Odd Semester (2023)

BINUS UNIVERSITY

BINUS INTERNATIONAL

Algorithm and Programming

Final Project Report

Student Information:

Surname: Jonathan **Given Name:** Kevin Jonathan **Student ID:** 2702342823

Course Code: COMP6047001 **Course Name:** Algorithm and Programming

Class **:** L1BC **Lecturer** **:** Jude Joseph Lamug Martinez, MCS

Type of Assignment: Final Project Report

Submission Pattern

Due Date **:** 12 January 2024 **Submission Date** **:** 11 January 2024

The assignment should meet the below requirements.

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

Plagiarism/Cheating

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

Declaration of Originality

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

Signature of Student:

Kevin Jonathan

A. Background

The project guidelines encouraged me to step out of my comfort zone and explore beyond our class lessons. That's why I chose to tackle a maze chase game similar to Pac-Man, but with some tweaks to make it my own. I chose to make a game like Pac-Man since it will require me to figure out how the movement algorithms of enemies work and also the collisions.

B. Project Specification

1. Game Name

- Santa Dash

2. Game Concept

- "Santa Dash" is a maze chase game where users play as Santa where the goal is to clear a maze of presents and candies while being pursued by multiple enemies.
- The game consists of objects such as three different types of snowman, that have slightly different movement patterns and have their own unique image and abilities. If a regular snowman collides with Santa, it will make Santa lose a life. If an "Ice Spirit Snowman" collides with Santa, it will give Santa a debuff effect that will last for two seconds which will half Santa's current movement speed. The last type of snowman that can be found in this game is the "Thief Snowman", when Santa collides with this type of snowman, Santa's score will get deducted by ten every second they collide with one another.
- There are also presents and candies that can be found in the majority part of the maze or map. Presents will add up Santa's score by ten and candies are power ups that players could find which will reduce all of the enemies' movement speed by two and make Santa immune to the debuff effect.

3. Game Flow Summary

- The player has to collect all presents and candies that can be found in the maze while trying to avoid all three types of snowman to get the highest score possible. Players can take the candy powerups scattered around the maze to help them reach their goal more easily. If the player is successful in collecting all the presents and candies they will win. If they lose all their three lives before doing so, then they lose.

4. Game Objective

- To collect all the presents and candies that can be found in the maze while being pursued before losing their three lives. Even though the "Thief Snowman" won't make the player lose a life, if the player is not careful enough it can make their score all the way down to zero, so it's best to avoid it along with the other two snowmen.

5. Game Display

- The characters such as the enemy snowmans and Santa have a cartoonish style, while the items such as the presents and candies have a pixel art style. The background is just a plain white background and the maze is drawn using the pygame draw function.

6. Game Mechanics

- Players can move in four directions which are right, left, up and down, if the next tile or grid they are planning to move to is empty or having a present or having a candy.

7. Game Physics

- Santa or the players and the enemies or evil snowmens have collision detection between each other and elements of the maze including the walls, presents and candy. A player can only move to a certain direction if in the next grid or tile they are trying to move to in a certain direction that has no walls or obstacles. When a player collides with a present or candy, it will remove the candy or present they collide with and leave an empty space behind. When players collide with a regular snowman, it will deduct their life by one and resets every entity position including the player's back to the starting positions. When a player collides with an "Ice Spirit Snowman" it will give the player a debuff or slowness and if the player collides with the "Thief Snowman" it will deduct their score.

8. Game Input

- Up Arrow Key - Change Santa's movement direction to up
- Down Arrow Key - Change Santa's movement direction to down
- Left Arrow Key - Change Santa's movement direction to left
- Right Arrow Key - Change Santa's movement direction to right
- Spacebar - Restart the game once its over
- Left Mouse Button - To exit or close the python program

9. Game Output

- Player's Santa Image - can move, rotate, and change direction based on the player's inputs.
- Snowman Image - will spawn in the top left of the maze and will try to pursue the player and take turns that will be advantageous for getting closer to the player. (Snowman.png that can be located in the enemy_images folder)
- Ice Spirit Snowman Image - will spawn in the top right of the maze and will turn only when it collides with walls, otherwise it will continue moving in the same direction. (Icespirit.png that can be located in the enemy_images folder)
- Thief Snowman Image - will spawn bottom left and will turn up and down to pursue and, left and right only when it collides with a wall. (Thief.png that can be located in the enemy_images folder)
- White background using pygame
- Maze - drawn with pygame draw function
- Present Image (present.png)
- Candy Image (candy.png)
- Player's amount of lives left
- Player's score
- Player's highscore
- Victory (Victory.png) or Game over Image (Gameover.png)
- Background music (game_bgm.wav)
- Score sound effect (score.wav)

10. Game Libraries/Modules

- Pygame - Used to make most elements of the game and run it, from the level or maze, to draw the player's image and enemies' image. Also used to get player's input for the movement and it's used to also handle collisions and more.
- Time - to activate a timer for power ups, debuff, and give a brief delay every time a player spawns into the game before it's able to move.
- Math - Used to import PI and use it for calculations for drawing some elements of the maze
- Pygame mixer - Used to play background music and sound effects when player collects presents and candies
- Copy - I used the deep copy function to create an independent copy of a nested list of the map or level of the maze, so when changes are made to the maze, I still can call and use the original and unmodified one.

11. Game Files

- run.py - contains the main file to run the game
- constants.py - contains a bunch of constant variables that are used throughout the game
- player.py - contains the Player class in charge of the player's object
- enemy.py - contains the Enemy class in charge of the enemies' objects
- level.py - contains the level or maze in the form of nested lists and also contains the function to convert the nested lists into the actual maze
- scoreboard.py - contain the function to display the score and highscore text. Also displays the amount of lives left remaining and the function displays the victory and game over images.
- map_assets - folder containing the present and candy images
- game_sounds - folder containing the score sound effect and background music
- character_images - folder containing images of the player
- enemy_images - folder containing images of the enemies
- Gameover.png - game over image when player lose all their lives
- Victory.ong - victory image when player successfully collect all the presents and candies
- icon.png - the icon for the game window

12. Game Images

- Most of the images were not made by me. Some of them are altered versions of the original image.
- Character Images

  (https://pzuh.itch.io/santa-claus-free-sprites?download)



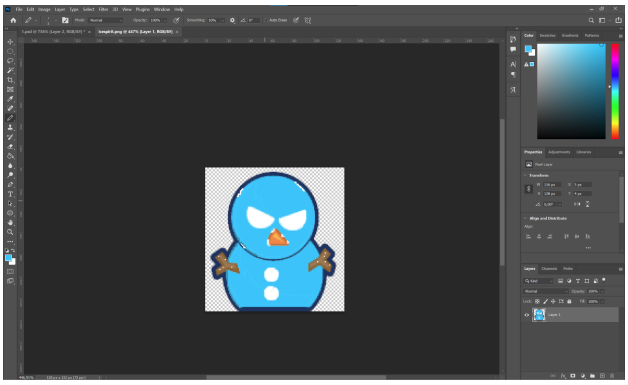

- Present and Candy Image

  (https://www.ascensiongamedev.com/topic/2486-free-christmas-sprites/)

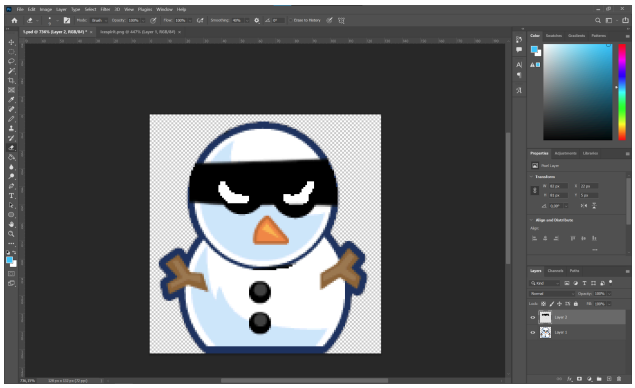
- Snowman Image

  (https://www.gameartguppy.com/shop/mini-monster-5-snowman/)

- Ice Spirit Snowman Image



- Thief Snowman Image



- Victory Image

- Game over Image

- Icon Image

13. Game Sounds

- Score sound effect

  (https://youtu.be/PGpbE9AfEd4?si=dnGJBG8RKmsTE27n)

- Background music

  (https://youtu.be/OWFVLhuhQ_E?si=lLJl6KQFefankHA4)

14. Game Fonts

- Freesansbold - for the score and highscore text

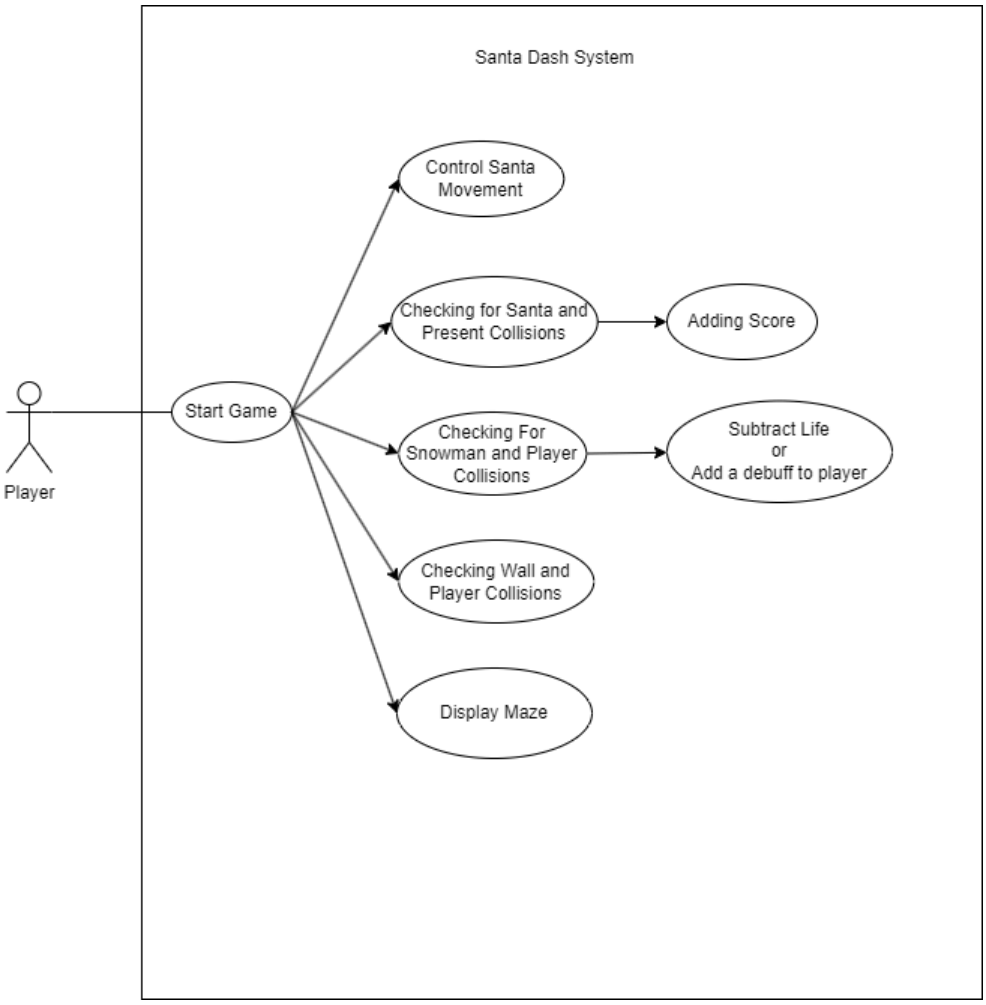
  (https://fonts2u.com/free-sans-bold.font)

- Poppins - for the game over and victory text in the image

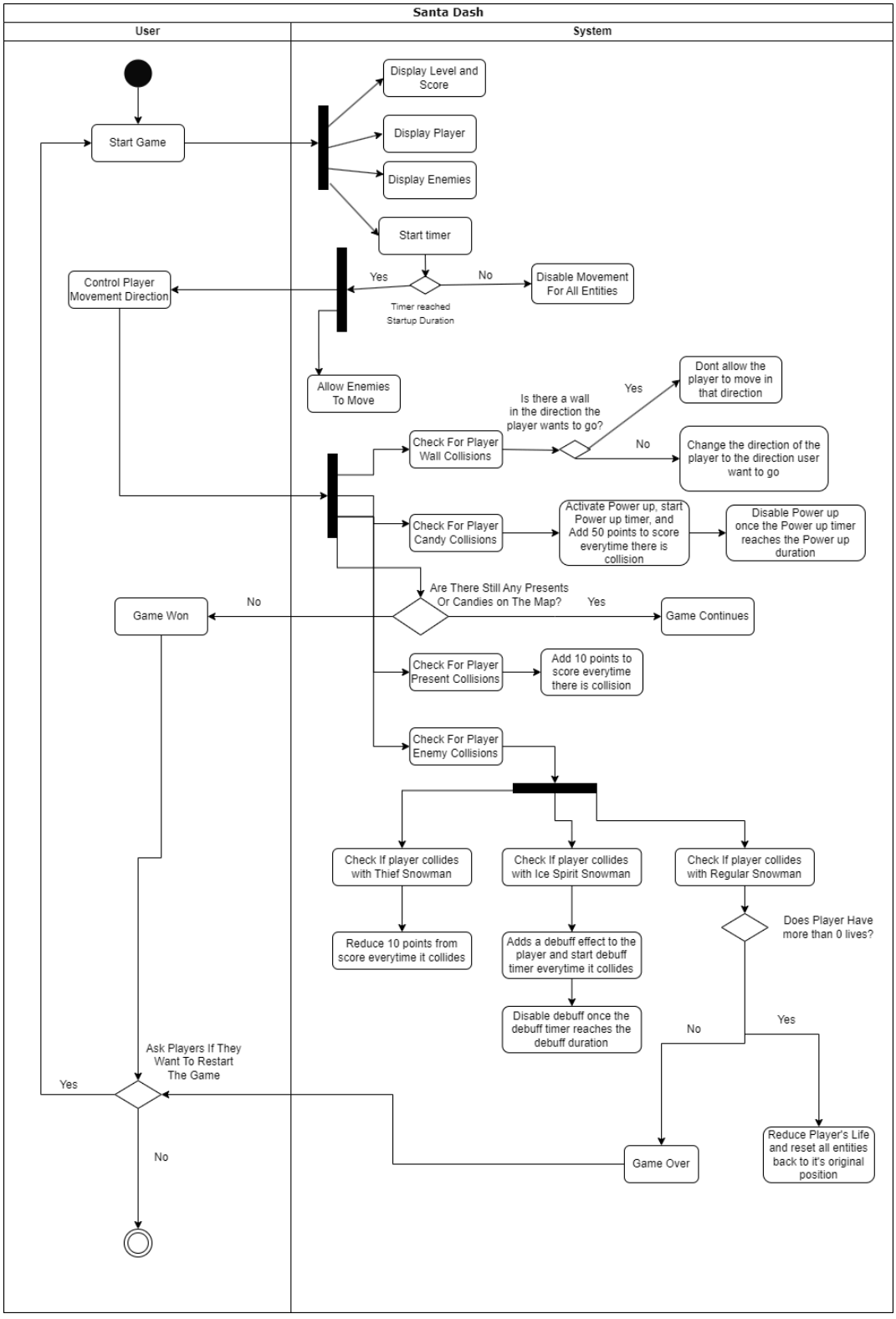
  (https://fonts.google.com/specimen/Poppins)

C. Solution Design

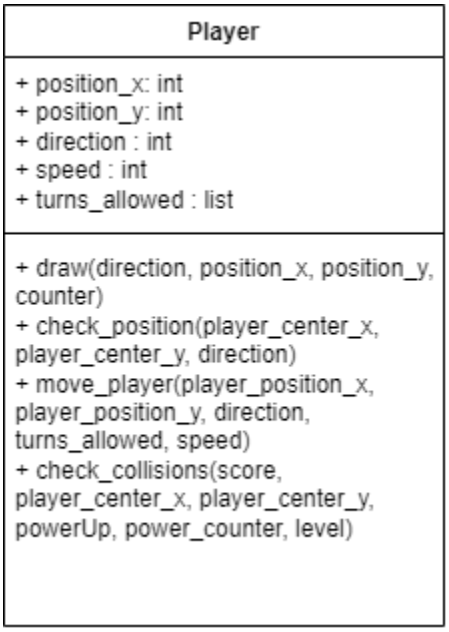
1. Use Case Diagram

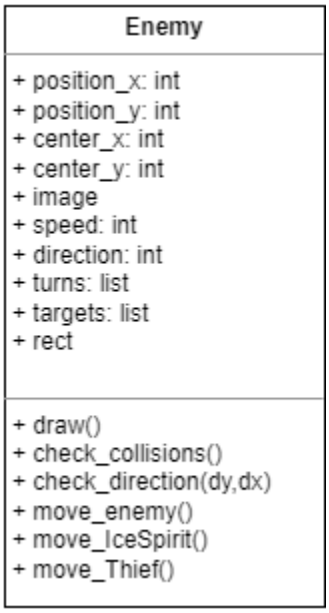


2. Activity Diagram



3. Class Diagram





D. Essential Algorithms

1. Drawing the maze

```
# Constants for the map
EMPTY = 0
PRESENT = 1
CANDY = 2
VERTICAL_LINE = 3
HORIZONTAL_LINE = 4
TOP_RIGHT = 5
TOP_LEFT = 6
BOTTOM_LEFT = 7
BOTTOM_RIGHT = 8
```

First of all, constants are defined to represent different parts or elements of the maze. Each constant holds a numeric value to signify a specific element (e.g., empty space, present, candy, various shapes like lines or arcs for the corners).

```
boards = [
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,6,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,5,0],
[0,3,6,4,4,4,4,4,4,4,4,4,4,4,5,6,4,4,4,4,4,4,4,4,4,4,4,5,3,0],
[0,3,3,2,1,1,1,1,1,1,1,1,1,3,3,1,1,1,1,1,1,1,1,1,1,2,3,3,0],
[0,3,3,1,6,5,1,6,5,1,6,4,5,1,3,3,1,6,4,5,1,6,5,1,6,5,1,3,3,0],
[0,3,3,1,3,3,1,3,3,1,3,0,3,1,3,3,1,3,0,3,1,3,3,1,3,3,1,3,3,0],
[0,3,3,1,3,3,1,3,3,1,3,0,3,1,3,3,1,3,0,3,1,3,3,1,3,3,1,3,3,0],
[0,3,3,1,7,8,1,7,8,1,7,4,8,1,7,8,1,7,4,8,1,7,8,1,7,8,1,3,3,0],
[0,3,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,3,0],
[0,3,3,1,6,4,4,5,1,6,5,1,6,4,4,4,4,5,1,6,5,1,6,4,4,5,1,3,3,0],
[0,3,3,1,7,4,4,8,1,3,3,1,7,4,5,6,4,8,1,3,3,1,7,4,4,8,1,3,3,0],
[0,3,3,1,1,1,1,1,1,3,3,1,1,1,3,3,1,1,1,3,3,1,1,1,1,1,3,3,0],
[0,3,3,1,6,4,4,4,4,8,7,4,5,1,3,3,1,6,4,8,7,4,4,4,4,5,1,3,3,0],
[0,3,3,1,7,4,4,4,4,5,6,4,8,1,7,8,1,7,4,5,6,4,4,4,4,8,1,3,3,0],
[0,3,3,1,1,1,1,1,1,3,3,1,1,1,1,1,1,1,1,3,3,1,1,1,1,1,1,3,3,0],
[4,8,7,4,4,4,4,5,1,3,3,1,6,4,4,4,4,5,1,3,3,3,1,6,4,4,4,4,8,7,4],
[0,0,0,0,0,0,0,3,1,7,8,1,7,4,4,4,4,8,1,7,8,1,3,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,0,0,0,0,0,0,0,0],
[4,4,4,4,4,4,4,8,1,6,5,1,6,4,4,4,4,5,1,6,5,1,7,4,4,4,4,4,4,4,4],
[0,0,0,0,0,0,0,0,1,3,3,1,7,4,4,4,4,8,1,3,3,1,0,0,0,0,0,0,0,0,0],
[4,4,4,4,4,4,4,5,1,3,3,1,1,1,1,1,1,1,1,3,3,1,6,4,4,4,4,4,4,4,4],
[0,0,0,0,0,0,0,3,1,3,7,4,5,1,6,5,1,6,4,8,3,1,3,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,3,1,3,6,4,8,1,7,8,1,7,4,5,3,1,3,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,3,1,3,3,1,1,1,1,1,1,1,1,3,3,1,3,0,0,0,0,0,0,0,0],
[4,5,6,4,4,4,4,8,1,7,8,1,6,4,4,4,4,5,1,7,8,1,7,4,4,4,4,5,6,4],
[0,3,3,1,1,1,1,1,1,1,1,1,3,0,0,0,0,3,1,1,1,1,1,1,1,1,3,3,0],
[0,3,3,1,6,4,4,4,4,4,5,1,3,0,0,0,0,3,1,6,4,4,4,4,4,5,1,3,3,0],
[0,3,3,1,3,0,0,0,0,0,3,1,3,0,0,0,0,3,1,3,0,0,0,0,0,3,1,3,3,0],
[0,3,3,1,7,4,4,4,4,4,8,1,7,4,4,4,4,8,1,7,4,4,4,4,4,8,1,3,3,0],
[0,3,3,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,3,0],
[0,3,7,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,8,3,0],
[0,7,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,8,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
]

level = copy.deepcopy(boards)
```

Then, a 2D array called “boards” is defined to represent the overall layout of the map or grid of the maze. Each number in this array corresponds to a tile type based on the defined constants. The 2D array is then copied using the deep copy function. This ensures

that the new level array is a separate copy from boards, meaning modifications made to level won't affect boards and vice versa.

```
# Loads present image
present_image = pygame.transform.scale(pygame.image.load('map_assets/present.png'), (15,15))
# Loads candy image
candy_image = pygame.transform.scale(pygame.image.load('map_assets/candy.png'), (25,25))
```

Next I load the images named 'present.png' and 'candy.png' from the 'map_assets' folder, then resizes them to specific dimensions using Pygame's pygame.transform.scale() function: the 'present.png' image is adjusted to 15x15 pixels, while the 'candy.png' image is resized to 25x25 pixels. These resized images are stored in present_image and candy_image variables, this is used in rendering these elements within the game.

```
def draw_level_elements(level):
    # Map each tile type to its respective drawing function
    shapes = {
        PRESENT: lambda i, j: screen.blit(present_image, (j * column_spacing + (0.3 * column_spacing), i * row_spacing + (0.3 * row_spacing))),
        CANDY: lambda i, j: screen.blit(candy_image, (j * column_spacing + (0.3 * column_spacing), i * row_spacing + (0.3 * row_spacing))),
        VERTICAL_LINE: lambda i, j: pygame.draw.line(screen, 'red', (j * column_spacing + (0.5 * column_spacing), i * row_spacing), (j * column_spacing + (0.5 * column_spacing), i * row_spacing + row_spacing), 3),
        HORIZONTAL_LINE: lambda i, j: pygame.draw.line(screen, 'red', (j * column_spacing, i * row_spacing + (0.5 * row_spacing)), (j * column_spacing + column_spacing, i * row_spacing + (0.5 * row_spacing)), 3),
        TOP_RIGHT: lambda i, j: pygame.draw.arc(screen, 'red', [(j * column_spacing - (column_spacing * 0.4)) - 2, (i * row_spacing + (0.5 * row_spacing)), column_spacing, row_spacing], 0, PI / 2, 3),
        TOP_LEFT: lambda i, j: pygame.draw.arc(screen, 'red', [(j * column_spacing + (column_spacing * 0.5)), (i * row_spacing + (0.5 * row_spacing)), column_spacing, row_spacing], PI / 2, PI, 3),
        BOTTOM_LEFT: lambda i, j: pygame.draw.arc(screen, 'red', [(j * column_spacing + (column_spacing * 0.5)), (i * row_spacing - (0.4 * row_spacing)), column_spacing, row_spacing], PI, 3 * PI / 2, 3),
        BOTTOM_RIGHT: lambda i, j: pygame.draw.arc(screen, 'red', [(j * column_spacing - (column_spacing * 0.4)) - 2, (i * row_spacing - (0.4 * row_spacing)), column_spacing, row_spacing], 3 * PI / 2, 2 * PI, 3),
    }

    # Iterate through the level and draw the corresponding shapes
    for i, row in enumerate(level):
        for j, tile in enumerate(row):
            # Check if the tile type has a corresponding drawing function
            if tile in shapes:
                # Execute the drawing function for the specific tile
                shapes[tile](i, j)
```

The “draw_level_elements” function is used to convert the numeric values from the 2D array representing the game map into visual elements on the Pygame window. The ” shapes” dictionary maps each tile type (constant) to a specific drawing function. For instance, if the number 1 represents a present, the function maps it to the PRESENT constant, which then triggers the drawing of the present image at the corresponding grid position. The “draw_level_elements” function also has a nested for loop which will iterate through every row and column of the level containing the 2D array. For each tile in the grid, it checks if the tile type matches any of the defined shapes in the shapes dictionary and executes the corresponding drawing function to visualize that tile on the screen. Based on the defined shapes (like arcs, lines, and images), the function draws these elements on the screen at specific positions calculated based on the grid coordinates.

2. Player Movement and Player-Wall Collisions

```
for event in pygame.event.get():
    # To exit out of the game
    if event.type == pygame.QUIT:
        running = False

    # Check for arrow inputs for santa movement
    if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_RIGHT:
            direction_command = RIGHT
        if event.key == pygame.K_LEFT:
            direction_command = LEFT
        if event.key == pygame.K_UP:
            direction_command = UP
        if event.key == pygame.K_DOWN:
            direction_command = DOWN
```

In the run.py file, there is a for loop to capture events using Pygame and it takes the keyboard inputs of the arrow keys (up, down, left, right) as user input to alter the player's movement direction. Initially, the input is stored in the direction_command variable, allowing for validation of whether the intended direction change is permissible before the player's movement is adjusted.

```
def check_position(self, player_center_x, player_center_y, direction):
    # Initialize a list to track possible turns in each direction
    turns = [False, False, False, False]
    # Calculate the index of the player's current column and row
    column_index = player_center_x // column_spacing
    row_index = player_center_y // row_spacing
```

In the player.py file, there is a function called “check_position”, this function is responsible for checking the available directions a player can move based on their current position and the game level layout. It first initializes a list called “turns” containing boolean values for each direction the player can possibly turn. Each index in the turns list represents a possible movement direction for the player (0 = RIGHT, 1 = LEFT, 2 = UP, 3 = DOWN). It will then calculate the column and row indices of the player's current position in the game level based on the provided player_center_x and player_center_y values.

```
# Check for possible movements based on direction and wall collision detection
if player_center_x // 30 < 29:
    if direction == RIGHT:
        # Check for collision when moving right
        if level[row_index][(player_center_x - fudge_factor) // column_spacing] < 3:
            turns[LEFT] = True
    elif direction == LEFT:
        # Check for collision when moving left
        if level[row_index][(player_center_x + fudge_factor) // column_spacing] < 3:
            turns[RIGHT] = True
    elif direction == UP:
        # Check for collision when moving up
        if level[(player_center_y + fudge_factor) // row_spacing][column_index] < 3:
            turns[DOWN] = True
    elif direction == DOWN:
        # Check for collision when moving down
        if level[(player_center_y - fudge_factor) // row_spacing][column_index] < 3:
            turns[UP] = True
```

The function first assesses the player's central x-coordinate position within the maze boundaries. Afterward, considering the specified direction and collision detection logic, it determines whether the player can proceed in that direction without encountering an obstacle, denoted by values lower than 3 within the level array which are the EMPTY, PRESENT, and CANDY constants. The wall collision detection logic operates by assessing adjacent grids relative to the player's position. It accomplishes this by applying a slight positional adjustment, known as the fudge factor, either adding or subtracting it from the player's central x or y coordinate position, depending on the movement direction. This assessment ensures accurate collision detection with the walls of the maze. Upon identifying a feasible turn in a particular direction, the function updates the turns value accordingly.

```
if direction in (UP, DOWN):
    # Making sure the player is roughly in the middle of the grid or tile
    if 12 <= player_center_x % column_spacing <= 18:
        # Additional collision checks for UP and DOWN movements
        if level[(player_center_y + fudge_factor) // row_spacing][column_index] < 3:
            turns[DOWN] = True
        if level[(player_center_y - fudge_factor) // row_spacing][column_index] < 3:
            turns[UP] = True
    if 12 <= player_center_y % row_spacing <= 18:
        if level[row_index][(player_center_x - column_spacing) // column_spacing] < 3:
            turns[LEFT] = True
        if level[row_index][(player_center_x + column_spacing) // column_spacing] < 3:
            turns[RIGHT] = True

if direction in (RIGHT, LEFT):
      # Making sure the player is roughly in the middle of the grid or tile
    if 12 <= player_center_x % column_spacing <= 18:
         # Additional collision checks for RIGHT and LEFT movements
        if level[(player_center_y + row_spacing) // row_spacing][column_index] < 3:
            turns[DOWN] = True
        if level[(player_center_y - row_spacing) // row_spacing][column_index] < 3:
            turns[UP] = True
    if 12 <= player_center_y % row_spacing <= 18:
        if level[row_index][(player_center_x - fudge_factor) // column_spacing] < 3:
            turns[LEFT] = True
        if level[row_index][(player_center_x + fudge_factor) // column_spacing] < 3:
            turns[RIGHT] = True
:
```

It will then perform specific collision checks for movements in four directions (UP, DOWN, LEFT, RIGHT) within the game's grid layout. The program will ensure that the player is near the center of the grid before examining adjacent spaces for potential movement. It will use the same collision logic as the code above and update the turns value accordingly.

```
    else:
        turns[RIGHT] = True
        turns[LEFT] = True

    return turns
```

If the player is out of the maze boundaries or near it, this else statement ensures that when the player is near the grid's boundary, they're still granted the ability to move in both right and left directions without restriction. Lastly, the function will return the turns list with all the updated values.

```
# Check allowed turns for the player based on its current position and direction
turns_allowed = player.check_position(player_center_x, player_center_y, player.direction)
```

The turns list is stored in a variable called turns_allowed in the run.py file.

```
    # Sets the player direction if its allowed
    if event.type == pygame.KEYUP:
        if event.key == pygame.K_RIGHT and direction_command == RIGHT:
            direction_command = player.direction
        if event.key == pygame.K_LEFT and direction_command == LEFT:
            direction_command = player.direction
        if event.key == pygame.K_UP and direction_command == UP:
            direction_command = player.direction
        if event.key == pygame.K_DOWN and direction_command == DOWN:
            direction_command = player.direction

# Changes direction of player if its allowed
if direction_command == RIGHT and turns_allowed[RIGHT]:
        player.direction = RIGHT
if direction_command == LEFT and turns_allowed[LEFT]:
        player.direction = LEFT
if direction_command == UP and turns_allowed[UP]:
        player.direction = UP
if direction_command == DOWN and turns_allowed[DOWN]:
        player.direction = DOWN
```

The snippet code above manages player direction changes based on released key inputs. It verifies if the released key aligns with the player's current direction, updating the direction command accordingly. Then, it checks if the intended direction change is allowed by the game's logic stored in turns_allowed. If permitted, it updates the player's direction, ensuring controlled movement aligned with game rules.

```
def move_player(self, player_position_x, player_position_y, direction, turns_allowed, speed):
        # Move the player according to the allowed directions and speed
        if direction == RIGHT and turns_allowed[RIGHT]:
            player_position_x += speed
        elif direction == LEFT and turns_allowed[LEFT]:
            player_position_x -= speed
        if direction == UP and turns_allowed[UP]:
            player_position_y -= speed
        elif direction == DOWN and turns_allowed[DOWN]:
            player_position_y += speed
        # Return the updated player position
        return player_position_x, player_position_y
```

Back to the player.py file, there is a function or method called “move_player”. The “move_player” function is in charge of updating the player’s x and y position based on the provided direction, permissible movements specified in turns_allowed, and a defined speed. It evaluates the direction input and, if the direction they want to turn to is allowed, it modifies the respective position—incrementing or decrementing player_position_x or player_position_y according to the specified speed. Lastly, the function returns the updated player_position_x and player_position_y, reflecting the movement made within the constraints of allowed directions and speed.

```
# Check if player_center_x is beyond the right boundary
if player_center_x > RIGHT_BOUNDARY:
    # Reset player_center_x and update player position for continuous movement
    player_center_x = RESET_RIGHT
    player.position_x = player_center_x + MOVEMENT_OFFSET

# Check if player_center_x is beyond the left boundary
elif player_center_x < LEFT_BOUNDARY:
    # Reset player_center_x and update player position for continuous movement
    player_center_x = RESET_LEFT
    player.position_x = player_center_x - MOVEMENT_OFFSET
```

There is also a code snippet in the run.py file to handle if the player is moving beyond the right or left boundary, it will reset their position to the designated right or left reset point and adjust their position for ongoing movement.

3. Animating and Drawing Player

```
# Makes an empty list to store images of player
player_images = []
# Iterates over all the images in the folder and put it in the list
for i in range(1, 5):
        player_images.append(pygame.transform.scale(pygame.image.load(f'character_images/{i}.png'), (50, 50)))
```

The initial code snippet in the constants.py file, loads images of the player character and scales them to a 50x50 pixel size, storing them in a list named player_images. This list contains multiple images representing different frames of animation for the player character.

```
# Main loop for image animation
if counter < MAX_COUNTER:
    counter += 1
else:
    # Reset counter
    counter = 0
```

The code manages animation by utilizing a counter variable. This counter is incremented within the main loop and, when it reaches a maximum value (MAX_COUNTER), it resets to zero. This allows controlling the pace of the animation.

```
def draw(self, direction, position_x, position_y, counter):
    # Depending on the 'direction' variable, different transformations of the player image are performed and drawn on the screen

    # If the direction is RIGHT, draw the player image at the given position
    if direction == RIGHT:
        screen.blit(player_images[counter // 5], (position_x, position_y))

    # If the direction is LEFT, flip the player image horizontally and draw it at the given position
    elif direction == LEFT:
        flipped_image = pygame.transform.flip(player_images[counter // 5], True, False)
        screen.blit(flipped_image, (position_x, position_y))

    # If the direction is UP, rotate the player image 90 degrees clockwise and draw it at the given position
    elif direction == UP:
        rotated_image = pygame.transform.rotate(player_images[counter // 5], 90)
        screen.blit(rotated_image, (position_x, position_y))

    # If the direction is DOWN, rotate the player image 270 degrees clockwise (or 90 degrees counter-clockwise) and draw it at the given position
    elif direction == DOWN:
        rotated_image = pygame.transform.rotate(player_images[counter // 5], 270)
        screen.blit(rotated_image, (position_x, position_y))
```

In the player.py file there is the “draw” function. The "draw" function determines the appearance of the player character based on the provided direction and the current animation frame indicated by the 'counter'. It selects the corresponding image from the 'player_images' list and applies transformations such as flipping or rotation, then draws this modified image onto the screen at the specified position coordinates (position_x, position_y). With a max counter set to 20 and 4 images for animation, each image is displayed for 5 iterations of the main loop. The counter increments on each iteration, and every 5 increments (counter reaches multiples of 5), it switches to the next image in the sequence. This arrangement ensures that each image in the animation sequence is displayed consecutively for a span of 5 iterations before transitioning to the next image in the cycle, resulting in a smooth and controlled animation sequence.

4. Player-Item Collision Handling

```
def check_collisions(self, score, player_center_x, player_center_y, powerUp, power_counter, level):
    # Define spacing based on screen dimensions
    row_spacing = (screen_height - 50) // 32
    column_spacing = screen_width // 30

    # Check if the player is within the horizontal bounds of the game area
    if 0 < player_center_x < 870:
        # Check if the player's current position corresponds to a present item
        if level[player_center_y // row_spacing][player_center_x // column_spacing] == PRESENT:
            # If a present is found at the player's position, remove it and increase the score by 10
            level[player_center_y // row_spacing][player_center_x // column_spacing] =  EMPTY
            score += 10
            score_sound.play()

        # Check if the player's current position corresponds to a candy item
        if level[player_center_y // row_spacing][player_center_x // column_spacing] == CANDY:
            # If a candy is found at the player's position, remove it and increase the score by 50
            level[player_center_y // row_spacing][player_center_x // column_spacing] = EMPTY
            score += 50
            score_sound.play()
            # For Powerup
            powerUp = True
            power_counter = 0

    # Return the updated score after checking collisions
    return score, powerUp, power_counter
```

In the player.py file, there is a function or method called “check_collisions”. This function is used to handle collision detection for Santa or the player with specific items

(like presents and candies) represented in a 2D grid-level array. It first verifies if the player is within the horizontal bounds of the maze. It then checks the player's current grid cell where the player is located. If the player is on a cell containing a present (PRESENT), it removes the present, increments the score by 10, and plays a sound effect. If the player is on a cell containing a candy (CANDY), it removes the candy, increments the score by 50, plays a sound effect, and activates a power-up by setting powerUp to True and resetting power_counter to 0. Finally, it returns the updated score, powerUp state, and power_counter after checking collisions.

5. Player Collision Detection with Enemies

```
player_circle = pygame.draw.circle(screen, 'white', (player_center_x, player_center_y), 10, 2)
# Draw the player on the screen based on its current direction and position
player.draw(player.direction, player.position_x, player.position_y, counter)
```

In the run.py file, in the main loop for the game, a circle is drawn behind the player's image to act as a "hitbox" for Santa's player collision with the enemies.

```
# Santa dies if collides with snowman
if player_circle.colliderect(evilSnowman.rect):
    if lives > 0:
        lives -= LIVES_PENALTY
        startup_counter = 0
        # Reset positions and directions
        player.position_x = START_PLAYER_X
        player.position_y = START_PLAYER_Y
        player.direction = RIGHT
        direction_command = RIGHT
        evilSnowman_position_x = START_EVIL_SNOWMAN_X
        evilSnowman_position_y = START_EVIL_SNOWMAN_Y
        evilSnowman_direction = RIGHT
        iceSpirit_position_x = START_ICE_SPIRIT_X
        iceSpirit_position_y = START_ICE_SPIRIT_Y
        iceSpirit_direction = RIGHT
        thief_position_x = START_THIEF_X
        thief_position_y = START_THIEF_Y
        thief_direction = RIGHT
        powerUp = False
        debuff = False
    else:
        game_over = True
        moving = False
        startup_counter = 0
```

If the player or Santa's circle collides with the regular evil snowman's rect, it checks if the player has remaining lives. If the player has lives left, the program will decrease the

number of lives by a certain penalty value which is one and it will reset various positions (player, evil snowman, ice spirit, thief), directions, and flags related to power-ups and debuffs. If the player has no lives left, it will set the game over flag to "True", stop the movement of game elements, and reset "startup_counter".

```
# If player collides with ice spirit, activate debuff for the player and start the timer
elif player_circle.colliderect(iceSpirit.rect):
    if powerUp ==  False:
        debuff = True
        debuff_timer = 0
```

If the player's circle collides with the "Ice Spirit Snowman" or the blue colored snowman's rect, it will activate a debuff for the player and start a timer for the debuff effect.

```
# If player collides with thief, reduce score
elif player_circle.colliderect(Thief.rect):
    if score > 0:
        score -= SCORE_PENALTY
```

If the player's collides with the "Thief Snowman" or the snowman with the mask and has a positive score, it will decrease the player's score by a penalty value which is ten.

6. Power up and Debuff

```
# Check if power-up is active and within duration
if powerUp and power_counter < MAX_POWER_DURATION:
    # Increment the power-up timer
    power_counter += 1
# If power-up duration is reached
elif powerUp and power_counter >= MAX_POWER_DURATION:
    # Reset the power-up timer
    power_counter = 0
    # Deactivate the power-up
    powerUp = False
```

If the power up is activated by a player colliding with a candy item, a counter or a timer will start which will last for about three seconds. If the power up counter which is the "power_counter" is equal to or goes over the maximum power up duration, then it will reset the "power_counter" to zero and set the value of "powerUp" to "False".

```
# Check if debuff is active and within duration
if debuff and debuff_timer < MAX_DEBUFF_DURATION:
    # Increment the debuff timer
    debuff_timer += 1
    # If debuff duration is reached
elif debuff and debuff_timer >= MAX_DEBUFF_DURATION:
    # Reset the debuff timer
    debuff_timer = 0
    # Deactivate the debuff
    debuff = False
```

The debuff timer logic is also the same with the power up timer logic.

```
if powerUp:
    # Half the enemies' speed
    modified_iceSpirit_speed = iceSpirit_speed // 2
    modified_evilSnowman_speed = evilSnowman_speed // 2
    modified_thief_speed = thief_speed // 2
else:
    # Set the enemies' speed back to the original value
    modified_iceSpirit_speed = iceSpirit_speed
    modified_evilSnowman_speed = evilSnowman_speed
    modified_thief_speed = thief_speed
```

When the power up is active and “powerUp” is set to “True” it modifies the speeds of various enemies (ice spirit, evil snowman, thief) by halving their original speeds. When the power up is no longer active, it sets back the enemies’ speed back to its original value.

```
if debuff:
    # Apply speed modification if debuff is active
    modified_speed = original_player_speed // 2
else:
    # Returns player's speed to the original speed
    modified_speed = original_player_speed
```

When the debuff is active it halves the player's speed (original_player_speed), creating a handicap by reducing the player's movement speed. When the debuff is not active, it restores the player's speed to its original value.

7. Enemy Movement

In the enemy.py file in the Enemy class there is a method or function called “move_enemy”. This method is for the regular evil snowman movement algorithm. The method checks the current direction of the enemy (RIGHT, LEFT, UP, or DOWN). Depending on the current direction, it checks various conditions to determine the next move of the enemy towards a target (self.targets) which contains the X and Y coordinates for the player’s position. The conditions involve checking whether the enemy can turn in a specific direction (it uses the same algorithm as the player’s wall collision) and comparing the target's position with the enemy's current position. If the enemy can turn in a certain direction, it updates its position accordingly (self.position_x or self.position_y). If the enemy cannot turn in the desired direction or the target is not in the expected position, it checks other available directions. The code also contains a section to handle cases where the enemy reaches the left or right boundary of the game world. In such cases, the enemy's position is adjusted to create a seamless movement.

The movement algorithm for the other two snowmen is similar to the regular snowman, but slightly modified. The “Ice Spirit Snowman” is going to turn whenever colliding with walls, otherwise it will continue moving straight in the direction it is currently going. The “Thief Snowman” turns up or down at any point to pursue, but left and right only on collision.

8. Winning The Game or Losing The Game

```
game_won = True
# For loop to check if there are still presents and candy in the map
for i in range(len(level)):
    if PRESENT in level[i] or CANDY in level[i]:
        game_won = False
```

In the main game loop in the run.py file, initially “game_won” is set to “True”. Then a for loop iterates through each row and column to check if either a present or candy item still exists in the maze or map. If there is still the present or candy item in the level, it will set “game_won” = “False”. If not, “game_won” will stay “True”.

```
# Santa dies if collides with snowman
if player_circle.colliderect(evilSnowman.rect):
    if lives > 0:
        lives -= LIVES_PENALTY
        startup_counter = 0
        # Reset positions and directions
        player.position_x = START_PLAYER_X
        player.position_y = START_PLAYER_Y
        player.direction = RIGHT
        direction_command = RIGHT
        evilSnowman_position_x = START_EVIL_SNOWMAN_X
        evilSnowman_position_y = START_EVIL_SNOWMAN_Y
        evilSnowman_direction = RIGHT
        iceSpirit_position_x = START_ICE_SPIRIT_X
        iceSpirit_position_y = START_ICE_SPIRIT_Y
        iceSpirit_direction = RIGHT
        thief_position_x = START_THIEF_X
        thief_position_y = START_THIEF_Y
        thief_direction = RIGHT
        powerUp = False
        debuff = False
    else:
        game_over = True
        moving = False
        startup_counter = 0
```

If Santa or the player dies when they have no more extra lives left, it will set the value of “game_over” to “True” and “moving” to “False”.

```
# Display the score, highscore, lives and game over or game won
draw_misc(score, lives, game_over, game_won, highscore)
```

When a new iteration of the main game loop occurs, this “draw_misc” function will take in the value of the “game_won” and “game_over”.

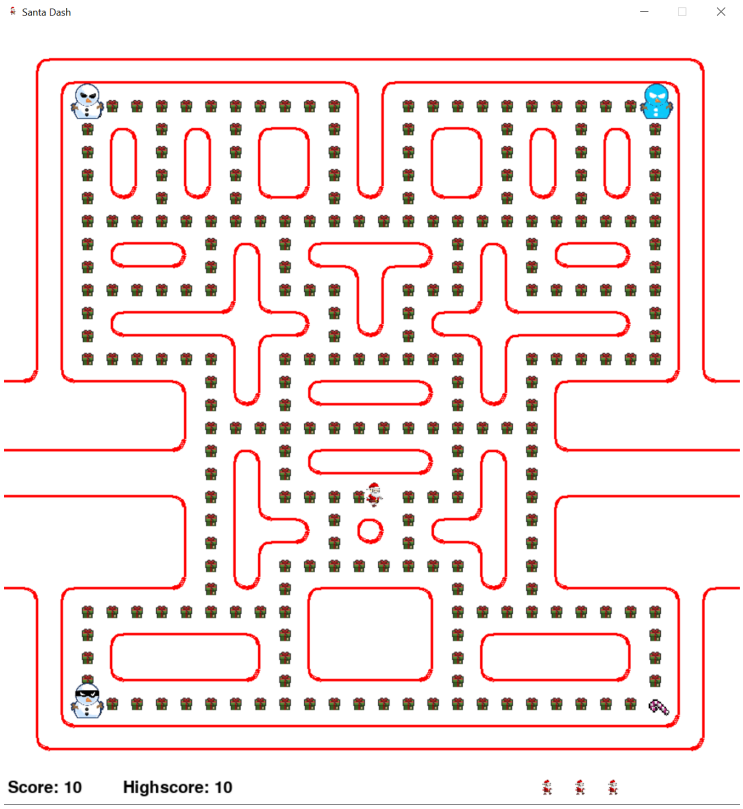
```python
# Function to draw various game elements on the screen
def draw_misc(score, lives, game_over, game_won, highscore):
    # Rendering text for score and highscore
    score_text = font.render(f'Score: {score}', True, 'black')
    highscore_text = font.render(f'Highscore: {highscore}', True, 'black')
    # Displaying score and highscore on the screen
    screen.blit(score_text, score_position)
    screen.blit(highscore_text, highscore_position)
    # Displaying player lives as small images on the screen
    for i in range(lives):
        screen.blit(pygame.transform.scale(player_images[0],PLAYER_LIVES_SIZE), (PLAYER_LIVES_START_POS_X + i * PLAYER_LIVES_GAP, PLAYER_LIVES_START_POS_Y))
    # Displaying game over screen if the game is over
    if game_over:
        screen.blit(game_over_image,(0,0))
    # Displaying victory screen if the game is won
    if game_won:
        screen.blit(victory_image, (0, 0))
```

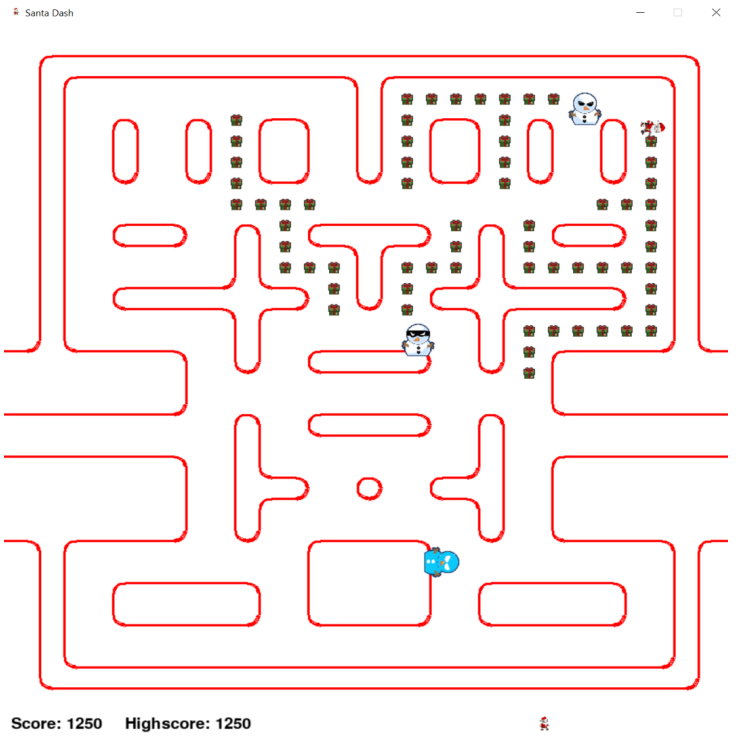
The “draw_misc” function will either display the game over image or game won image depending on which value is set to “True”.

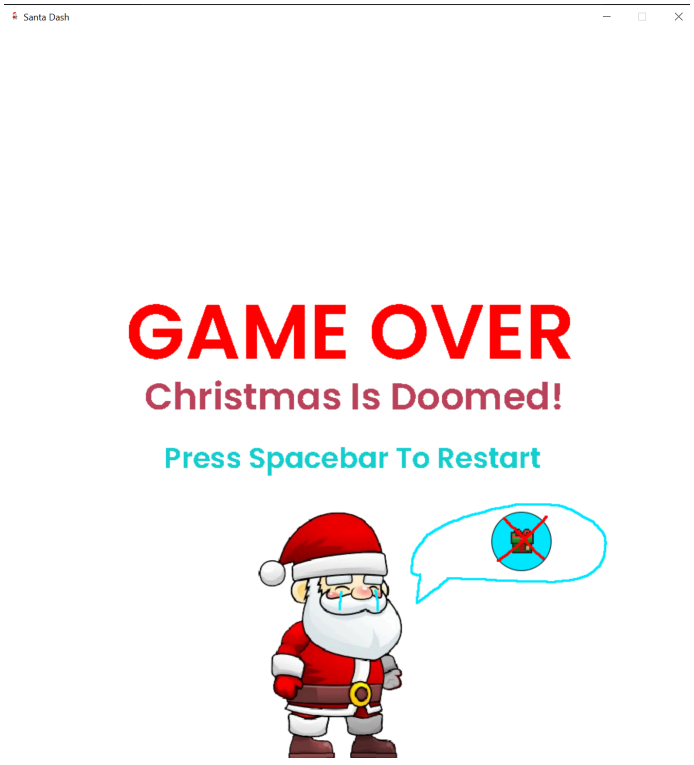
```python
victory_image = pygame.image.load('Victory.png')
victory_image = pygame.transform.scale(victory_image, (screen_width, screen_height))

game_over_image = pygame.image.load('Gameover.png')
game_over_image = pygame.transform.scale(game_over_image, (screen_width, screen_height))
```

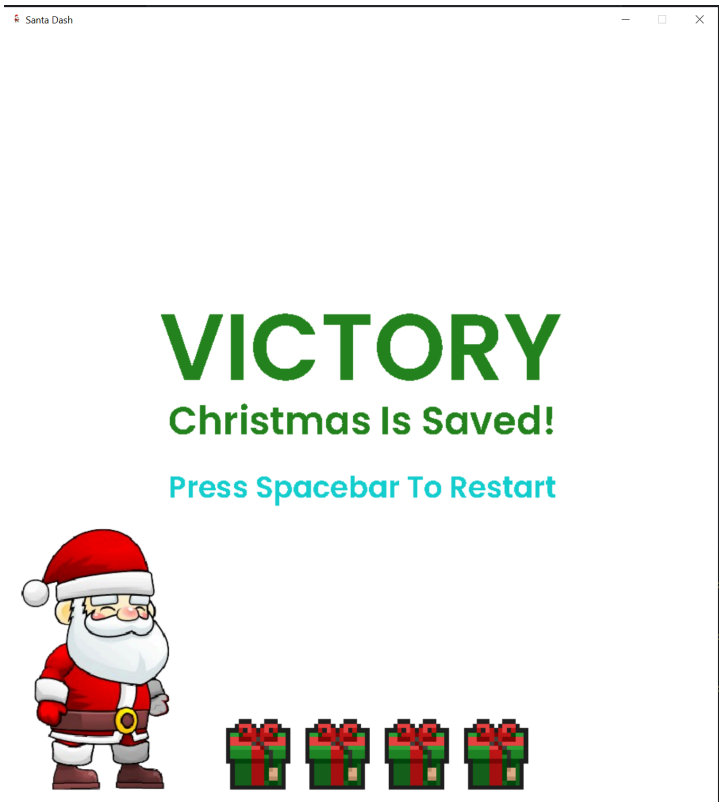
The victory image and game over image is loaded and scaled in the constants.py file.

E. Screenshots of the Game






Santa Dash
GAME OVER
Christmas Is Doomed!
Press Spacebar To Restart


Santa Dash
VICTORY
Christmas Is Saved!
Press Spacebar To Restart

F. Lessons Learned/ Reflection

Making this game was a fun and stressful experience. I started on this project quite late since I haven't really got a solid idea of what to make until there are a few weeks left until the presentation. This has taught me to start thinking of ideas for the final project from the very beginning it was introduced, so I have more time to work on it and stress about it less.

Since I have liked playing games since I was a child, when I made this game, I learned new things that weren't taught in class before, like how a character could be animated in pygame, how maps are actually made for a game, and how some AI movement and pathfinding algorithms work. Even though my game's enemy movement algorithm is just a bunch of If and Else statements and it's probably not the most efficient code, it's still cool to see how the enemies come to life and tries its best to pathfind to the player's position. I learned that for most games with enemies that can pathfind the player, they usually use either one of these four pathfinding algorithms: A*, Djikstra, Breadth First Search, or Depth First Search. I tried implementing the A* algorithm into my code, but I failed a couple of times and time was running out, so I just stuck with the If Else statements.