASE_DAMAGE;

extern int PLAYER_CANCEL_ACTION_CHANCE;


// Monsters

extern int GOBLIN_COLOR;

extern int GOBLIN_BASE_HEALTH;

extern int GOBLIN_BASE_DAMAGE;

extern int GOBLIN_CANCEL_ACTION_CHANCE;

extern int GOBLIN_ATK_DEF_CHANCE;


extern int CHUPACABRA_COLOR;

extern int CHUPACABRA_BASE_HEALTH;

extern int CHUPACABRA_BASE_DAMAGE;

extern int CHUPACABRA_CANCEL_ACTION_CHANCE;

extern int CHUPACABRA_ATK_DEF_CHANCE;


extern int FOOT_COLOR;

extern int FOOT_BASE_HEALTH;

extern int FOOT_BASE_DAMAGE;

extern int FOOT_CANCEL_ACTION_CHANCE;
```

To make it easier to update and maintain the game, the characteristics of each monster in the monster classes are initialized using global configurations defined in the **CommonConfig.h** file. By modifying these global configurations, it becomes simpler to make changes to the game or assist with gameplay testing.

### **Player Class:**



```
class Player : public Entity{
private:
    bool isProtected;
public:
    Player();

    bool hasProtection();
    void setHasProtection(bool isProtected);

    void setPlayer();
    void takeDamage(int amount);

    void displayHealth();
    void removeHealth();
    void displayDamage(int color = PLAYER_COLOR);
    void displayProtection();
```

Attributes:

- **isProtected()**: There is an item (Shield of Angel) that helps to prevent an attack from the monsters once, this attribute indicates a flag to refuse taking damage.

**Display methods**: Use to update the player's attributes on the screen

**Room Class**:

```
class Room {
private:
    Room* leftRoom;
    Room* rightRoom;

    string roomName;
    string description;

    int nameColor;
    int descColor;
public:
    Room(string roomName, int nameColor = WHITE, int descColor = WHITE);
    ~Room();

    void setDescription(string description);
    void setLeftRoom(Room *left);
    void setRightRoom(Room* right);

    string getName();
    string getDescription();
    Room *getLeftRoom();
    Room *getRightRoom();

    void displayRoomNameAndDesc();
    void removeRoomName();
    virtual bool processRoom(Player* player) = 0;
```

Attributes:

**leftRoom, rightRoom:** Pointer of class Room, contains the next left or right room when generating new ones.

**roomName, description:** Will be displayed in the **Description** area, every time the player moves to the next room.

Methods:

**displayRoomNameAndDesc()** and **removeRoomName()**: Print out and update the name and description attribute at the center of the **Description** area with the following nameColor and descriptionColor.

**processRoom()**: A pure virtual method that is used to process each room type differently.

### Empty Room:

```
class EmptyRoom : public Room {  
private:  
    bool hasItem;  
public:  
    EmptyRoom();  
  
    bool Item();  
  
    bool processRoom(Player* player);  
};
```

Attributes:

**hasItem**: A boolean flag attribute indicates whether there is an item in the room, so the player can decide to pick it up or not.

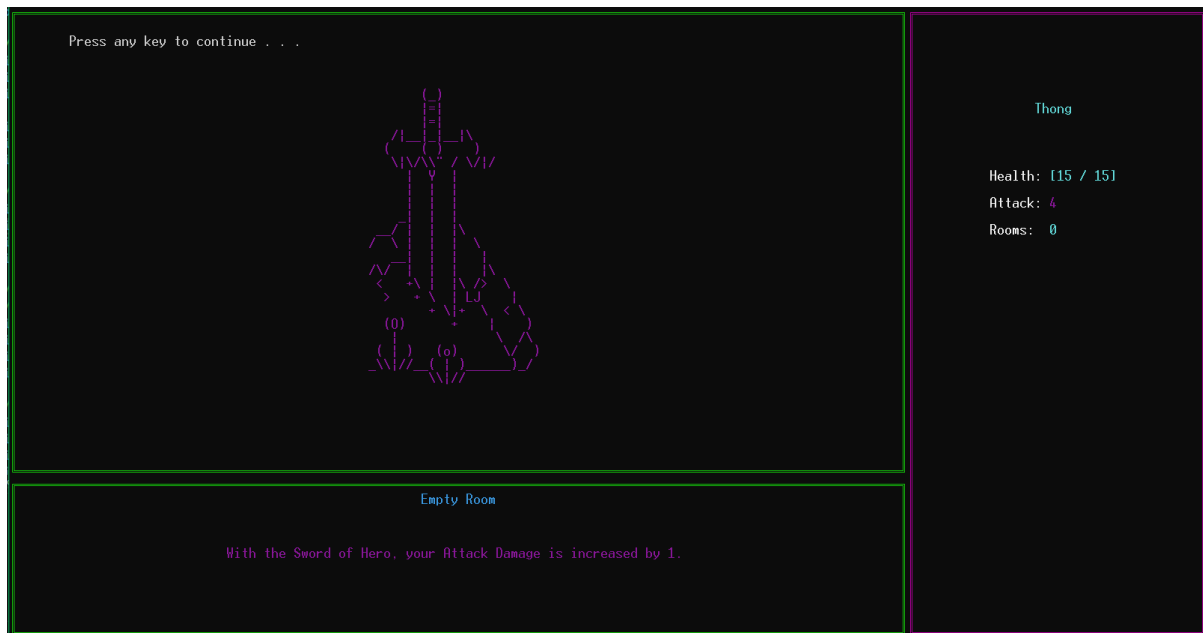
Methods:

**Item()**: return the **hasItem** attribute.

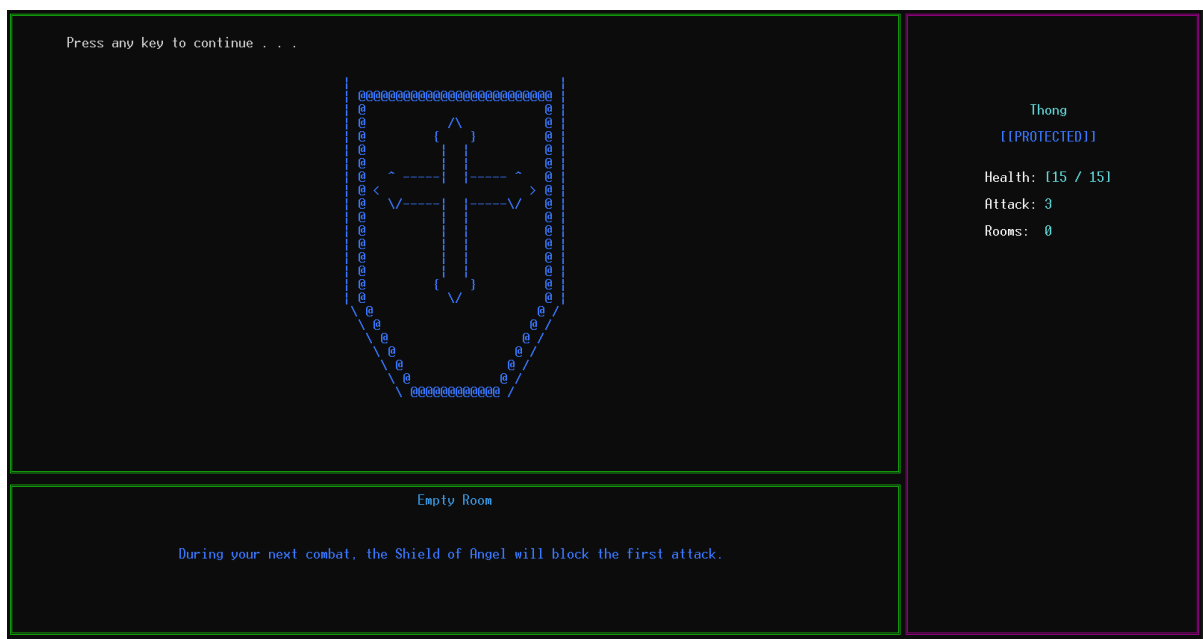
**processRoom()**: draw art and request the player to enter inputs for praying (restore their health) and picking up the items (as there is a **Cactus** which gives a negative effect and reduces 1 HP)

There are 3 items that currently can be found in the Empty Room:

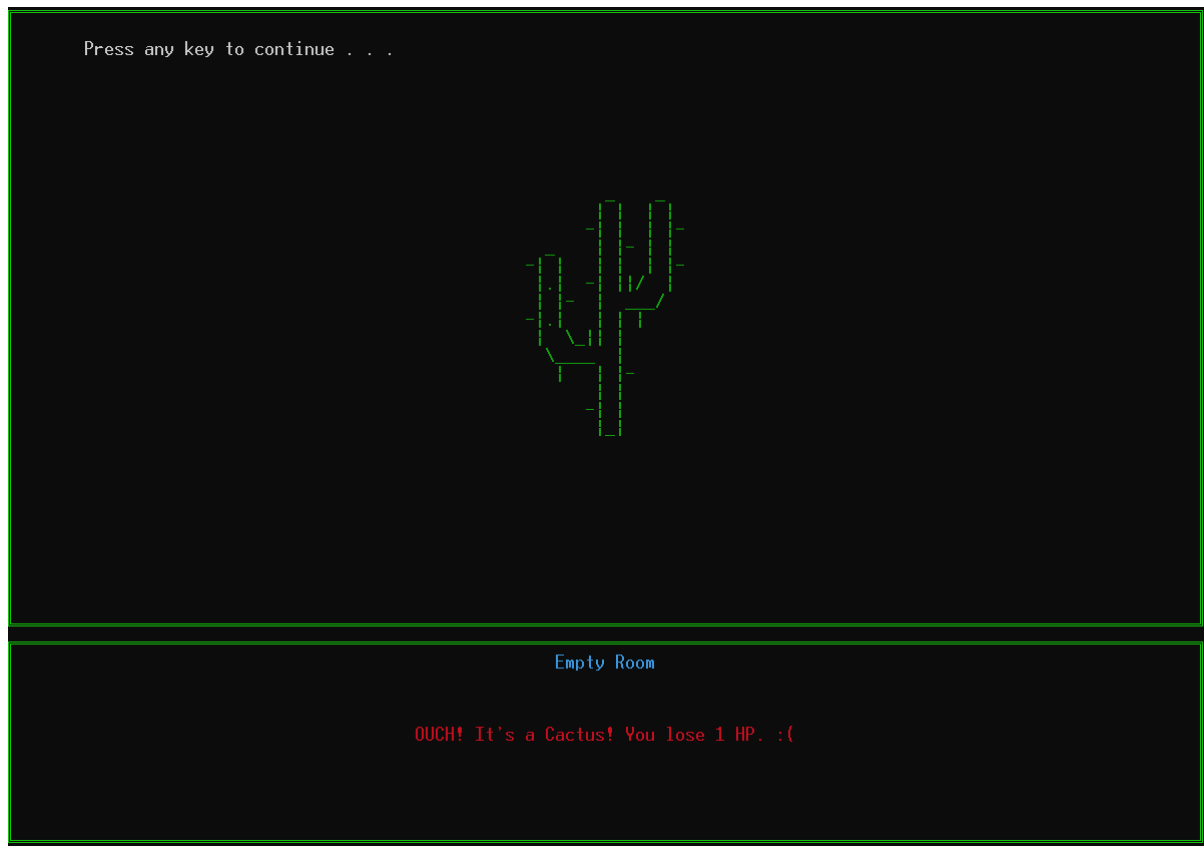
+ **The Sword of Hero**: Increase the Attack Point of the player by 1.



+ **The Shield of Angel:** Block the first attack from a monster.



+ **Cactus:** Reduce 1 HP of the player



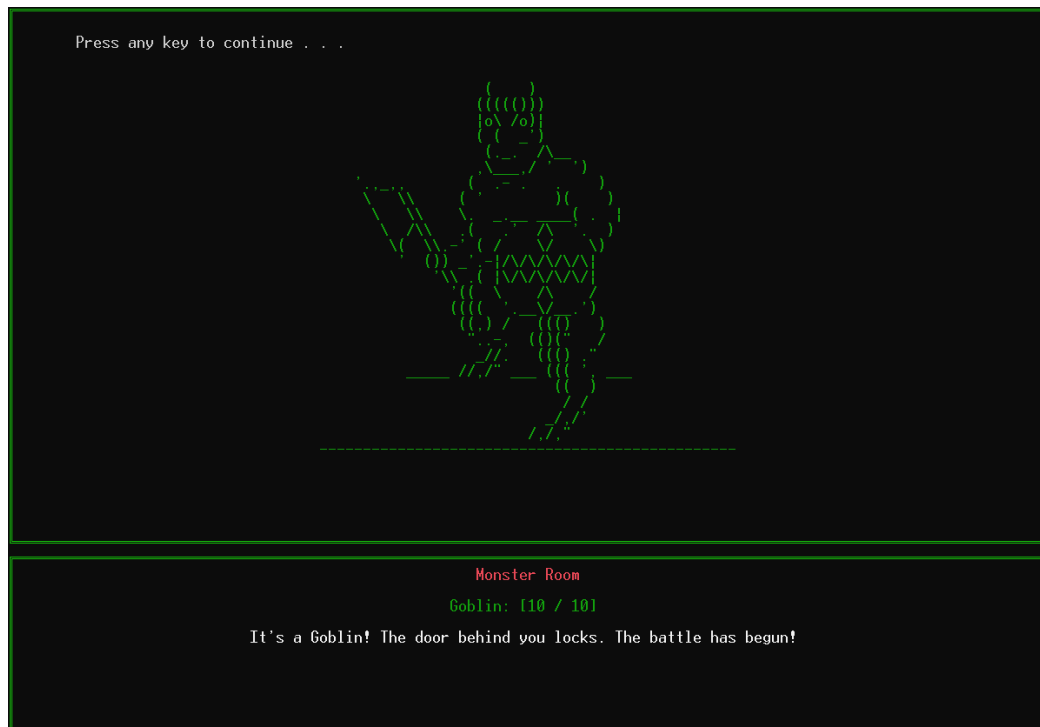
It is not always guaranteed to have an item spawned in the Empty Room. Each of the items also has different randomized chances to appear as follows:

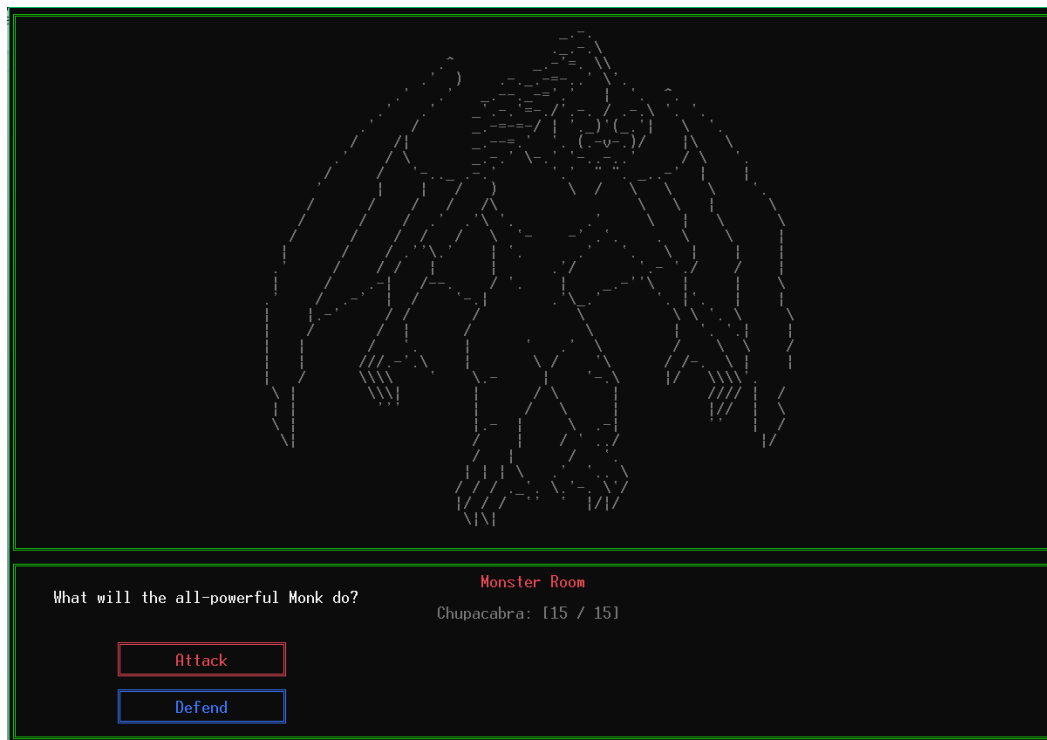
```
int ITEM_CHANCE = 20;  
int SWORD_CHANCE = 20;  
int SHEILD_CHANCE = 30;  
int CACTUS_CHANCE = 50;
```

In which the rarest one is the **Sword of Heros**.

### **Monster Room:**

There are currently 3 types of monsters with different characteristics that can appear in the **Monster Room**:

**Goblin****Foot**



### Chupacabra

```
class MonsterRoom : public Room {
private:
    Monster* monster;
public:
    MonsterRoom(int monsterId = generateRand(0, 2));
    ~MonsterRoom();

    bool processActions(Entity *E1, Entity *E2, int E1_action, int E2
_action);

    bool processRoom(Player* player);
};
```

Attributes:

**monster:** A pointer contains the monster in the room, which is randomly initialized and assigned from the constructor.

Methods:

**processActions():** Automatically simulate and generate the success chances for the fighting actions between the Player and Monster. Also, check for the health condition of both sides to terminate the fight and update them on the screen.

**processRoom():** Draw the monster's ASCII Art model and call the processActions() to process the fight.

### Treasure Room:

```
class TreasureRoom : public Room {  
public:  
    TreasureRoom();  
    bool processRoom(Player* player);  
};
```

Methods:

**processRoom():** Print the prompts to ask the player to collect the treasure (the player to still choose to ignore it and go to the next room to continue to explore). After collecting the treasure, the game will be over.

### GameHandler:

A class manages all the objects and processes the game.



```
class GameHandler {  
private:  
    Player* player;  
    Room* curRoom;  
  
    int roomsExplored;  
  
    //Catch other buttons from the keyboard to check whether it is a  
    command or not  
    string buf;
```

Attributes:

**player:** The pointer contains player's object

**curRoom:** The pointer contains the current room's object

**roomsExplored:** Counter for how many rooms have been explored so far by the player.

**buf:** catch the buf from the keyboard to compare with the command.

Methods:

**processGame():** draw the game's GUI and status of the player, also start a different thread to catch the buffer from the keyboard. Also randomly generated left and right room for the current room and displayed the prompt for the player to choose which room needs to go next.

```
void GameHandler::processGame()
{
    drawGame();

    thread sub_thread(&GameHandler::subThread, this);

    while (true) {
        curRoom->displayRoomNameAndDesc();
        if (!curRoom->processRoom(player))
            break;

        GenerateNewRooms();
        MoveRoom();
    }

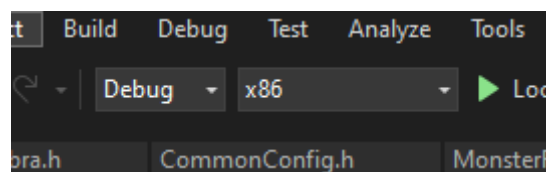
    sub_thread.detach();
}
```

### 3) Extre Features

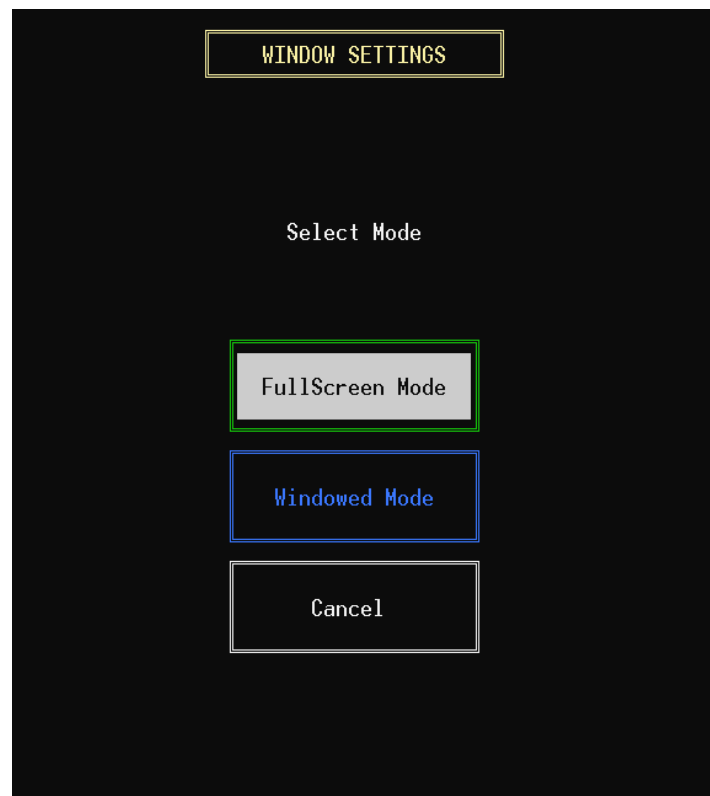
The game also has some extra features that can be extended further in the future:

#### **Window Full Screen Mode:**

- Because all the drawing work is done on a parameterized global screen dimension configuration. So the application can be responsive and flexible with any screen.
- Make sure to run the on x86 platforms instead of x64 to use this feature (The window full mode function can only be used in x86 platform):

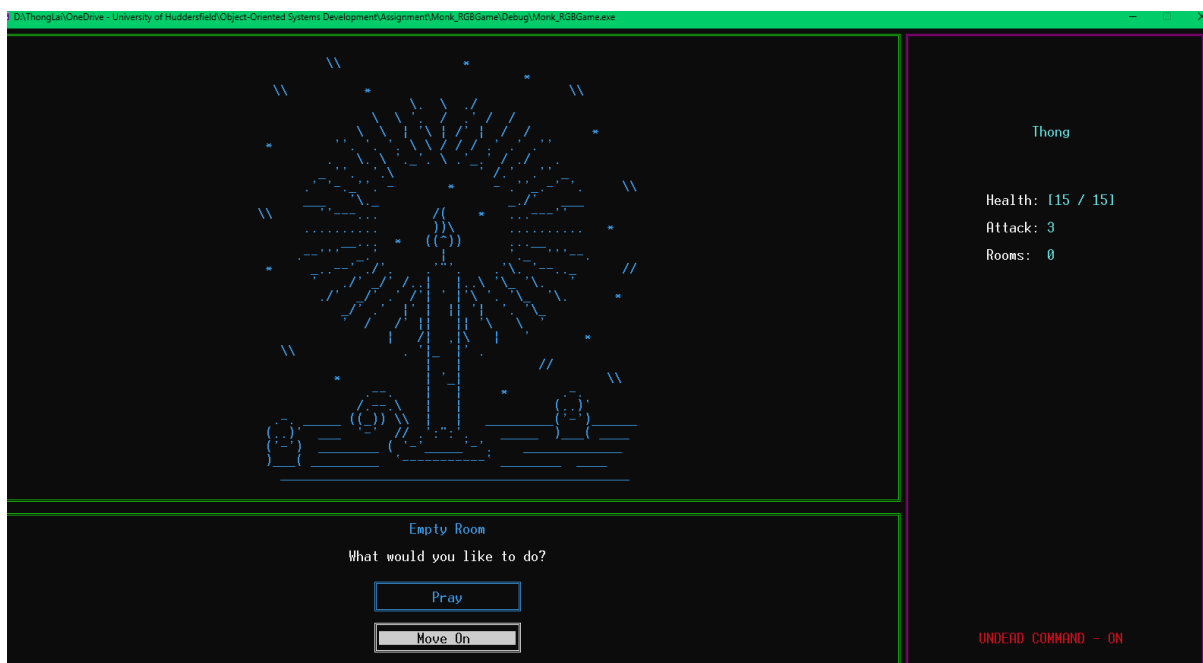


- To experience the game in full mode: Go to Settings->Windows->FullScreen Mode



Cheat Code:

- There is also a secret code to activate the Undead mode to prevent the player from taking any damage.
- In the middle of the gameplay, type T-H-O-N-G ("THONG") to activate it:



- There will be a red display to let the player know they cannot take any more damage.
- Typing the same command will undo this mode.

---

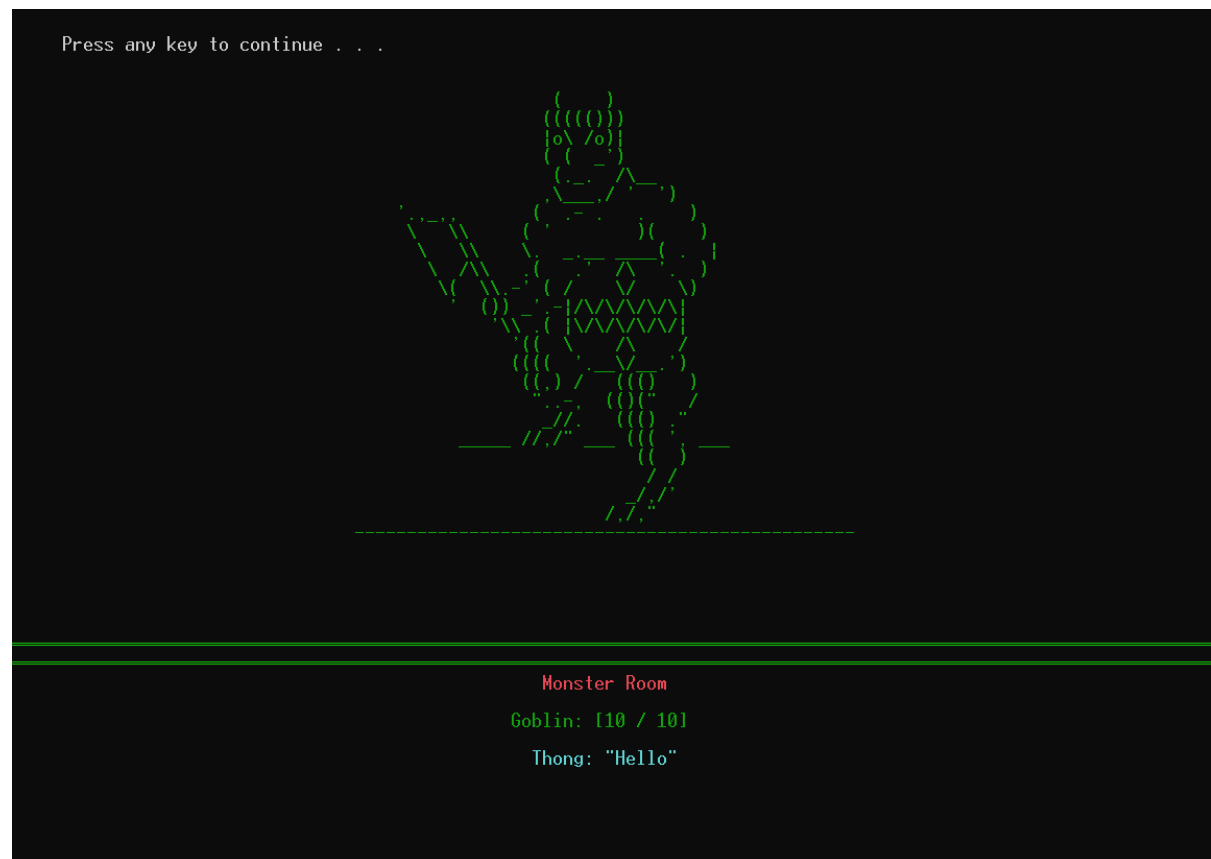
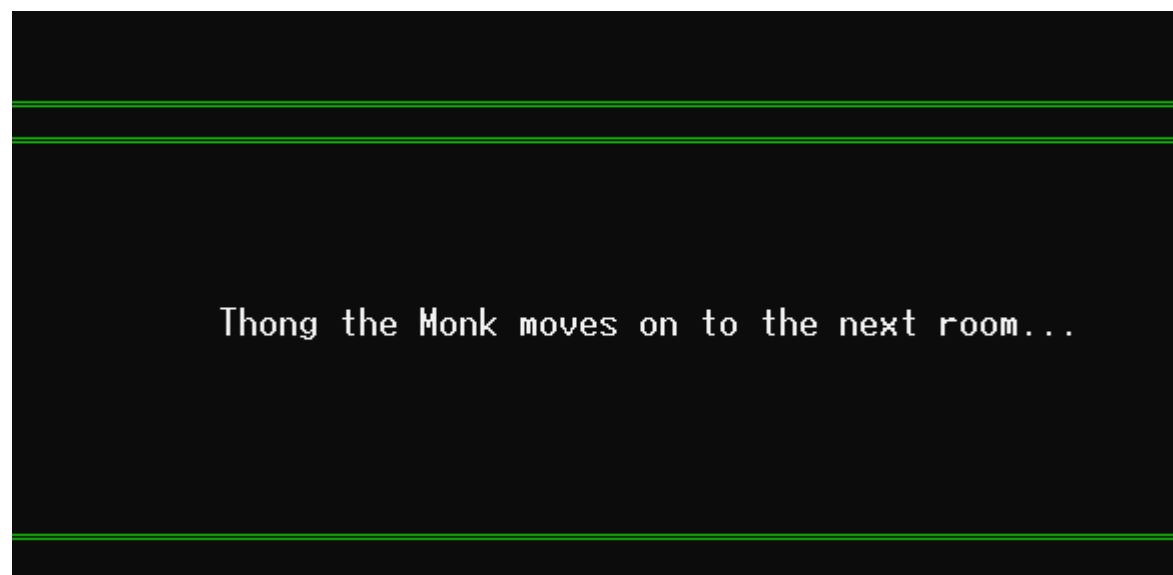
## III: Test Cases

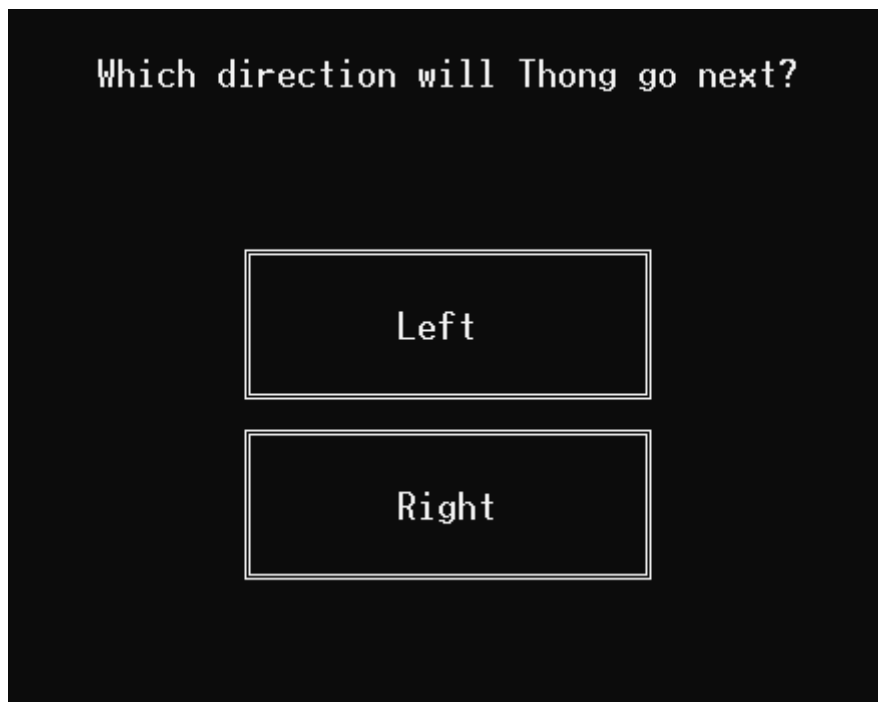
---

### Enter name and description:

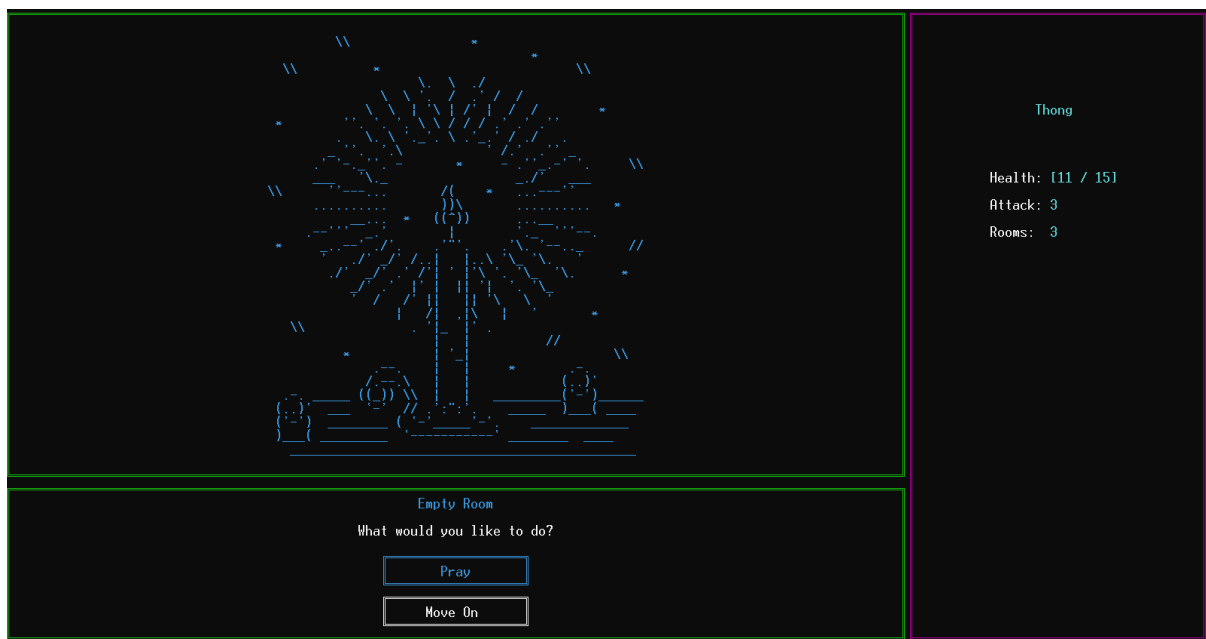




**Move between rooms:**

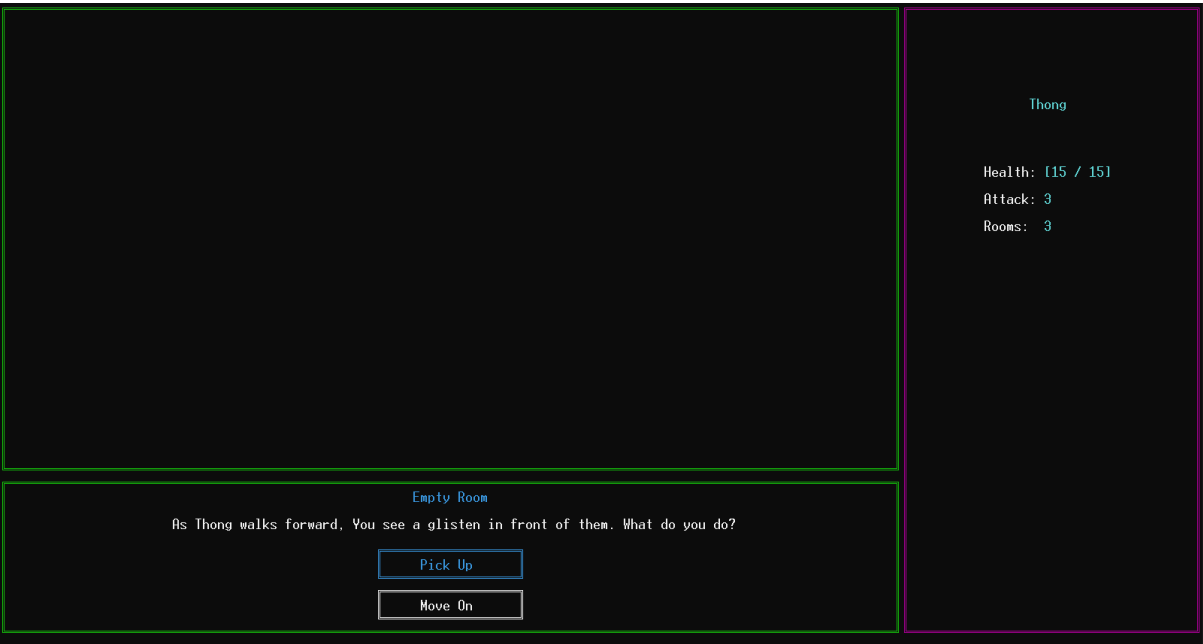


**Restore health when praying:**






**Pick up item:**





Press any key to continue . . .



Thong

Health: [15 / 15]


Attack: 3

Rooms: 3

Empty Room

It's the Sword of Hero!

Press any key to continue . . .



Thong

Health: [15 / 15]

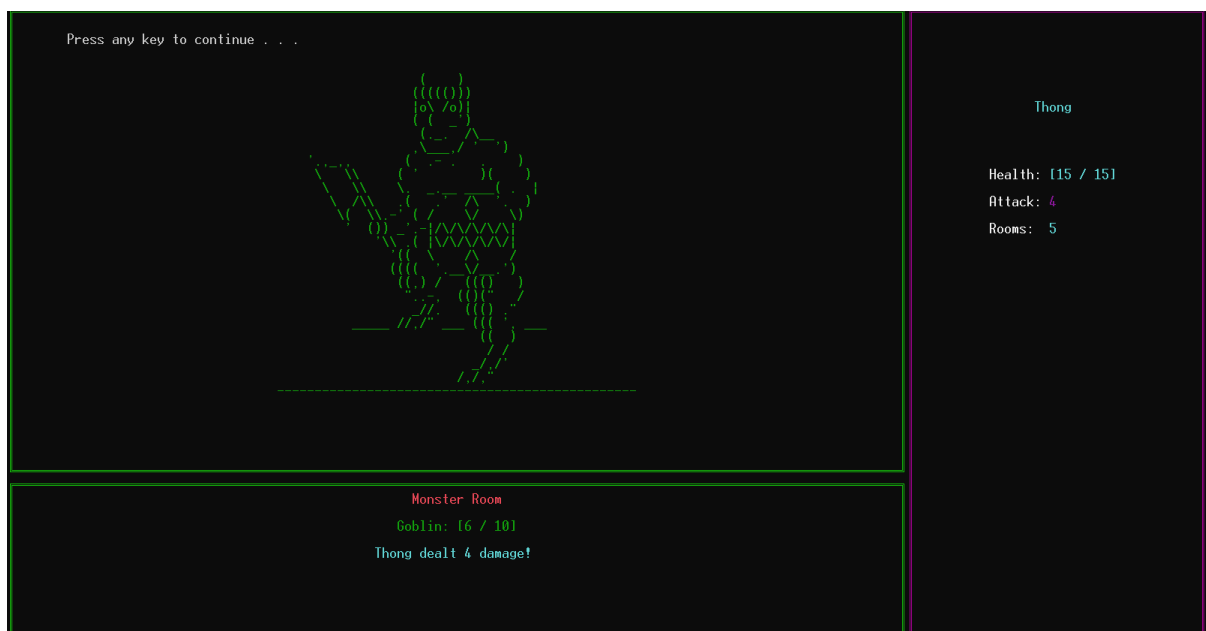
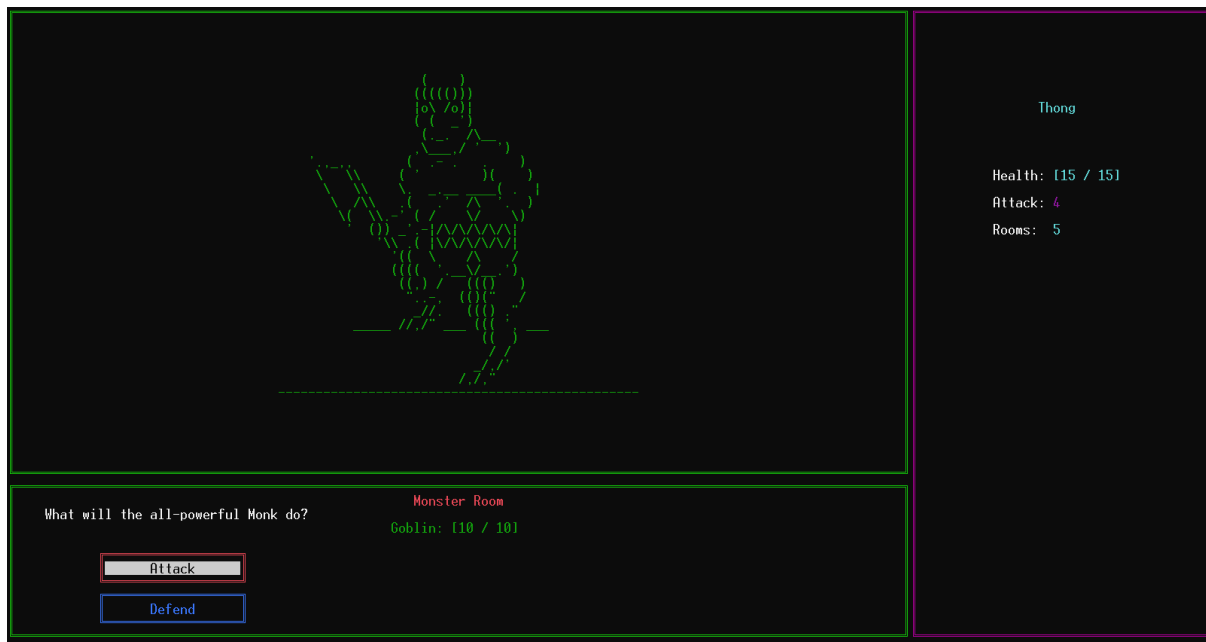
Attack: 4

Rooms: 3

Empty Room

With the Sword of Hero, your Attack Damage is increased by 1.

**Fighting Monsters:**



**Collect Treasure and end game:**



---

## IV: Evaluation

---

- The game uses inheritance extensively, with classes like Monster, Goblin, Entity, etc. This makes the program semantically clearer and promotes code reusability.
- GameHandler class to manage all objects and process the game. It also uses a multithreading approach to catch the buffer from the keyboard. The game also uses classes like **BOX**, **MENU**, and **Status** to provide a user-friendly interface. includes features like full-screen mode and a cheat code for an "Undead" mode. It also includes a variety of room types, monsters, and items for the player to interact with.
- It effectively uses OOP principles and provides a user-friendly interface with a variety of features and has been thoroughly tested to ensure its functionality.
- With these implementations, it also can be extended further with more Monster, items, or other functionality like saving the game or leveling easily.

---

## VI: Conclusions

---

This project focused on an object-oriented programming design approach and applied as much of OOP learned knowledge as possible such as:

- + Inheritance: an important knowledge of this project, many classes that have inheritance relationships such as Monster – Goblin, and Monster – Entity, ... They make the program semantically clearer and easy to reuse code.
- + Encapsulation: many classes are created to make sure that all methods and properties can be used flexibly. It makes code easy to reuse and protects private/protected properties from external influences.
- + Polymorphism: there are many “child” classes of a base class (Entity – Monster, Entity – Player) but their methods do not always have the same processing logic (Monster and Player have **displayHealth()** method, but the Monster’s health is displayed on **Description Screen**, and the Player’s health display on **Status Screen**). We need to use polymorphism for many methods so that their “child” classes can have their own way of working.
- + Abstract: some pure virtual methods are built such as (ex: Room doesn’t have **processRoom** method in definition). And, when we are working with the main thread or sub-thread, we don’t need to remember the implementation of some functions, we just need to know what they can do and use their result.
- + There is also other OOP knowledge that is implemented in the code such as friend (to overload some operators, ...), template (used in check-in impact of Player), working with files (to store data), ....

The project also needs some knowledge about multithreading in C/C++, handling with Windows console and font functions, getting the keyboard’s buffer, ...

All these pieces of knowledge are combined to create a project that fully meets the requirements of the topic with optimized UX-UI.