UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Santa's Escape Room

A Pygame-Based Grid Navigation Game

Technical University of Cluj-Napoca,
Department of Computer Science

Authors:
Pop Maria Alexandra
Pop Alexandra
Group: 30435

Laboratory Supervisor:
Ing. Andrei Dumitraș

Course Supervisor:
Assoc. Prof. Dr. Eng. Adrian Groza

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Contents

1. Abstract

Santa's Escape Room is a grid-based puzzle game where players guide Santa to collect presents, avoid obstacles, and outsmart the Grinch. The game features an **AI-driven navigation system** powered by **Prover9**, which uses logical inference to process clues and determine optimal moves, blending manual and autonomous gameplay.

Inspired by **Wumpus World**, the game incorporates grid navigation challenges, clue-based decision-making, and adaptive AI strategies, offering an engaging platform for exploring logic-based game design.

2. Introduction

2.1 Background:

Escape room games are puzzle-solving experiences where players navigate challenges and uncover clues to achieve specific goals. These games emphasize logic, critical thinking, and strategic problem-solving in immersive environments.

Inspired by **Wumpus World**, a grid-based AI problem-solving scenario, **Santa's Escape Room** incorporates similar mechanics: navigating a grid with hidden dangers, interpreting clues, and making strategic decisions to ensure success. This blend of logic and dynamic challenges creates an engaging and festive gameplay experience.

2.2 Objective:

Implement a dynamic escape room game using both manual and AI-driven autonomous navigation with Prover9.

2.3 Overview:

Santa's Escape Room is a grid-based puzzle game where players guide Santa to collect presents, avoid obstacles, and escape through the chimney. The gameplay blends manual controls with AI-driven navigation powered by **Prover9**, enabling strategic decision-making.

The game's technical implementation uses Python and Pygame for grid rendering, animations, and interactive gameplay, while Prover9 handles logical reasoning for AI navigation, creating a challenging and immersive experience.

3. Game Concept

3.1 Game Story:

On Christmas Eve, Santa sneaks into a house to deliver presents for the kids, but trouble follows as the mischievous Grinch sneaks in behind him, determined to steal the presents. To escape, Santa must avoid leaving any trace of his visit—especially the flour traps the kids have set, which stick to his boots and reveal his steps. Adding to the challenge, Santa needs energy to outrun the Grinch, which he gains by eating all the cookies the kids left for him, cleverly disguised as presents.

The final goal is for Santa to find the chimney, signaled by a cold breeze, and escape to his sleigh without getting caught or leaving any clues behind. The adventure is filled with dynamic challenges and clever clues to guide Santa safely through the house.

Objects and Their Clues

Presents (Cookies):

- Clue: A "cookie smell" (yellow visual indicator) appears in adjacent cells, hinting at the delicious cookies hidden in presents nearby. Santa must collect all presents to gain energy.

Exit (Chimney):

- Clue: A "cold breeze" (light blue visual indicator) signals the chimney is near, guiding Santa to the escape route where his sleigh awaits.

Obstacles (Flour Traps):

- Clue: A "flour smell" (dark gray visual indicator) warns Santa of traps where flour has been scattered. If Santa steps on the flour, it will stick to his boots, revealing his presence to the kids.

Grinch:

- Clue: A "Grinch sound" (dark green visual indicator) warns Santa of the Grinch's proximity. The Grinch moves unpredictably, creating danger zones that Santa must avoid to protect the presents.

3.2 Rules:

- Santa must collect all presents to unlock the exit.
- Grinch moves randomly but interacts dynamically with Santa's position.
- Obstacles block certain paths and provide environmental challenges.

4. Theoretical Background

4.1. Wumpus World Inspiration:

"The Wumpus world is a cave which has 4/4 rooms connected with passage ways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow. In the Wumpus world, there are some Pits rooms which are bottomless, and if agent falls in Pits, then he will be stuck there forever. The exciting thing with this cave is that in one room there is a possibility of nding a heap of gold. So the agent goal is to nd the gold and climb out the cave without fallen into Pits or eaten by Wumpus. The agent will get a reward if he comes out with gold, and he will get a penalty if eaten by Wumpus or falls in the pit2" [1]

Similarities between Wumpus World and Santa's Escape Room

- *Grid-Based Navigation:*
  Both games feature environments divided into grid-like rooms or cells connected by passages, requiring strategic movement.
- *Clue-Based Decision Making:*
  In Wumpus World, the agent uses sensory clues (stench, breeze) to infer the location of dangers like the Wumpus or pits. Similarly, Santa interprets clues like "cookie smells," "flour smells," "cold breezes," and "Grinch sounds" to navigate safely.
- *Hidden Dangers:*
  Wumpus World features hidden dangers (Wumpus and pits) that can lead to the agent's failure. Santa's Escape Room incorporates dynamic obstacles like the Grinch and flour traps, which Santa must avoid to succeed.
- *Reward and Escape:*
  In Wumpus World, the agent's goal is to collect gold and exit the cave safely. In Santa's Escape Room, Santa must collect presents (cookies) for energy and escape the house.
- Strategic Path Planning:
  Both games require careful path planning based on partial knowledge, as the player or agent gathers information progressively through exploration.
- *AI Application:*
  Wumpus World uses a knowledge-based agent for logical inference and safe navigation. Similarly, Santa's Escape Room uses Prover9 for logical reasoning, enabling Santa to make informed moves based on environmental clues.

4.2. First-Order Logic

“ First-order logic (FOL***)***, also known as predicate logic or first-order predicate calculus, is a powerful framework used in various fields such as mathematics, philosophy, linguistics, and computer science. In artificial intelligence (AI), FOL plays a crucial role in knowledge representation, automated reasoning, and natural language processing

First Order Logic extends propositional logic by incorporating quantifiers and predicates, allowing for more expressive statements about the world. The key components of FOL include constants, variables, predicates, functions, quantifiers, and logical connectives.

1. **Constants**: Constants represent specific objects within the domain of discourse. For example, in a given domain, Alice, 2, and NewYork could be constants.
2. **Variables**: Variables stand for unspecified objects in the domain. Commonly used symbols for variables include x, y, and z.
3. **Predicates**: Predicates are functions that return true or false, representing properties of objects or relationships between them. For example, Likes(Alice, Bob) indicates that Alice likes Bob, and GreaterThan(x, 2) means that x is greater than 2.
4. **Functions**: Functions map objects to other objects. For instance, MotherOf(x) might denote the mother of x.
5. **Quantifiers**: Quantifiers specify the scope of variables. The two main quantifiers are:
    - **Universal Quantifier (∀)**: Indicates that a predicate applies to all elements in the domain.
        - For example, ∀x (Person(x) → Mortal(x)) means "All persons are mortal."
    - **Existential Quantifier (∃)**: Indicates that there is at least one element in the domain for which the predicate holds.
        - For example, ∃x (Person(x) ∧ Likes(x, IceCream)) means "There exists a person who likes ice cream."
6. **Logical Connectives**: Logical connectives include conjunction (∧), disjunction (∨), implication (→), biconditional (↔), and negation (¬). These connectives are used to form complex logical statements..” [3]

4.3. Prover9

“Prover9 is an automated theorem prover for first-order and equational logic, and Mace4 searches for finite models and counterexamples.” [2]
Prover9 is specifically built to:

- Automate Reasoning: Prove theorems and validate logical assertions.
- Handle First-Order Logic: Work with predicates, quantifiers, and logical rules efficiently.
- Solve Decision Problems: Identify whether a move, path, or state satisfies the game’s constraints.

4.4. PyGame

"Pygame is a cross-platform set of Python modules designed for writing video games. It includes computer graphics and sound libraries designed to be used with the Python programming language." [4]

In a project, Pygame can be structured into modules for **game logic**, **graphics rendering**, and **user interaction**. The core of a Pygame project revolves around the **game loop**, which continuously updates the game state, processes user inputs, and renders visuals to the screen.

Pygame Core Functions and Descriptions

- pygame.init() - Initializes all Pygame modules.
- pygame.display.set_mode() - Sets up the game window.
- pygame.draw.* - Draws shapes (e.g., rectangles, circles, lines).
- pygame.image.load() - Loads an image into the game.
- pygame.mixer.Sound() - Plays sound effects.
- pygame.mixer.music - Plays background music.
- pygame.sprite.Sprite - Base class for game objects with built-in functionality like rendering and collision handling.

5. Use Cases

"A use case is a concept used in software development, product design, and other fields to describe how a system can be used to achieve specific goals or tasks. It outlines the interactions between users or actors and the system to achieve a specific outcome.." [5]

1. Menu Navigation

Result: Player starts the game, views instructions, or exits.
Steps:

1. The main menu displays three options: **Start Game**, **Instructions**, and **Exit**.
2. The player uses **arrow keys** to navigate and **Enter** to confirm.
    - **Start Game:** Launches the game.
    - **Instructions:** Displays gameplay rules and controls.
    - **Exit:** Closes the game window.

2. Collecting Presents

Result: Santa collects presents, progressing toward the exit.
Steps:

1. The player moves Santa using **arrow keys**.
2. When Santa reaches a present:
    - The present is removed from the grid.
    - A feedback message is displayed:
      "Present collected!"
3. Santa must collect all presents to unlock the exit.

3. Hitting an Obstacle

Result: The game ends if Santa steps into an obstacle.
Steps:

1. Santa moves into a cell containing an obstacle.
2. The move is invalid, and a pop-up message is displayed:
   "The kids outsmarted you! Your steps are uncovered with flour."
3. The game ends immediately, transitioning back to the main menu.

4. Grinch Collision

Result: The game ends if Santa collides with the Grinch.
Steps:

1. The Grinch moves randomly every 2 seconds.
2. If the Grinch's position matches Santa's position:
    - A collision is detected.
    - A pop-up message is displayed:
      "Grinch stole the Christmas!"
3. The game ends immediately.

5. Winning the Game

Result: Santa collects all presents and escapes through the chimney.
Steps:

1. After collecting all presents, the "cold breeze" clue appears near the exit.
2. Santa moves to the exit point.
3. A victory pop-up is displayed:
   "Santa saved the Christmas!"

6. Autonomous Mode

Result: AI controls Santa to collect presents and escape.
Steps:

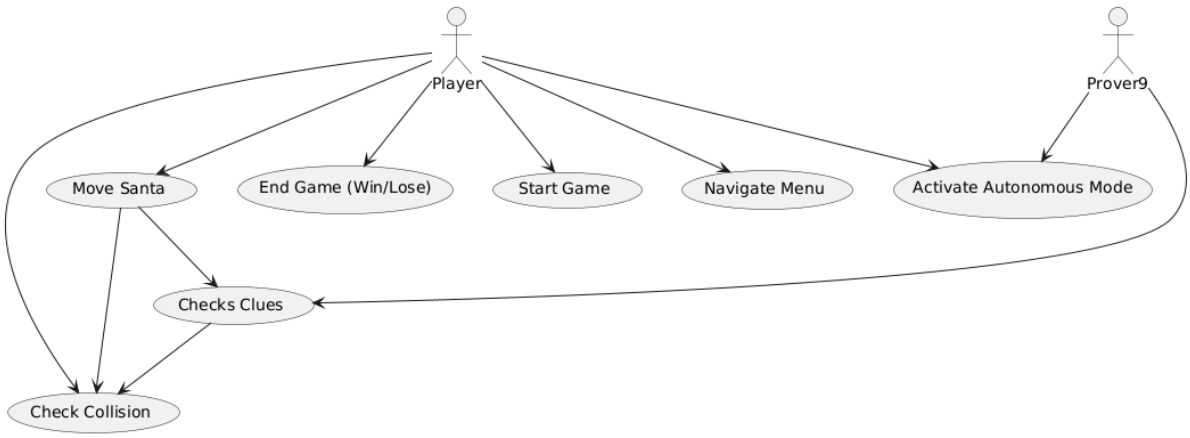
1. The player presses **Enter** during gameplay to activate AI mode.
2. The AI:
    - Analyzes Santa's position and known clues.
    - Calculates the next move using logical rules.
    - Updates Santa's position automatically.
3. Feedback messages display AI decisions (e.g., "Prover9 determined the next move.").
4. The AI continues until Santa escapes or fails.

7. Implementation

6.1 Software Architecture:

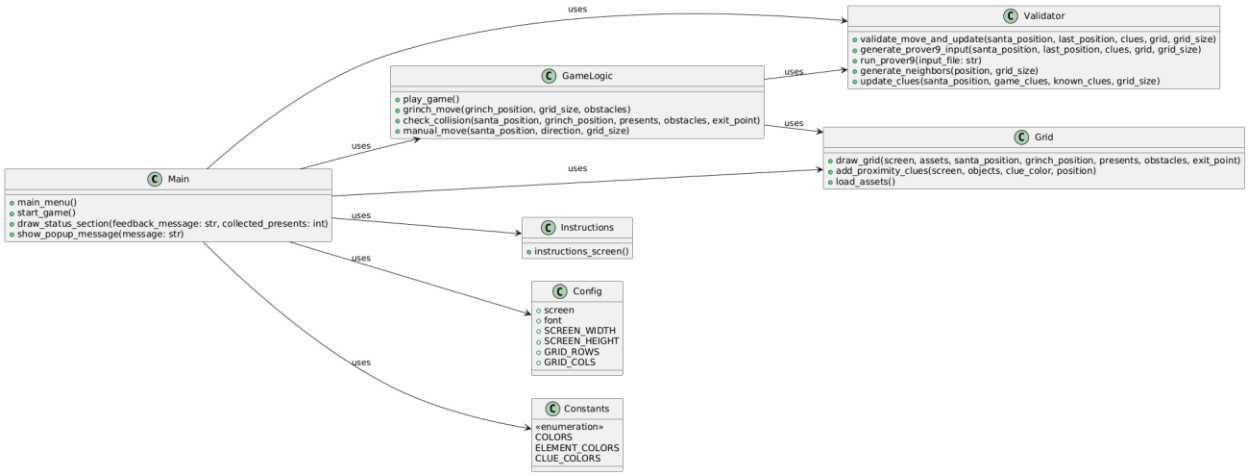
Use Case Diagram

A **use case diagram** illustrates the interactions between the system's users (actors) and its functionalities (use cases). It helps to visualize how the system will be used and what tasks it can perform.



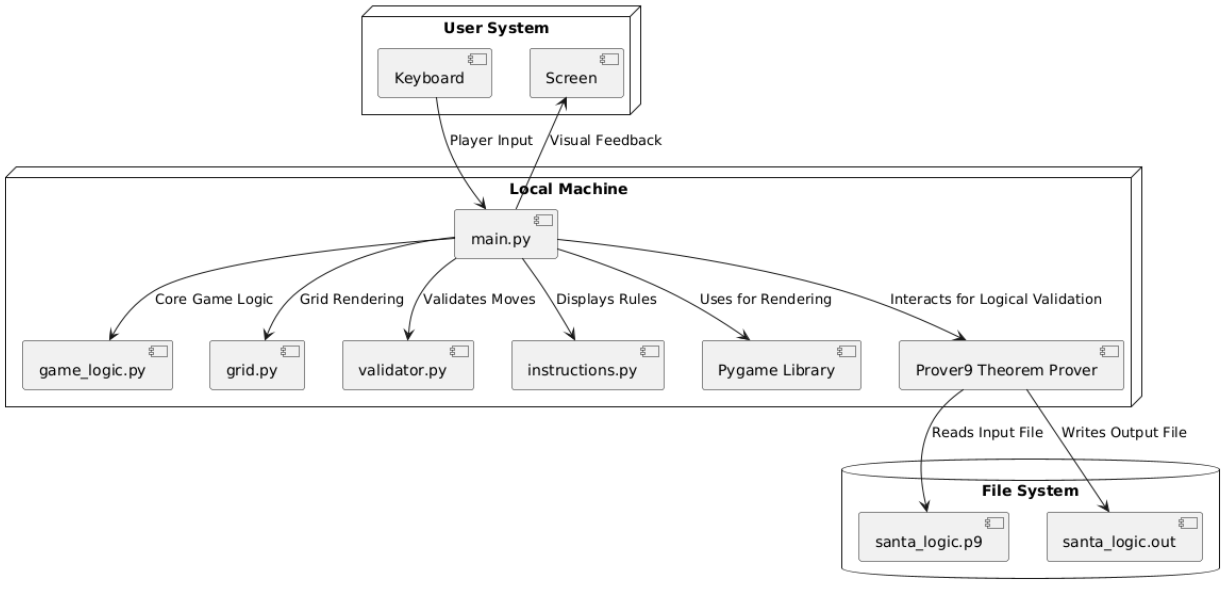
Class Diagram

A **class diagram** is a type of UML diagram that represents the static structure of a system by showing its classes, their attributes, methods, and relationships.



Deployment Diagram

The deployment diagram for **Santa's Escape Room** showcases how the project's software components (main.py, game_logic.py, etc.) interact with external tools (e.g., **Prover9**) and hardware elements like the user's keyboard and screen.



6.2 Tools and Libraries:

- Pygame for graphics and gameplay mechanics.
- Prover9 for logical reasoning and decision-making.

6.3 Code Organization:

the project is organized into modular Python files, each focusing on a specific aspect of the game. Additionally, it integrates Prover9 logic through external files for advanced AI decision-making.

1. **main.py**

Purpose: Serves as the entry point for the game, managing the game flow, user input, and interactions between all components.
Details:

- **Key Functions:**
  - main_menu(): Displays the main menu, allowing the user to start the game, view instructions, or exit.
  - start_game(): Initializes and runs the main game loop, handling both manual and autonomous gameplay modes.
  - draw_status_section(): Displays the current status of the game, such as collected presents and feedback messages.
  - show_popup_message(): Displays pop-up messages for events like game completion or collision with the Grinch.
- **Interactions:**
  - Integrates modules like game_logic, validator, and grid to handle gameplay mechanics.
  - Directly interacts with Pygame for rendering and user input.

2. **config.py**

Purpose: Configures the screen dimensions, fonts, and other essential settings required for rendering the game window.
Details:

- **Key Variables:**
  - screen: The Pygame screen object for rendering game elements.
  - font: Font configuration for displaying text in the game.
  - SCREEN_WIDTH and SCREEN_HEIGHT: Dimensions of the game window.
  - STATUS_HEIGHT: Height reserved for the status section.
- **Interactions:**
  - Used by main.py and grid.py to set up and render the game visuals.

3. **constants.py**

Purpose: Stores all the game's constant values to ensure consistency and ease of maintenance.
Details:

- **Key Constants:**
  - COLORS: Defines RGB color codes for various game elements (e.g., grid lines)
  - ELEMENT_COLORS: Specifies colors for objects like Santa, the Grinch, presents, and obstacles.
  - CLUE_COLORS: Assigns colors to visual clues such as yellow= "cookie smell"
  - GRID_ROWS and GRID_COLS: Determines the size of the grid.
  - CELL_SIZE: Calculates the size of each cell based on screen width/grid columns.
- **Interactions:**
  - Used throughout the project to ensure consistency in visual/ game settings.

4. **grid.py**

Purpose: Manages the rendering of the game grid and all visual elements, including Santa, presents, obstacles, and clues.
Details:

- **Key Functions:**
  - load_assets(): Loads and scales image assets for grid elements (e.g., Santa0
  - draw_grid(): Draws the grid and places game elements like Santa, presents, obstacles, and the Grinch.
  - add_proximity_clues(): Dynamically generates visual clues near game objects.
  - draw_legend(): Displays a legend explaining visual clues like cookie smells
- **Interactions:**
  - Called by main.py during each game loop to update the game visuals.

5. **game_logic.py**

Purpose: Implements the core gameplay rules and mechanics, handling interactions between Santa, the Grinch, and other game elements.
Details:

- **Key Functions:**
  - play_game(): Handles gameplay logic for both manual and autonomous modes.
  - grinch_move(): Moves the Grinch randomly within the grid, avoiding obstacles.
  - check_collision(): Detects interactions between Santa and objects like presents, obstacles, and the Grinch.
  - manual_move(): Processes player input to move Santa within the grid.
  - determine_next_move(): Uses Prover9 or fallback logic to decide Santa's move.
- **Interactions:**
  - Used by main.py to control gameplay logic and ensure rule adherence.

6. **validator.py**

Purpose: Facilitates AI navigation by integrating **Prover9** for logical reasoning and decision-making.

Details:

- **Key Functions:**
  - generate_prover9_input(): Creates an input file (santa_logic.p9) for Prover9 with grid relationships and clues.
  - run_prover9(): Executes Prover9 using the generated input and processes its output (santa_logic.out).
  - validate_move_and_update(): Uses Prover9 output or fallback logic to decide Santa's next move.
  - generate_neighbors(): Identifies valid neighboring cells around Santa.
  - update_clues(): Updates Santa's knowledge of nearby objects based on clues.
- **Interactions:**
  - Used by game_logic.py for Prover9-based autonomous decision-making.

7. **instructions.py**

Purpose: Displays the game instructions to the user before starting the game.

Details:

- **Key Functions:**
  - instructions_screen(): Renders the instructions screen, providing gameplay rules and controls.
- **Interactions:**
  - Invoked by main_menu() in main.py.

8. **Prover9 Files**

Purpose: Used for logical decision-making in AI-controlled gameplay.

Details:

- **santa_logic.p9:**
  - Dynamically generated input file for Prover9.
  - Contains first-order logic rules, grid relationships, and clues based on Santa's position.
- **santa_logic.out:**
  - Output file generated by Prover9 after processing santa_logic.p9.
  - Indicates whether a valid move exists and provides logical reasoning for navigation.

Interactions:

- Managed by validator.py to integrate Prover9 into the game.

6.4 Key Functions and Logic:

1.Santa's Movement Logic

- **Functions:**
  - manual_move(santa_position, direction, grid_size) (from game_logic.py)

```
def manual_move(santa_position, direction, grid_size):
    """
    Moves Santa manually based on player input.
    """
    new_position = santa_move(santa_position, direction)
    if 0 <= new_position[0] < GRID_ROWS and 0 <= new_position[1] < GRID_COLS:
        return new_position
    return santa_position
```

  - check_collision(santa_position, grinch_position, presents, obstacles, exit_point) (from game_logic.py)

```
def check_collision(santa_position, grinch_position, presents, obstacles, exit_point):
    """
    Handles collisions with obstacles, presents, the Grinch, and the exit.
    """
    santa_tuple = tuple(santa_position)

    # Check if Santa is caught by the Grinch
    if santa_tuple == tuple(grinch_position):
        return "Santa caught by the Grinch! Game Over!"

    # Check for presents
    if santa_tuple in presents:
        presents.remove(santa_tuple)
        return "Present collected!"

    # Check for obstacles
    if santa_tuple in obstacles:
        return "Blocked by an obstacle!"

    # Check for exit
    if santa_tuple == exit_point:
        if len(presents) == 0:
            return "Congratulations! You've saved Christmas!"
        else:
            return "Collect all presents before exiting!"

    return "Move successful!"
```

2.Grinch Movement

- **Function:**
  - grinch_move(grinch_position, grid_size, obstacles) (from game_logic.py)

```python
def grinch_move(grinch_position, grid_size, obstacles):
    """
    Moves the Grinch randomly in one of the four directions: up, down, left, or right.
    Grinch avoids obstacles and respects grid boundaries.
    """
    directions = [
        (0, -1),  # Left
        (0, 1),   # Right
        (-1, 0),  # Up
        (1, 0),   # Down
    ]

    random.shuffle(directions)  # Shuffle directions to make movement random

    for dx, dy in directions:
        new_x = grinch_position[0] + dx
        new_y = grinch_position[1] + dy

        # Check if the new position is within grid boundaries and not an obstacle
        if 0 <= new_x < grid_size[0] and 0 <= new_y < grid_size[1]:
            if (new_x, new_y) not in obstacles:
                return [new_x, new_y]

    # If no valid move is found, stay in the same position
    return grinch_position
```

3.Grid Rendering and Visual Elements

- **Functions:**
  - draw_grid(screen, assets, santa_position, grinch_position, presents, obstacles, exit_point) (from grid.py)

```python
def draw_grid(screen, assets, santa_position, grinch_position, presents, obstacles, exit_point):
    """
    Draws the game grid based on the provided positions for Santa, Grinch, presents, obstacles, and the exit.
    Santa can occupy the same position as an object temporarily (overwriting).
    """
    screen.fill(COLORS["background"])

    # Draw the grid
    for row in range(GRID_ROWS):
        for col in range(GRID_COLS):
            rect = pygame.Rect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE)
            pygame.draw.rect(screen, COLORS["grid"], rect, 1)

    # Add proximity indicators for objects
    add_proximity_clues(screen, presents, CLUE_COLORS["cookie_smell"], "top_left")  # Clues for presents
    add_proximity_clues(screen, obstacles, CLUE_COLORS["flour_smell"], "bottom_right")  # Clues for obstacles
    add_proximity_clues(screen, [exit_point], CLUE_COLORS["cold_breeze"], "top_right")  # Clues for exit
    add_proximity_clues(screen, [tuple(grinch_position)], CLUE_COLORS["grinch_sound"], "bottom_left")  # Clues for Grinch

    # Draw game elements, except Santa (drawn last to allow overwriting)
    for present in presents:
        if present != tuple(santa_position):
            screen.blit(assets["present"], (present[1] * CELL_SIZE, present[0] * CELL_SIZE))
    for obstacle in obstacles:
        if obstacle != tuple(santa_position):
            screen.blit(assets["obstacle"], (obstacle[1] * CELL_SIZE, obstacle[0] * CELL_SIZE))
    if exit_point != tuple(santa_position):
        screen.blit(assets["exit"], (exit_point[1] * CELL_SIZE, exit_point[0] * CELL_SIZE))
    if grinch_position != santa_position:
        screen.blit(assets["grinch"], (grinch_position[1] * CELL_SIZE, grinch_position[0] * CELL_SIZE))

    # Draw Santa last (overwriting other objects temporarily)
    screen.blit(assets["santa"], (santa_position[1] * CELL_SIZE, santa_position[0] * CELL_SIZE))
```

- add_proximity_clues(screen, objects, clue_color, position) (from grid.py)

```python
def add_proximity_clues(screen, objects, clue_color, position):
    """
    Adds visual clues dynamically based on proximity to specific elements.
    The clues are placed in specific parts of the adjacent cells:
        - "top_left": cookie smell (present clue)
        - "bottom_right": flour smell (trap)
        - "top_right": cold breeze (exit)
        - "bottom_left": grinch sound (grinch)
    """
    offsets = [
        (0, -1),  # Left
        (0, 1),   # Right
        (-1, 0),  # Up
        (1, 0),   # Down
    ]

    for obj in objects:
        for dx, dy in offsets:
            nx, ny = obj[0] + dx, obj[1] + dy
            if 0 <= nx < GRID_ROWS and 0 <= ny < GRID_COLS:
                rect_x = ny * CELL_SIZE
                rect_y = nx * CELL_SIZE
                if position == "top_left":
                    pygame.draw.circle(screen, clue_color, (rect_x + 10, rect_y + 10), 5)
                elif position == "bottom_right":
                    pygame.draw.circle(screen, clue_color, (rect_x + CELL_SIZE - 10, rect_y + CELL_SIZE - 10), 5)
                elif position == "top_right":
                    pygame.draw.circle(screen, clue_color, (rect_x + CELL_SIZE - 10, rect_y + 10), 5)
                elif position == "bottom_left":
                    pygame.draw.circle(screen, clue_color, (rect_x + 10, rect_y + CELL_SIZE - 10), 5)
```

Game Feedback via Pop-Up Messages

- **Function:**
    - show_popup_message(message) (from main.py)

```python
def show_popup_message(message):
    """
    Displays a pop-up message at the center of the screen and pauses for 3 seconds.
    """
    popup_font = pygame.font.Font(None, 40)
    formatted_message = format_popup_message(message)

    popup_height = len(formatted_message) * 50
    start_y = (SCREEN_HEIGHT - popup_height) // 2

    screen.fill(COLORS["background"])
    for i, line in enumerate(formatted_message):
        popup_surface = popup_font.render(line, True, ELEMENT_COLORS["grinch"])
        popup_rect = popup_surface.get_rect(center=(SCREEN_WIDTH // 2, start_y + i * 50))
        screen.blit(popup_surface, popup_rect)

    pygame.display.flip()
    pygame.time.delay(3000)
```

5. Menu and Game Initialization

- **Functions:**
  - main_menu() (from main.py)
  - start_game() (from main.py)

(for details, check Appendix because the code is too long to insert a screenshot here)

8. AI Decision Making

8.1 Using Prover9:

How Prover9 Processes Clues

1. **Input to Prover9: the function generate_prover9_input(santa_position, last_position, clues, grid, grid_size) from validator.py generates this file**
   - **Current Position:** Santa's current coordinates on the grid (e.g., (x, y)).
   - **Known Clues:** Information about nearby objects, including:
     - **Cookie Smell:** Indicates presents are nearby.
     - **Flour Smell:** Warns of obstacles in proximity.
     - **Cold Breeze:** Guides Santa to the exit.
     - **Grinch Sound:** Alerts Santa to avoid the Grinch.
   - **Grid State:** Relationships and attributes of adjacent cells, including:
     - Whether a cell is adjacent to presents, obstacles, the Grinch, or the exit.
     - Logical rules governing safety and valid moves.

Example of generated file:

santa_logic.p9

```
% --- Santa Escape Room Logic ---
% Propositions:
% cookie_smell(x, y), cold_breeze(x, y), grinch_sound(x, y), safe(x, y), move_to(x, y, u, v)
% present(x, y), grinch(x, y), exit(x, y), obstacle(x, y), adjacent(x, y, u, v)

% --- Adjacent Relations ---
adjacent(6, 6, 6, 5).
adjacent(6, 6, 6, 7).
adjacent(6, 6, 5, 6).
adjacent(6, 6, 7, 6).

% --- Rules ---
all x all y (
    cookie_smell(x, y) <-> exists u exists v (adjacent(x, y, u, v) & present(u, v))
).

all x all y (
    cold_breeze(x, y) <-> exists u exists v (adjacent(x, y, u, v) & exit(u, v))
).

all x all y (
    grinch_sound(x, y) <-> exists u exists v (adjacent(x, y, u, v) & grinch(u, v))
).

all x all y (
    safe(x, y) <-> ~grinch(x, y) & ~obstacle(x, y)
).

all x all y all u all v (
    move_to(x, y, u, v) <-> (
        adjacent(x, y, u, v) & safe(u, v) & (u != last_x | v != last_y)
    )
).

```

```
% --- Backup Rule: Move to a position without a Grinch clue ---
all x all y all u all v (
    backup_move(x, y, u, v) <-> (
        adjacent(x, y, u, v) & ~grinch_sound(u, v) & ~(u = last_x & v = last_y)
    )
).

% --- Observations ---
santa_position(6, 6).
last_position(6, 6).
obstacle(6, 7).
obstacle(5, 6).

% --- Goal: Find a safe move or backup move ---
goal: (exists u exists v (move_to(6, 6, u, v))) |
    (exists u exists v (backup_move(6, 6, u, v))).
```

2. **Processing Logic in Prover9: the function run_prover9(input_file) from validator.py runs the upper generated file (santa_logic.p9)**
    - Prover9 takes the input file (santa_logic.p9), which encodes:
        - Logical relationships between grid cells (e.g., adjacency rules).
        - Definitions of safety based on the absence of obstacles or the Grinch.
        - Clues associated with cells to infer the safest path.
    - Prover9 evaluates possible moves based on:
        - Adjacency to the current position.
        - Clue indicators in nearby cells.
        - Logical constraints to avoid unsafe moves.

3. **Output from Prover9**
    - Prover9 produces an output file (santa_logic.out) containing:
        - **Next Best Move:** The safest and most logical move for Santa, based on the processed clues and grid state.
        - **Fallback Rule:** If no safe move exists, Prover9 selects a backup move to minimize risks (e.g., moving away from the Grinch or obstacles).

8.2 AI Navigation Workflow:

- **Step 1:** Prover9 validates potential moves based on grid logic.
- **Step 2:** Known clues are updated dynamically after each move.
- **Step 3:** Collision detection checks for obstacles or the Grinch

9. Appendix

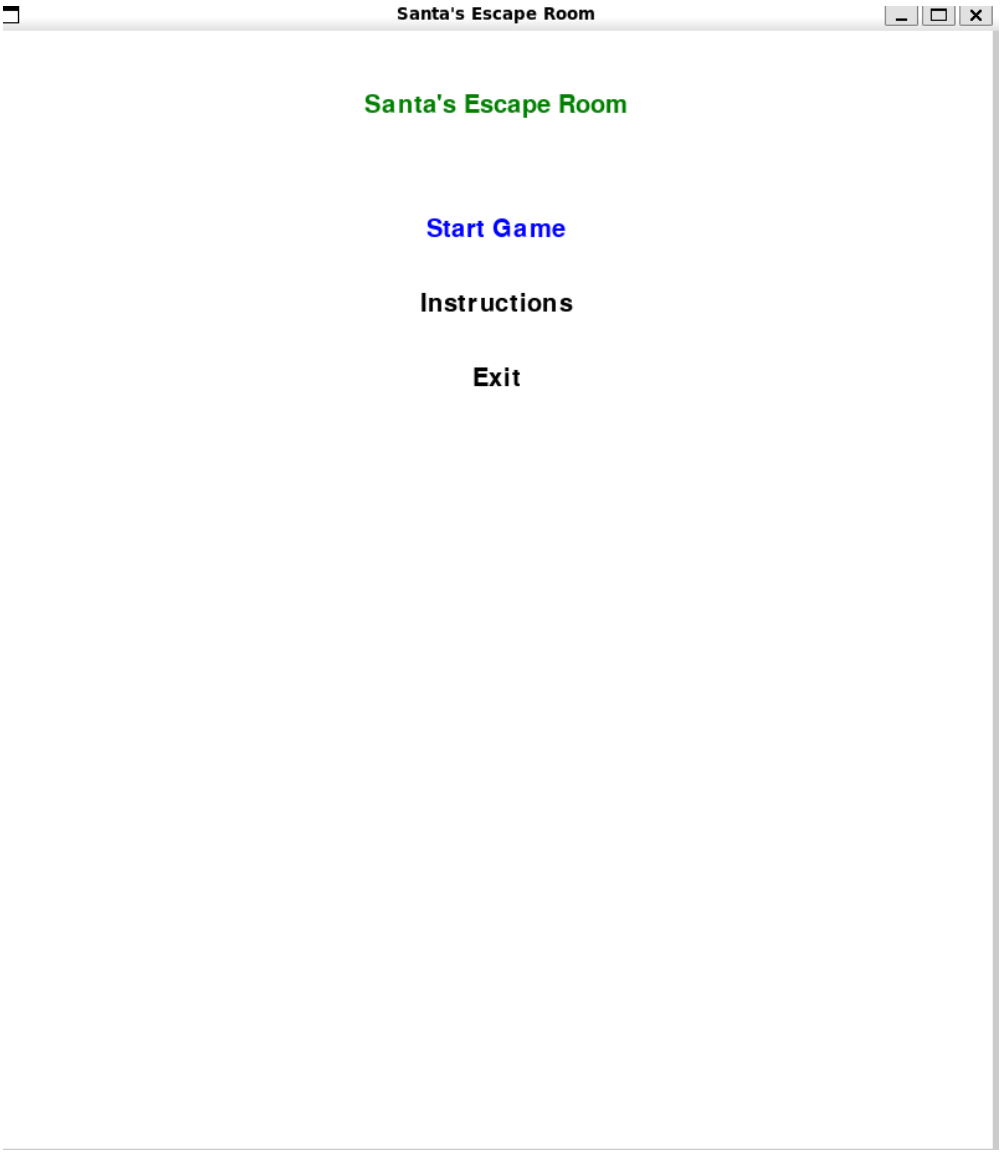
- **A.1 Full Code:**

Found in the link: LINK TO GIT REPOSITORY

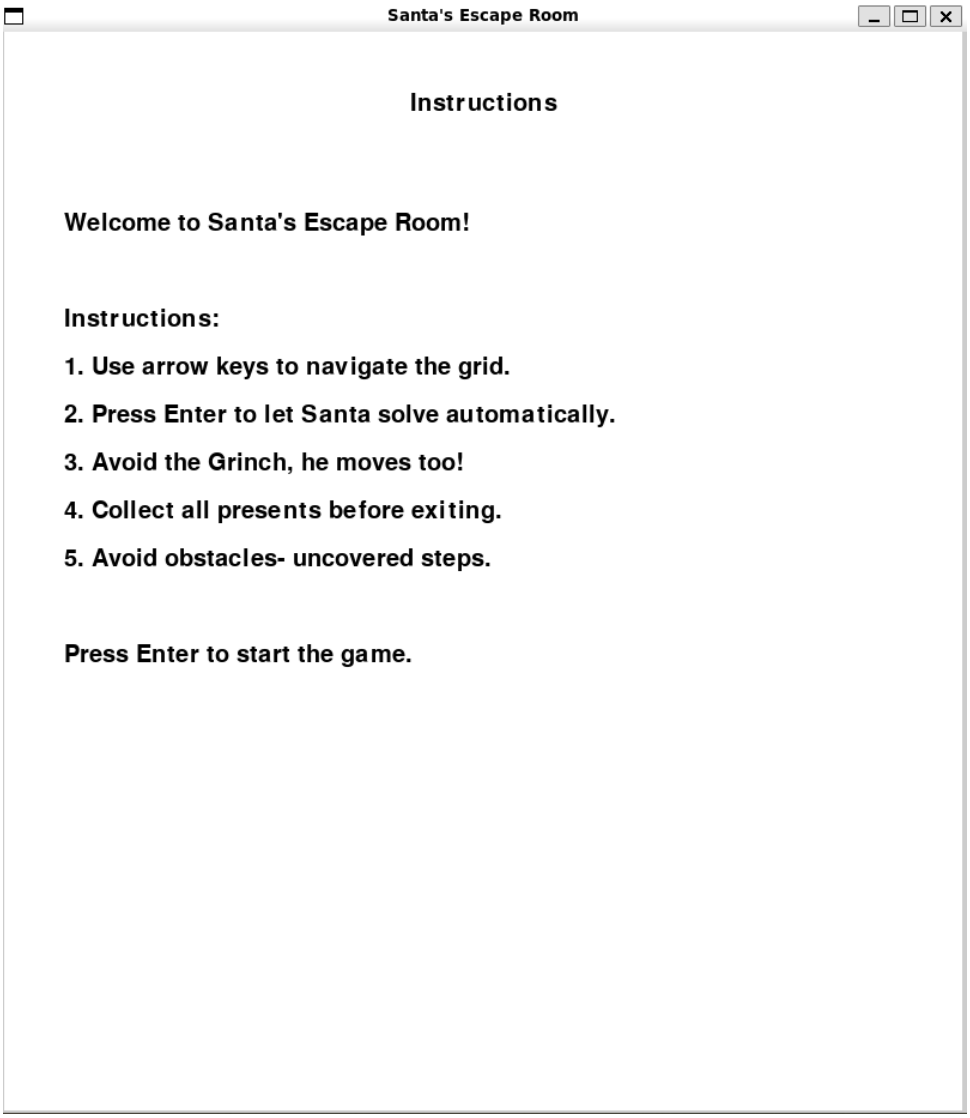
- **A.2 Figures:**

Screenshots of gameplay showcasing grid, Santa, and proximity clues.

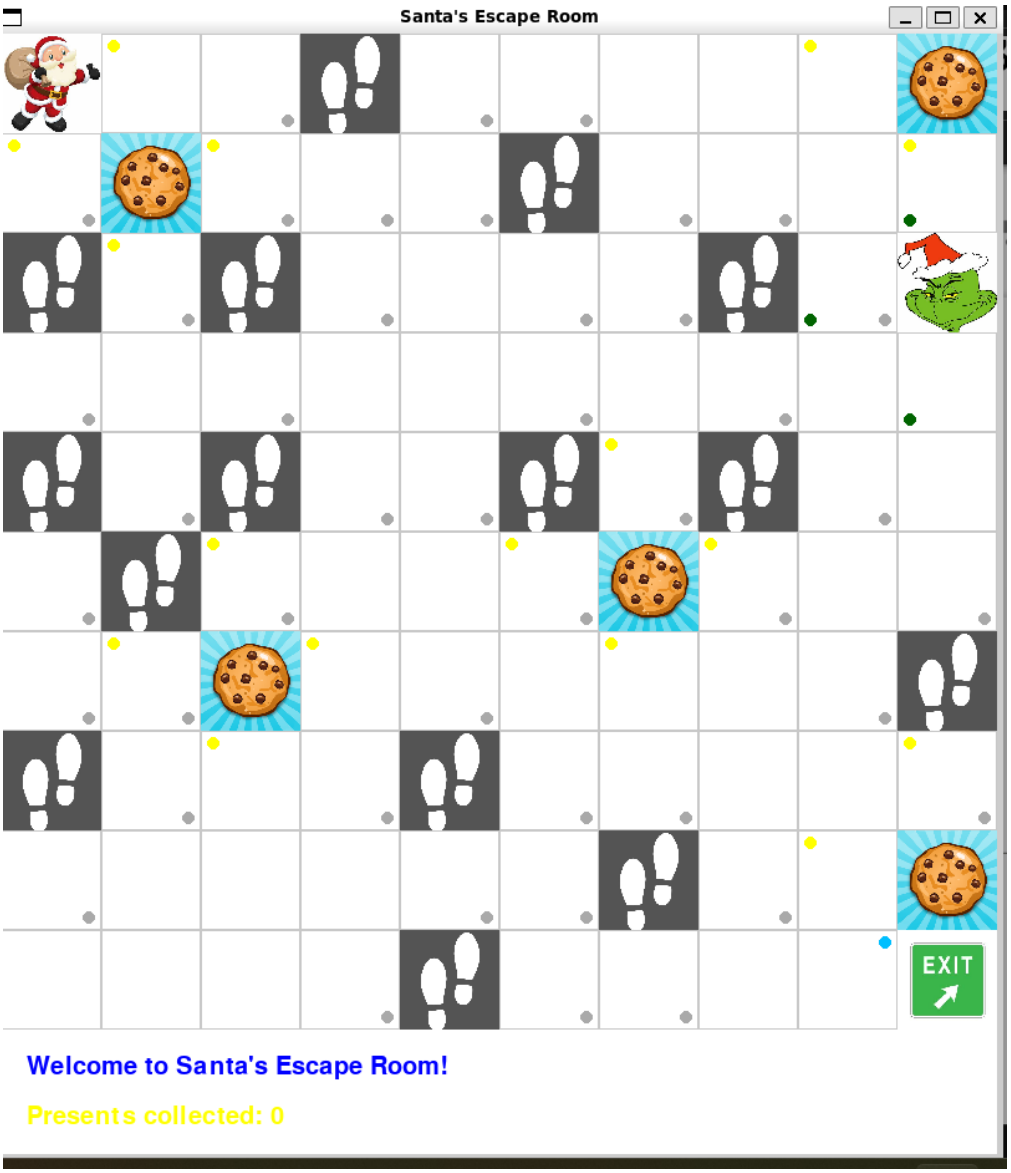
Menu:



Instructions:



Game(random generated, every time we enter the game it will look different):



10. Bibliography

1. Chatbots with Natural Language Understanding in Medical domain Roxana Ciorea and Adrian Groza Technical University of Cluj-Napoca, November 15, 2020- page 3
2. Prover9 and Mace4 – Link opened in 05.01.2025
3. First-Order Logic in Artificial Intelligence - GeeksforGeeks – Link opened in 05.01.2025
4. Pygame - Wikipedia – Link opened in 05.01.2025
5. What is a use case and how to write one | Wrike- Link opened in 05.01.2025