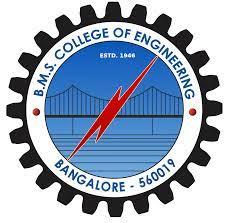
**B. M. S. COLLEGE OF ENGINEERING***(Autonomous Institute, Affiliated to VTU, Belagavi)*

**Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019**

**DEPARTMENT OF MACHINE LEARNING**

**Academic Year: 2023-2024 (Session: November 2023 - February 2024)**



**OBJECT ORIENTED PROGRAMMING**

**(23AM5PCOOP)**

**ALTERNATIVE ASSESSMENT TOOL (AAT)**

**SNAKE GAME USING SOLID PRINCIPLES**

**Submitted by**

| Student Name: | **Rahul Satish Vadhyar**  **Saikalyan Sathish** |
| --- | --- |
| USN: | **1BM21AI099**  **1BM21AI111** |
| Date: | **28-01-2024** |
| Semester & Section: | **5C** |
| Total Pages: | **17** |
| Student Signature: |  |

**Valuation Report (to be filled by the faculty)**

| Score: |  |
| --- | --- |
| Comments: |  |
| Faculty In-charge: | **SHRADHA NAIK** |
| Faculty Signature:  with date |  |

1. **Introduction and description of problem statement**

In the realm of game development, the creation of classic arcade games serves as an engaging way for programmers to hone their skills while exploring the principles of software design. This report documents the development of a Python-based Snake game using the Pygame library, with a special emphasis on adhering to the SOLID principles. These principles, introduced by Robert C. Martin, are a set of five design principles that aim to enhance the maintainability, flexibility, and scalability of object-oriented software.

SOLID is an acronym representing Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. Each of these principles plays a crucial role in fostering a modular, extensible, and robust codebase. The application of SOLID principles in game development not only results in code that is easier to understand and maintain but also provides a solid foundation for future enhancements and modifications.

The Snake game is a classic and straightforward arcade experience that has captivated players for decades. In this game, players control a snake that traverses a two-dimensional grid or arena. The primary objective is to consume food items, represented by distinct symbols on the grid, in order to grow the length of the snake. As the snake consumes more food, its size increases, intensifying the challenge of navigating the confined space.

The game introduces a unique twist with a crucial constraint: the snake continually moves in a single direction, and players must strategically manipulate its path to avoid collisions with the game boundaries and, critically, its own tail. With each successful consumption of food, the player earns points, fostering a sense of accomplishment and competition. The game continues until the snake collides with a wall or itself, at which point the player's score is tallied, and they may choose to embark on a new attempt to surpass their previous achievements. The simplicity of its mechanics combined with the progressively escalating difficulty makes the Snake game an enduring and enjoyable test of reflexes and strategic thinking.

1. **Code**

file : main.py

import game

game = game.Game()

game.run()

file: snake.py

import pygame

class Snake:

"""

The Snake class represents the snake in the game.

Attributes:

-----------

\_\_screen : pygame.Surface

The main screen surface where the snake is displayed.

\_\_blockSize : int

The size of each block of the snake.

positions : list

A list of positions of the blocks of the snake. Each position is represented as a list of two integers.

direction : str

The current direction of the snake. It can be 'right', 'left', 'up', or 'down'.

"""

def \_\_init\_\_(self, screen, blockSize):

self.\_\_screen = screen

self.\_\_blockSize = blockSize

self.positions = [[500,500],[490,500],[480,500]]

self.direction = 'right'

def update(self):

"""

Updates the position of the snake based on its current direction.

If the snake hits itself, it returns False. Otherwise, it returns True.

"""

head = self.positions[0].copy()

if self.direction == 'right':

head[0] += self.\_\_blockSize

if self.direction == 'left':

head[0] -= self.\_\_blockSize

if self.direction == 'up':

head[1] -= self.\_\_blockSize

if self.direction == 'down':

head[1] += self.\_\_blockSize

for position in self.positions[:-1]:

if head[0] == position[0] and head[1] == position[1]:

return False

self.positions.insert(0, head)

self.positions.pop()

return True

def addBlock(self, position):

"""

Adds a new block to the snake at the given position.

"""

self.positions.insert(0, position)

def draw(self):

"""

Draws the snake on the screen.

"""

for position in self.positions:

pygame.draw.rect(self.\_\_screen, (0,0,255), (\*position, self.\_\_blockSize, self.\_\_blockSize))

file: SnakeScene.py

from gameScene import GameScene

from snake import Snake

import pygame

import random

class SnakeScene(GameScene):

"""

The SnakeScene class represents the snake game scene. It inherits from the GameScene class and implements the drawScene and update methods.

Attributes:

-----------

screen : pygame.Surface

The main screen surface where the game is displayed.

blockSize : int

The size of each block of the snake.

snake : Snake

The snake object.

font : pygame.font.Font

The font used to render the text.

gameOver : bool

A flag indicating whether the game is over.

appleLocation : list

The location of the apple. It is represented as a list of two integers.

score : int

The current score of the game.

running : bool

A flag indicating whether the game is running.

"""

def \_\_init\_\_(self, screen):

self.screen = screen

self.blockSize = 10

self.snake = Snake(self.screen, self.blockSize)

self.font = pygame.font.Font('freesansbold.ttf', 23)

self.gameOver = False

self.appleLocation = [900,900]

self.score = 0

self.running = True

def update(self):

"""

Gets the user input, updates the snake and the apple, and returns the running and gameOver flags.

"""

self.\_\_getInput()

self.gameOver = not self.snake.update()

if self.snake.positions[0][0] == self.appleLocation[0] and self.snake.positions[0][1] == self.appleLocation[1]:

self.score += 1

self.snake.addBlock(self.appleLocation)

while(self.appleLocation in self.snake.positions):

self.appleLocation = [random.randint(self.blockSize + 10,self.blockSize\*10 - 10)\*self.blockSize, random.randint(self.blockSize + 10,self.blockSize\*10 - 10)\*self.blockSize]

self.gameOver = self.snake.positions[0][0] < 0 or \

self.snake.positions[0][0] > 1000 or \

self.snake.positions[0][1] < 0 or \

self.snake.positions[0][1] > 1000

return self.running, self.gameOver

def drawScene(self):

"""

Draws the snake scene. It displays the snake, the apple, and the score.

"""

self.snake.draw()

self.\_\_drawApple()

self.\_\_displayScore()

def \_\_drawApple(self):

"""

Draws the apple on the screen.

"""

pygame.draw.rect(self.screen, (255,0,0), (\*self.appleLocation, self.blockSize, self.blockSize))

def \_\_displayScore(self):

"""

Displays the current score.

"""

#display score

score = self.font.render('Score:' + str(self.score), True, (255,255,255), (0,0,0))

scoreRect = score.get\_rect()

scoreRect.center = (50,30)

self.screen.blit(score, scoreRect)

def \_\_getInput(self):

"""

Gets the user input. If the user presses the arrow keys, it changes the direction of the snake. If the user presses the escape key or closes the window, it stops the game.

"""

#get all the events

for event in pygame.event.get():

#if the event is quit, quit the game

if event.type == pygame.QUIT:

self.running = False

#Get the keypresses and assign the directions

if not self.gameOver:

if event.type == pygame.KEYDOWN:

if event.key == pygame.K\_d and self.snake.direction != 'left':

self.snake.direction = 'right'

if event.key == pygame.K\_a and self.snake.direction != 'right':

self.snake.direction = 'left'

if event.key == pygame.K\_s and self.snake.direction != 'up':

self.snake.direction = 'down'

if event.key == pygame.K\_w and self.snake.direction != 'down':

self.snake.direction = 'up'

if event.key == pygame.K\_ESCAPE:

self.running = False

file: mainMenu.py

from gameScene import GameScene

import pygame

class MainMenuScene(GameScene):

"""

The MainMenuScene class represents the main menu scene of the game.

It inherits from the GameScene class and implements the drawScene and update methods.

Attributes:

-----------

screen : pygame.Surface

The main screen surface where the game is displayed.

font : pygame.font.Font

The font used to render the text.

running : bool

A flag indicating whether the game is running.

start : bool

A flag indicating whether the game should start.

"""

def \_\_init\_\_(self, screen):

self.screen = screen

self.font = pygame.font.Font('freesansbold.ttf', 23)

self.running = True

self.start = False

def update(self):

"""

Gets the user input and returns the running and start flags.

"""

self.\_\_getInput()

return self.running, self.start

def drawScene(self):

"""

Draws the main menu scene. It displays the title and the instructions.

"""

self.\_\_displayTitle()

self.\_\_displayInstructions()

def \_\_displayTitle(self):

"""

Displays the game title.

"""

#display title

title = self.font.render('Snake Game', True, (255,255,255), (0,0,0))

titleRect = title.get\_rect()

titleRect.center = (500,500)

self.screen.blit(title, titleRect)

def \_\_displayInstructions(self):

"""

Displays the game instructions.

"""

#display instructions

instructions = self.font.render('Press any key to start', True, (255,255,255), (0,0,0))

instructionsRect = instructions.get\_rect()

instructionsRect.center = (500,550)

self.screen.blit(instructions, instructionsRect)

def \_\_getInput(self):

"""

Gets the user input. If the user presses any key, the game starts.

If the user presses the escape key or closes the window, the game quits.

"""

#get all the events

for event in pygame.event.get():

#if the event is quit, quit the game

if event.type == pygame.QUIT:

self.running = False

#Get the keypresses and assign the directions

if event.type == pygame.KEYDOWN:

if event.key == pygame.K\_ESCAPE:

self.running = False

else:

self.start = True

file: gameScene.py

from abc import ABC, abstractmethod

class GameScene(ABC):

"""

The GameScene class is an abstract base class that represents a game scene.

It provides the basic structure for different game scenes like the main menu scene,

the snake game scene, and the game over scene.

"""

@abstractmethod

def drawScene(self):

"""

An abstract method that should be implemented in the subclasses to draw the game scene.

"""

pass

@abstractmethod

def update(self):

"""

An abstract method that should be implemented in the subclasses to update the game scene.

"""

pass

file : GameOverScene.py

from gameScene import GameScene

import pygame

class GameOverScene(GameScene):

"""

The GameOverScene class represents the game over scene. It displays the game over message and the final score.

Attributes:

-----------

screen : pygame.Surface

The main screen surface where the game over scene is displayed.

score : int

The final score of the game.

font : pygame.font.Font

The font used to render the text.

running : bool

A flag indicating whether the game is running.

gameOver : bool

A flag indicating whether the game is over.

"""

def \_\_init\_\_(self, screen):

"""

Initializes the game over scene with the given screen.

"""

self.screen = screen

self.score = 0

self.font = pygame.font.Font('freesansbold.ttf', 23)

self.running = True

self.gameOver = True

def update(self):

"""

Updates the game over scene. It gets the user input and returns the running and gameOver flags.

"""

self.\_\_getInput()

return self.running, self.gameOver

def drawScene(self):

"""

Draws the game over scene. It displays the score and the game over message.

"""

self.\_\_displayScore()

self.\_\_displayGameOver()

def \_\_displayScore(self):

"""

Displays the final score.

"""

#display score

score = self.font.render('Score:' + str(self.score), True, (255,255,255), (0,0,0))

scoreRect = score.get\_rect()

scoreRect.center = (500,500)

self.screen.blit(score, scoreRect)

def \_\_displayGameOver(self):

"""

Displays the game over message.

"""

#display game over

gameOver = self.font.render('Game Over', True, (255,255,255), (0,0,0))

gameOverRect = gameOver.get\_rect()

gameOverRect.center = (500,550)

anyKey = self.font.render('Press any key to restart', True, (255,255,255), (0,0,0))

anyKeyRect = anyKey.get\_rect()

anyKeyRect.center = (500,600)

self.screen.blit(gameOver, gameOverRect)

self.screen.blit(anyKey, anyKeyRect)

def \_\_getInput(self):

"""

Gets the user input. If the user quits the game or presses the escape key, it stops the game. If the user presses any other key, it restarts the game.

"""

#get all the events

for event in pygame.event.get():

#if the event is quit, quit the game

if event.type == pygame.QUIT:

self.running = False

#Get the keypresses and assign the directions

if event.type == pygame.KEYDOWN:

if event.key == pygame.K\_ESCAPE:

self.running = False

else:

self.gameOver = False

file: game.py

import pygame

from SnakeScene import SnakeScene

from GameOverScene import GameOverScene

from mainMenuScene import MainMenuScene

class Game:

"""

The Game class represents the main game loop. It initializes the game window,

manages the game scenes and controls the game flow.

Attributes:

-----------

screen : pygame.Surface

The main screen surface where the game is displayed.

clock : pygame.time.Clock

A clock object that can be used to track the amount of time.

snakeScene : SnakeScene

The scene for the snake game.

gameOver : bool

A flag indicating whether the game is over.

gameOverScene : GameOverScene

The scene displayed when the game is over.

mainMenuScene : MainMenuScene

The main menu scene.

Methods:

--------

\_\_init\_\_() -> None:

Initializes pygame, sets the game window and initializes the game scenes.

run() -> None:

The main game loop. It controls the game flow and updates the game scenes.

"""

def \_\_init\_\_(self):

#initalize pygame and set caption, icon and window size

pygame.init()

pygame.display.set\_caption("Snake Game")

logo = pygame.image.load('../snake.png')

pygame.display.set\_icon(logo)

self.screen = pygame.display.set\_mode((1000,1000))

#clock initalization

self.clock = pygame.time.Clock()

self.snakeScene = SnakeScene(self.screen)

self.gameOver = False

self.gameOverScene = GameOverScene(self.screen)

self.mainMenuScene = MainMenuScene(self.screen)

def run(self):

"""

The main game loop. It controls the game flow and updates the game scenes.

The game starts with the main menu scene. When the player starts the game,

it switches to the snake scene. If the player loses, it switches to the game over scene.

The player can restart the game from the game over scene.

"""

running = True

start = False

while running:

if not start:

running, start = self.mainMenuScene.update()

self.screen.fill((0,0,0))

self.mainMenuScene.drawScene()

#update display

pygame.display.update()

#set framerate

self.clock.tick(10)

else:

break

while running:

if not self.gameOver:

running, self.gameOver = self.snakeScene.update()

if self.gameOver:

self.gameOverScene.score = self.snakeScene.score

else:

running, self.gameOver = self.gameOverScene.update()

if not self.gameOver:

self.snakeScene = SnakeScene(self.screen)

self.screen.fill((0,0,0))

if not self.gameOver:

self.snakeScene.drawScene()

else:

self.gameOverScene.drawScene()

#update display

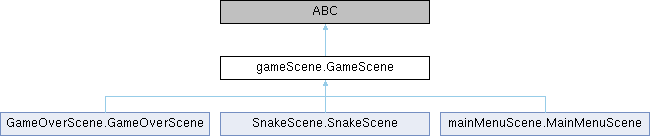
pygame.display.update()

#set framerate

self.clock.tick(10)

1. **Explaination of code**

The provided code implements a simple snake game using the Pygame library in Python. The game follows an object-oriented design and is structured around different scenes: MainMenuScene, GameOverScene, and SnakeScene. These scenes inherit from the abstract class GameScene, which defines methods for drawing scenes and updating game states.



The Snake class manages the snake's state, direction, and behavior, while the SnakeScene class handles the main game logic, including user input, updating the snake's position, and managing collisions with the apple. The game incorporates the SOLID principles, emphasizing modularity, maintainability, and extensibility.

The Game class orchestrates the flow of the game, managing transitions between scenes such as the main menu, gameplay, and game-over screens. The code leverages Pygame for graphics and user input handling. Overall, this implementation showcases a structured and modular approach to building a simple snake game in Python.

Now let's analyze how the given code adheres to the SOLID principles:

1. Single Responsibility Principle (SRP): The MainMenuScene, GameOverScene, Snake, and SnakeScene classes each have a clear and single responsibility. For example:

* MainMenuScene handles the main menu and user input for starting the game.
* GameOverScene handles the game-over screen and user input for restarting the game.
* Snake manages the state and behavior of the snake in the game.
* SnakeScene manages the game scene, including updating the game state and rendering.

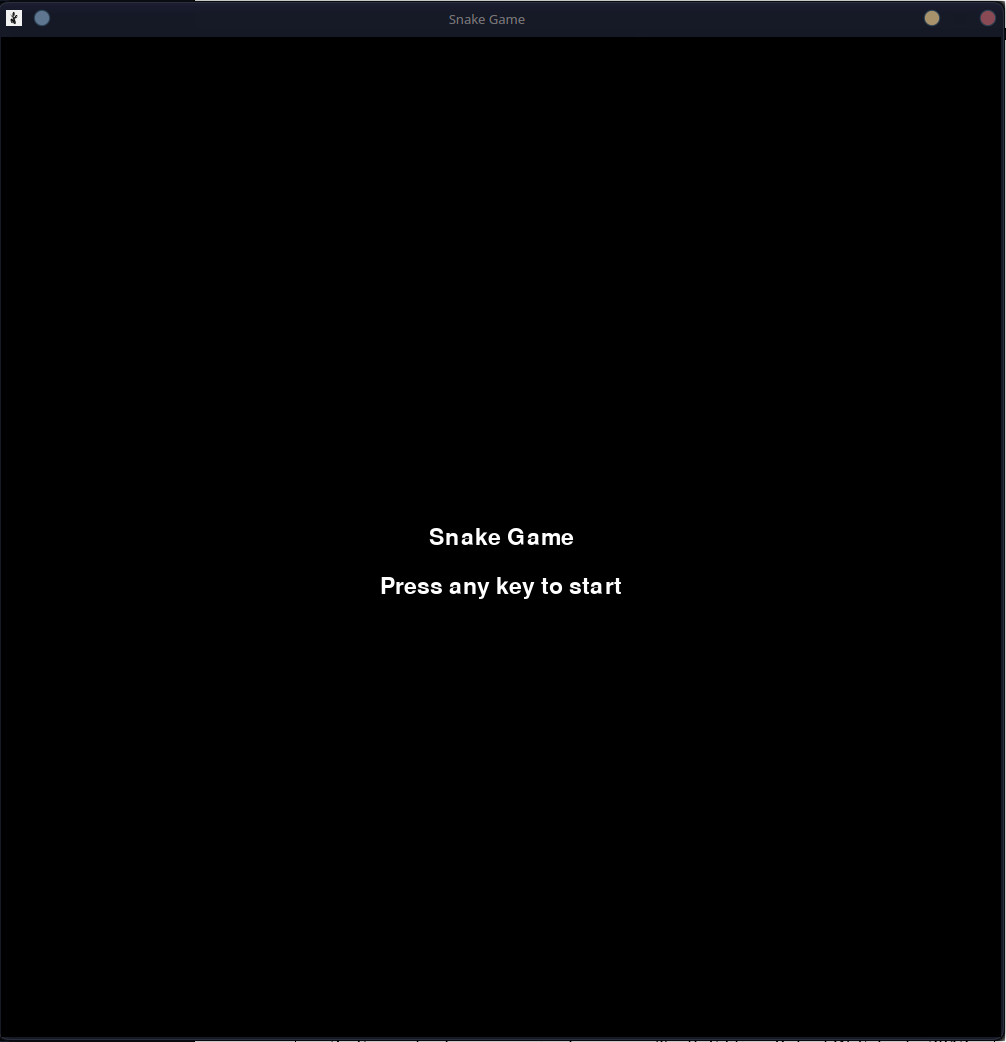
2. Open/Closed Principle (OCP): The code follows the Open/Closed Principle as it allows for extending the functionality without modifying existing code. For example, new scenes or game features can be added without modifying the existing scenes or game logic. Each scene (e.g., MainMenuScene, GameOverScene, SnakeScene) can be extended independently.

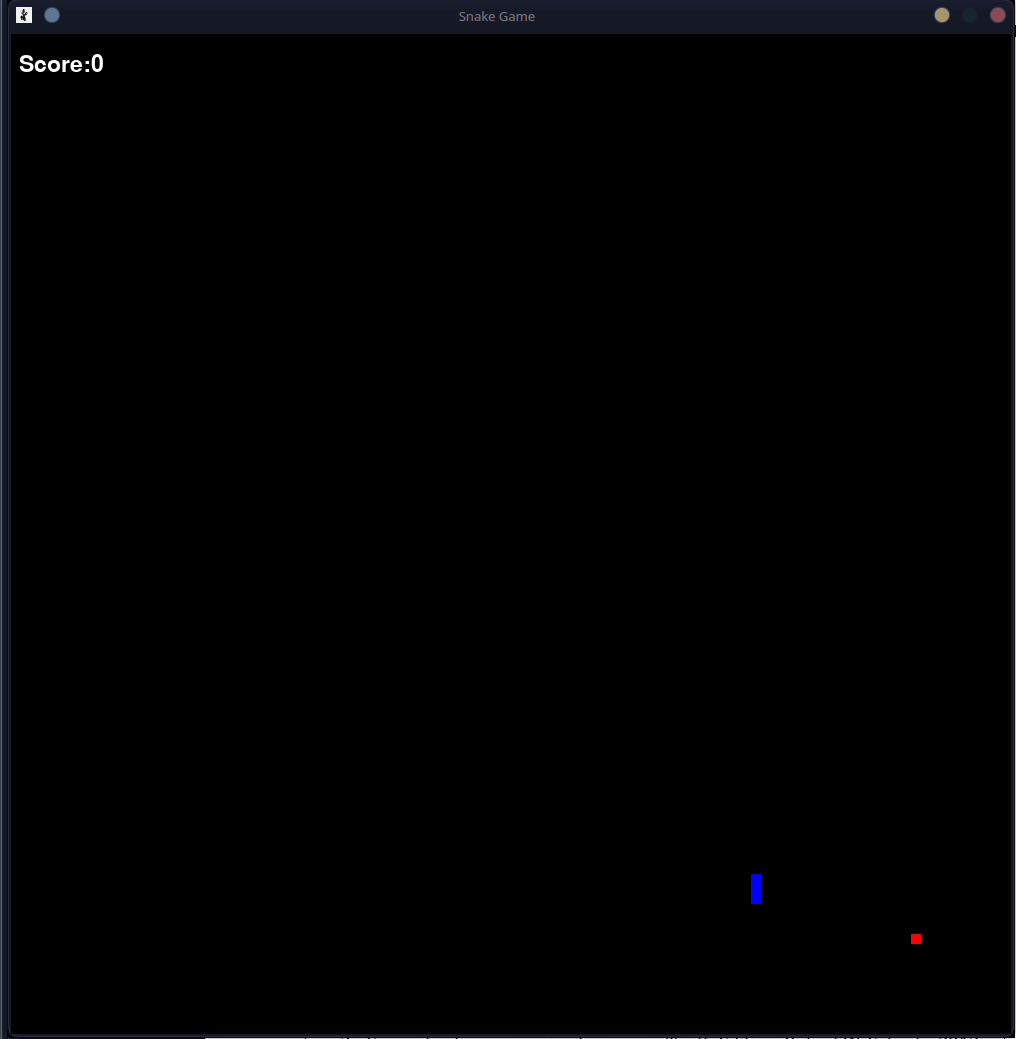
3. Liskov Substitution Principle (LSP): The code generally follows the Liskov Substitution Principle, as subclasses (MainMenuScene, GameOverScene, SnakeScene) can be used interchangeably with their base class (GameScene). This allows for flexibility in managing different scenes within the game.

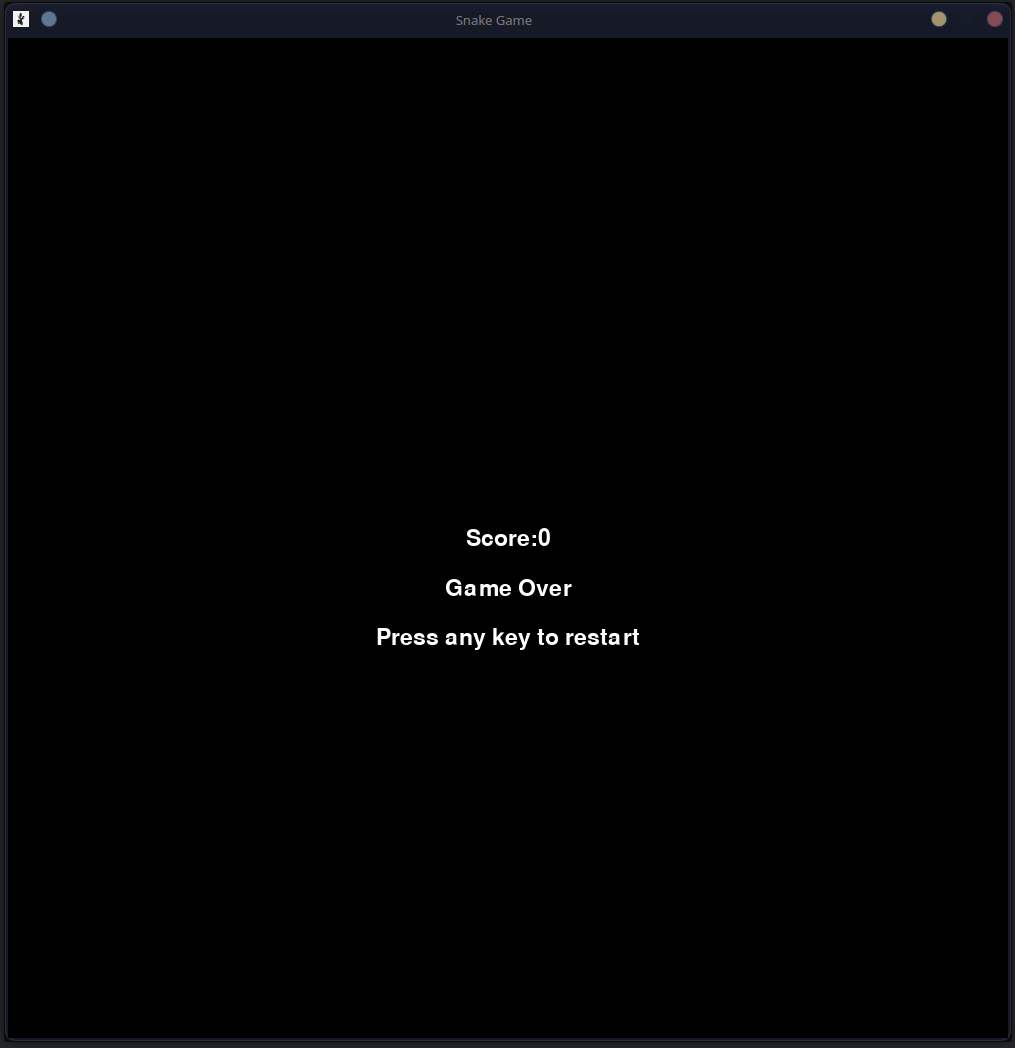
4. Interface Segregation Principle (ISP): The code does not explicitly implement the Interface Segregation Principle, but it doesn't violate it either. Each concrete class (e.g., MainMenuScene, GameOverScene, SnakeScene) implements all methods from the GameScene interface.

5. Dependency Inversion Principle (DIP): The code adheres to the Dependency Inversion Principle as it depends on abstractions (GameScene and its methods) rather than concrete implementations. For instance, the Game class interacts with the GameScene interface, allowing for different scenes to be used without modifying the Game class itself.

1. **Output screenshots**

****

****

****

1. **Conclusion**

In conclusion, the development of the Snake game using Python and Pygame, enriched by the application of SOLID principles, showcases the fusion of classic gaming with modern software design methodologies. By adhering to the principles of Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion, the project has not only produced a functional and enjoyable game but has also laid the groundwork for a scalable and maintainable codebase.

The journey through the implementation of the Snake game has underscored the importance of modular and well-organized code, fostering code reusability and ease of extension. The integration of SOLID principles has provided a clear architectural structure, allowing for seamless modifications and enhancements, contributing to the longevity of the codebase.

Moreover, the exploration of the Snake game's mechanics has highlighted the enduring appeal of this classic arcade experience. The simple yet challenging gameplay, where players strategically guide the snake through the digital arena, consuming food to grow while avoiding collisions, continues to captivate and entertain. As a testament to the versatility of Python and the accessibility of Pygame, the project serves as an entry point for aspiring game developers and a practical illustration of how SOLID principles can elevate the quality of software design.

In essence, the creation of the Snake game is not merely a technical exercise but a fusion of art and science, demonstrating how a classic gaming experience can be revitalized through thoughtful software architecture. This endeavor serves as an inspiration for future game developers to not only embrace the joy of creating games but also to do so with a commitment to sound design principles, ensuring a foundation that stands the test of time.