**The Princess and the Dragon – Escape from the Castle**

****

**Team Members:**

**Maria Dubrovsky** – 323480533  
**Daniela Lavi** – 318608395

**Gameplay Video:**  
<https://youtube.com/shorts/WBco6Go-j-w?feature=share>

**Presentation Video:**  
<https://youtu.be/4AhVj6XVq-g>

**GitHub Repository:**  
<https://github.com/mashadu99/Princess-Maze>

**General Background:**

As part of the course requirements, we were asked to implement an advanced algorithmic solution within an interactive system. We chose to develop a maze adventure game called "Princess Maze", in which a princess must escape a haunted castle while being chased by a dragon.

The game features three levels, each introducing a new layer of complexity:

Level 1 – Basic movement and health system using hearts

Level 2 – Addition of ice tiles, which cause the player to slip unpredictably to preset locations

**L**evel 3 – Addition of mud tiles, which slow down the player's movement

The princess starts with three hearts, and each time the dragon catches her, she loses one. When all hearts are gone, the game ends. To successfully complete a level, the player must collect a key and reach the castle gate.

Each level is structured as a unique maze, gradually increasing in size and complexity — with more walls, loops, and dead ends. Strategic placement of ice, mud, hearts, and keys was essential to balance difficulty and ensure fair gameplay, giving the player enough room to strategize and adapt.

**Problem We Identified:**

The core challenge was to make the dragon’s behavior feel intelligent and reactive — not based on hardcoded paths. We needed a way for the dragon to dynamically adapt its movement to the terrain, the real-time changes, and the player’s decisions.

Another challenge was designing a maze that strikes the right balance: **difficult but fair** — ensuring that both the princess and the dragon had access to key areas of the map without getting trapped or breaking the game’s logic.

**Our Algorithmic Solution:**

We implemented the A\* (A-Star) pathfinding algorithm for the dragon, allowing it to calculate the shortest path to the princess at each turn. The algorithm also considered tile costs — for example, ice tiles were cheaper to cross than mud tiles — giving the dragon the ability to make weighted decisions about movement.

Additionally, during the maze design phase, we used Breadth-First Search (BFS) to verify that each level was solvable - ensuring that:

The princess could always reach the exit gate-

The dragon always had a path to the princess-

The key could be collected through a valid path-

**System Architecture:**

Built using Python and the Pygame library for interactive graphics and animations

-Three progressively challenging levels with unique maps

A centralized Game class that controls all game logic-

A separate A\* implementation in src/pathfinding/astar.py-

-Modular file structure with reusable components

Internal testing and documentation throughout development-

**Insights & Reflections:**

This project required us to think algorithmically, understanding how cost-based heuristics influence decision-making in real-time systems. Implementing A\* in a visual, interactive environment gave us a much deeper appreciation for how theoretical algorithms are brought to life in games and simulations.

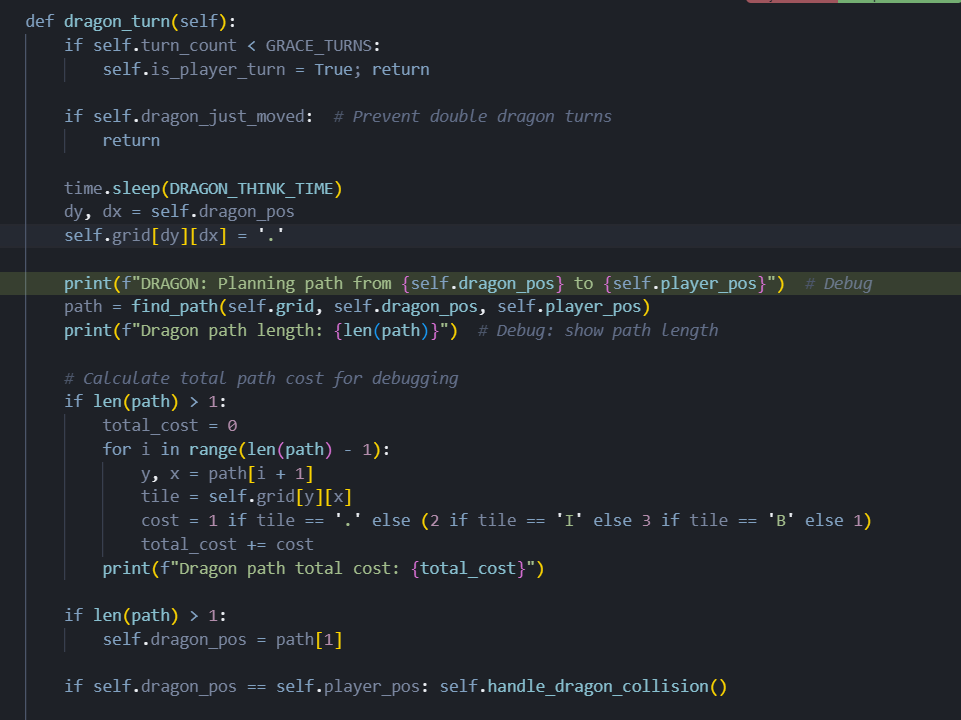
We also learned how to design fair and engaging challenges, making the gameplay both satisfying and beatable. Balancing fun and frustration required thoughtful level design, careful placement of obstacles, and tuning of AI behavior.

Ultimately, this experience gave us a better understanding of:

Integrating search algorithms into game logic--

Designing for both human and AI agents-

The importance of dynamic environments and stateful decision-making-



תמונה שמכילה טקסט, צילום מסך, תוכנה, תכונות מולטימדיה

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

*# C:\Users\User\Desktop\maze\_game\main.py*

import pygame

import sys

import time

from copy import deepcopy

from grid\_visualizer import GridVisualizer

from src.pathfinding.astar import find\_path

*# הגדרות כלליות*

TILE\_SIZE = 48

GRACE\_TURNS = 3

PLAYER\_HEARTS = 3

DRAGON\_THINK\_TIME = 0.5 *# שניות*

LEVELS = [

*# שלב 1 - קטן ופשוט*

    {"grid": [

        ['W','W','W','W','W','W','W','W','W','W'],

        ['W','P','.','.','.','.','.','K','.','W'],

        ['W','.','W','W','.','W','W','.','W','W'],

        ['W','.','.','.','.','.','.','.','.','W'],

        ['W','.','W','.','W','W','W','.','W','W'],

        ['W','.','.','.','.','.','.','.','.','W'],

        ['W','W','W','.','W','.','W','W','.','W'],

        ['W','.','.','.','.','D','.','.','.','W'],

        ['W','.','W','W','W','W','W','.','G','W'],

        ['W','W','W','W','W','W','W','W','W','W']

    ]},

*# שלב 2 - תוספת קרח ומפתח*

    {"grid": [

        ['W','W','W','W','W','W','W','W','W','W','W','W'],

        ['W','P','.','.','W','.','.','I','.','.','K','W'],

        ['W','.','W','.','W','.','W','W','W','W','.','W'],

        ['W','D','.','.','.','.','.','.','.','W','.','W'],

        ['W','.','W','W','W','W','W','.','W','W','.','W'],

        ['W','.','.','.','.','.','.','.','.','.','.','W'],

        ['W','W','W','W','.','W','W','I','W','W','.','W'],

        ['W','.','.','.','.','.','.','.','.','.','.','W'],

        ['W','.','W','W','W','W','W','W','W','.','.','W'],

        ['W','.','.','.','.','.','.','.','.','.','.','W'],

        ['W','W','W','W','W','W','W','W','W','W','G','W']

    ]},

*# שלב 3 - גריד גדול עם בוץ, קרח, לב נוסף*

    {"grid": [

        ['W','W','W','W','W','W','W','W','W','W','W','W','W','W','W'],

        ['W','P','.','.','W','.','.','I','.','.','K','.','.','H','W'],

        ['W','.','W','.','W','.','W','W','W','W','.','W','.','.','W'],

        ['W','D','.','.','.','.','.','.','.','W','.','W','.','.','W'],

        ['W','.','W','W','W','W','W','B','W','W','.','W','.','.','W'],

        ['W','.','.','B','.','.','.','.','.','.','.','W','.','.','W'],

        ['W','W','W','W','.','W','W','I','W','W','.','W','B','.','W'],

        ['W','.','.','.','B','.','.','.','.','.','.','W','.','.','W'],

        ['W','.','W','W','W','W','W','W','W','B','.','W','.','.','W'],

        ['W','.','.','.','.','.','B','.','.','.','.','W','.','.','W'],

        ['W','.','W','W','W','W','W','W','W','W','.','W','.','.','W'],

        ['W','.','.','.','B','.','.','.','.','.','.','W','.','.','W'],

        ['W','.','W','W','W','W','W','W','W','W','.','W','B','.','W'],

        ['W','.','.','.','.','.','.','B','.','.','.','W','.','.','W'],

        ['W','W','W','W','W','W','W','W','W','W','W','W','W','G','W']

    ]}

]

class Game:

    def \_\_init\_\_(self, screen):

        self.screen = screen

        self.visualizer = GridVisualizer(TILE\_SIZE)

        self.running = True

        self.clock = pygame.time.Clock()

        self.dragon\_just\_moved = False  *# Flag to prevent double dragon turns*

        self.player\_slipped = False  *# Flag to track if player slipped on ice*

        self.reset\_game()

    def reset\_game(self, level\_idx=0):

        self.game\_state = 'playing'

        self.level\_idx = level\_idx

        self.turn\_count = 0

        self.is\_player\_turn = True

        self.message, self.message\_timer = "", 0

        self.setup\_level()

    def setup\_level(self):

        self.grid = deepcopy(LEVELS[self.level\_idx]["grid"])

        self.player\_pos = self.\_find\_char('P')

        self.dragon\_pos = self.\_find\_char('D')

        self.start\_pos = self.player\_pos

*# שמירת מספר הלבבות הנוכחי (לא מאפס בכל שלב)*

        if not hasattr(self, 'hearts'):

            self.hearts = PLAYER\_HEARTS

        self.has\_key = False

        self.set\_message(f"Level {self.level\_idx + 1}", 3)

    def \_find\_char(self, char):

        for y, r in enumerate(self.grid):

            if char in r: return (y, r.index(char))

    def set\_message(self, text, duration):

        self.message = text

        self.message\_timer = time.time() + duration

    def handle\_input(self):

        if not self.is\_player\_turn:

            return  *# Prevent input when it's not the player's turn*

        for event in pygame.event.get():

            if event.type == pygame.QUIT: self.running = False

            if event.type == pygame.KEYDOWN:

                if self.game\_state != 'playing':

                    if event.key == pygame.K\_r: self.reset\_game(0 if self.game\_state == 'win' else self.level\_idx)

                elif self.is\_player\_turn and not self.player\_slipped:  *# Prevent movement if slipped*

                    moves = {pygame.K\_UP: (-1, 0), pygame.K\_DOWN: (1, 0), pygame.K\_LEFT: (0, -1), pygame.K\_RIGHT: (0, 1)}

                    if event.key in moves: self.move\_player(\*moves[event.key])

    def move\_player(self, dy, dx):

        py, px = self.player\_pos

        ny, nx = py + dy, px + dx

        if not (0 <= ny < len(self.grid) and 0 <= nx < len(self.grid[0])) or self.grid[ny][nx] == 'W': return

        self.grid[py][px] = '.'

        target\_tile = self.grid[ny][nx]

        if target\_tile == 'K':

            self.has\_key = True

            self.set\_message("You found the key!", 2)

            self.player\_pos = (ny, nx)

        elif target\_tile == 'H':

            if self.hearts < PLAYER\_HEARTS:

                self.hearts += 1

                self.set\_message("Extra heart collected!", 2)

            else:

                self.set\_message("You're already at full health!", 2)

            self.player\_pos = (ny, nx)

        elif target\_tile == 'G':

            if self.has\_key:

                if self.level\_idx + 1 < len(LEVELS):

                    self.level\_idx += 1

                    self.setup\_level()

                    self.set\_message("Level Complete!", 2)

                else:

                    self.game\_state = 'win'

                return

            else:

                self.set\_message("The gate is locked!", 2)

                self.grid[py][px] = 'P'; return

        elif target\_tile == 'I':

            print(f"ICE: Slipping from {self.player\_pos} back to {self.start\_pos}")  *# דיבאג*

            self.set\_message("Slipped on ice!", 2)

            self.player\_pos = self.start\_pos  *# Return to start position*

            print(f"ICE: Princess returned to start position: {self.player\_pos}")  *# דיבאג*

        elif target\_tile == 'B':

            print(f"MUD: Player stuck in mud at {(ny, nx)}")  *# Debug*

            self.set\_message("Stuck in mud!", 1)

            self.grid[ny][nx] = '.'  *# הבוץ נשבר והופך לרצפה רגילה*

            self.grid[py][px] = 'P'  *# משאירה את עצמה באותו מקום*

            self.end\_player\_turn()

            return

        elif (ny, nx) == self.dragon\_pos:

            self.handle\_dragon\_collision()

            self.grid[py][px] = 'P'; self.end\_player\_turn(); return

        else: self.player\_pos = (ny, nx)

        self.grid[self.player\_pos[0]][self.player\_pos[1]] = 'P'

        self.end\_player\_turn()

    def handle\_dragon\_collision(self):

        self.hearts -= 1

        self.set\_message("Ouch! A dragon hit you!", 2)

        if self.hearts <= 0:

            self.game\_state = 'lose'

        else:

*# איפוס השלב הנוכחי במקום כל המשחק*

            self.setup\_level()

            self.set\_message(f"Level {self.level\_idx + 1} restarted! Hearts remaining: {self.hearts}", 3)

    def end\_player\_turn(self):

        self.turn\_count += 1

        self.is\_player\_turn = False

        self.player\_slipped = False *# Reset slipped flag when player ends turn*

        self.dragon\_just\_moved = False  *# Reset flag when player ends turn*

    def dragon\_turn(self):

        if self.turn\_count < GRACE\_TURNS:

            self.is\_player\_turn = True; return

        if self.dragon\_just\_moved:  *# Prevent double dragon turns*

            return

        time.sleep(DRAGON\_THINK\_TIME)

        dy, dx = self.dragon\_pos

        self.grid[dy][dx] = '.'

        print(f"DRAGON: Planning path from {self.dragon\_pos} to {self.player\_pos}")  *# Debug*

        path = find\_path(self.grid, self.dragon\_pos, self.player\_pos)

        print(f"Dragon path length: {len(path)}")  *# Debug: show path length*

*# Calculate total path cost for debugging*

        if len(path) > 1:

            total\_cost = 0

            for i in range(len(path) - 1):

                y, x = path[i + 1]

                tile = self.grid[y][x]

                cost = 1 if tile == '.' else (2 if tile == 'I' else 3 if tile == 'B' else 1)

                total\_cost += cost

            print(f"Dragon path total cost: {total\_cost}")

        if len(path) > 1:

            self.dragon\_pos = path[1]

        if self.dragon\_pos == self.player\_pos: self.handle\_dragon\_collision()

        dy, dx = self.dragon\_pos

        self.grid[dy][dx] = 'D'

        self.dragon\_just\_moved = True  *# Set flag to prevent double moves*

        self.is\_player\_turn = True

    def run(self):

        while self.running:

            self.handle\_input()

            if self.game\_state == 'playing' and not self.is\_player\_turn: self.dragon\_turn()

            if time.time() > self.message\_timer: self.message = ""

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

            self.visualizer.draw\_grid(self.screen, self.grid)

            self.visualizer.draw\_hud(self.screen, self.hearts, self.has\_key, self.level\_idx + 1, self.turn\_count, GRACE\_TURNS)

            if self.message: self.visualizer.draw\_message(self.screen, self.message)

            if self.game\_state != 'playing': self.visualizer.draw\_game\_over(self.screen, self.game\_state == 'win')

            pygame.display.flip()

            self.clock.tick(60)

if \_\_name\_\_ == '\_\_main\_\_':

    pygame.init()

    max\_w = max(len(lvl["grid"][0]) for lvl in LEVELS)

    max\_h = max(len(lvl["grid"]) for lvl in LEVELS)

    screen = pygame.display.set\_mode((max\_w \* TILE\_SIZE, max\_h \* TILE\_SIZE + 60))

    pygame.display.set\_caption("Princess Maze - A\* Agent")

    Game(screen).run()

    pygame.quit()

    sys.exit()

import heapq

def movement\_cost(tile):

    """

    מחשב את עלות התנועה למשבצת מסוימת.

    """

    if tile == 'B':  *# בוץ*

        return 3

    elif tile == 'I':  *# קרח*

        return 2

    else:  *# משבצת רגילה*

        return 1

def manhattan\_distance(pos1, pos2):

    """

    מחשב את מרחק מנהטן בין שתי נקודות.

    זוהי היוריסטיקה שלנו - הערכה אופטימית של המרחק.

    """

    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

def find\_path(grid, start, end):

    """

    אלגוריתם A\* למציאת הנתיב הקצר ביותר עם תמיכה בעלויות תנועה שונות.

    :param grid: רשימה של רשימות המייצגת את מפת המשחק.

    :param start: קואורדינטת התחלה (y, x).

    :param end: קואורדינטת יעד (y, x).

    :return: רשימת קואורדינטות של הנתיב, או רשימה ריקה אם לא נמצא נתיב.

    """

    print(f"A\*: Finding path from {start} to {end}")  *# Debug*

    open\_list = [(0, start, [])]  *# ערימת מינימום: (f\_cost, position, path)*

    closed\_set = set()

    g\_costs = {start: 0}

    while open\_list:

        current\_f, current\_pos, path = heapq.heappop(open\_list)

        if current\_pos == end:

            print(f"A\*: Found path with total cost {g\_costs[current\_pos]}")  *# Debug*

            return path + [current\_pos]

        if current\_pos in closed\_set:

            continue

        closed\_set.add(current\_pos)

        y, x = current\_pos

        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  *# 4 כיוונים*

        for dy, dx in directions:

            neighbor\_pos = (y + dy, x + dx)

            ny, nx = neighbor\_pos

            if not (0 <= ny < len(grid) and 0 <= nx < len(grid[0])):

                continue

*# הדרקון מתייחס לקירות כמכשולים*

            if grid[ny][nx] == 'W':

                continue

*# חישוב עלות התנועה למשבצת השכנה*

            tile\_cost = movement\_cost(grid[ny][nx])

            if tile\_cost > 1:  *# Debug: show when considering expensive tiles*

                print(f"A\*: Considering tile {grid[ny][nx]} at {neighbor\_pos} with cost {tile\_cost}")

            new\_g\_cost = g\_costs[current\_pos] + tile\_cost

            if neighbor\_pos not in g\_costs or new\_g\_cost < g\_costs[neighbor\_pos]:

                g\_costs[neighbor\_pos] = new\_g\_cost

                h\_cost = manhattan\_distance(neighbor\_pos, end)

                f\_cost = new\_g\_cost + h\_cost

                print(f"A\*: Adding {neighbor\_pos} with g\_cost={new\_g\_cost}, h\_cost={h\_cost}, f\_cost={f\_cost}")  *# Debug*

                heapq.heappush(open\_list, (f\_cost, neighbor\_pos, path + [current\_pos]))

    print("A\*: No path found")  *# Debug*

    return [] *# לא נמצא נתיב*

*# C:\Users\User\Desktop\maze\_game\grid\_visualizer.py*

import pygame

import os

class GridVisualizer:

    """

    מחלקה האחראית על כל הויזואליזציה של המשחק.

    טוענת את התמונות ומציירת את המפה, השחקנים והממשק.

    """

    def \_\_init\_\_(self, tile\_size):

        self.tile\_size = tile\_size

        self.images = self.\_load\_images()

        try:

            self.font = pygame.font.Font(os.path.join('assets', 'fonts', 'PressStart2P-Regular.ttf'), 24)

            self.small\_font = pygame.font.Font(os.path.join('assets', 'fonts', 'PressStart2P-Regular.ttf'), 14)

        except (pygame.error, FileNotFoundError):

            self.font = pygame.font.Font(None, 36)

            self.small\_font = pygame.font.Font(None, 22)

    def \_create\_mud\_image(self, floor\_image):

        """יוצר תמונת בוץ מתוך תמונת הרצפה"""

        mud\_image = floor\_image.copy()

*# הוספת צבע חום לבוץ*

        mud\_surface = pygame.Surface(mud\_image.get\_size(), pygame.SRCALPHA)

        mud\_surface.fill((139, 69, 19, 100))  *# צבע חום עם שקיפות*

        mud\_image.blit(mud\_surface, (0, 0))

        return mud\_image

    def \_load\_images(self):

        image\_files = {

            'P': 'princess.png', 'D': 'dragon.png', 'W': 'tree\_wall.png',

            'G': 'castle\_gate.png', 'K': 'golden\_key.png', 'I': 'ice\_tile.png',

            'H': 'heart.png',  *# לב נוסף*

            '.': 'floor.png', 'HEART': 'heart.png', 'KEY\_ICON': 'golden\_key.png'

        }

        loaded\_images = {}

*# שימוש בנתיב מוחלט כדי להבטיח טעינה נכונה של התמונות*

        path = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), 'assets', 'images')

        for key, filename in image\_files.items():

            try:

                img = pygame.image.load(os.path.join(path, filename)).convert\_alpha()

                size = (self.tile\_size // 2, self.tile\_size // 2) if key in ['HEART', 'KEY\_ICON'] else (self.tile\_size, self.tile\_size)

                loaded\_images[key] = pygame.transform.scale(img, size)

            except pygame.error as e:

                print(f"Error loading image {filename}: {e}")

                loaded\_images[key] = pygame.Surface((self.tile\_size, self.tile\_size))

*# יצירת תמונת בוץ מתוך תמונת הרצפה*

        if '.' in loaded\_images:

            loaded\_images['B'] = self.\_create\_mud\_image(loaded\_images['.'])

        return loaded\_images

    def draw\_grid(self, screen, grid):

        for y, row in enumerate(grid):

            for x, tile in enumerate(row):

                screen.blit(self.images['.'], (x \* self.tile\_size, y \* self.tile\_size))

                if tile != '.':

                    screen.blit(self.images.get(tile), (x \* self.tile\_size, y \* self.tile\_size))

    def draw\_hud(self, screen, hearts, has\_key, level, turn\_count, grace\_turns, dragon\_state="waiting"):

        hud\_rect = pygame.Rect(0, screen.get\_height() - 60, screen.get\_width(), 60)

        pygame.draw.rect(screen, (10, 20, 30), hud\_rect)

        pygame.draw.line(screen, (200, 200, 200), (0, screen.get\_height() - 60), (screen.get\_width(), screen.get\_height() - 60), 2)

*# הצגת לבבות*

        for i in range(hearts):

            screen.blit(self.images['HEART'], (10 + i \* (self.tile\_size // 2 + 5), screen.get\_height() - 45))

*# הצגת מפתח*

        key\_icon = self.images['KEY\_ICON']

        key\_icon.set\_alpha(255 if has\_key else 80)

        screen.blit(key\_icon, (screen.get\_width() - 50, screen.get\_height() - 45))

*# הצגת מספר השלב*

        level\_text = self.font.render(f"Level: {level}", True, (255, 215, 0))

        screen.blit(level\_text, level\_text.get\_rect(centerx=screen.get\_width() // 2, y=screen.get\_height() - 50))

*# הצגת מצב הדרקון*

        if turn\_count < grace\_turns:

            info\_text = self.small\_font.render(f"Grace: {grace\_turns - turn\_count}", True, (0, 255, 150))

        else:

            info\_text = self.small\_font.render("Dragon is hunting!", True, (255, 50, 50))

        screen.blit(info\_text, (120, screen.get\_height() - 42))

    def draw\_message(self, screen, text):

        if not text: return

        message\_surface = self.font.render(text, True, (255, 255, 255))

        rect = message\_surface.get\_rect(center=(screen.get\_width() / 2, screen.get\_height() / 2 - 100))

        bg\_rect = rect.inflate(20, 20)

        pygame.draw.rect(screen, (50, 50, 80, 200), bg\_rect, border\_radius=10)

        screen.blit(message\_surface, rect)

    def draw\_game\_over(self, screen, won):

        overlay = pygame.Surface(screen.get\_size(), pygame.SRCALPHA)

        overlay.fill((0, 0, 0, 180))

        screen.blit(overlay, (0, 0))

        text, color = ("You Won!", (0, 255, 0)) if won else ("Game Over", (255, 0, 0))

        title = self.font.render(text, True, color)

        subtitle = self.small\_font.render("Press 'R' to Restart", True, (255, 255, 255))

        screen.blit(title, title.get\_rect(center=(screen.get\_width() / 2, screen.get\_height() / 2 - 30)))

        screen.blit(subtitle, subtitle.get\_rect(center=(screen.get\_width() / 2, screen.get\_height() / 2 + 30)))

