**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
**(БГТУ им. В.Г. Шухова)**


ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

# Лабораторная работа №15

по дисциплине: ООП
тема: «Знакомство с библиотеками языка Python. PyGame.»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

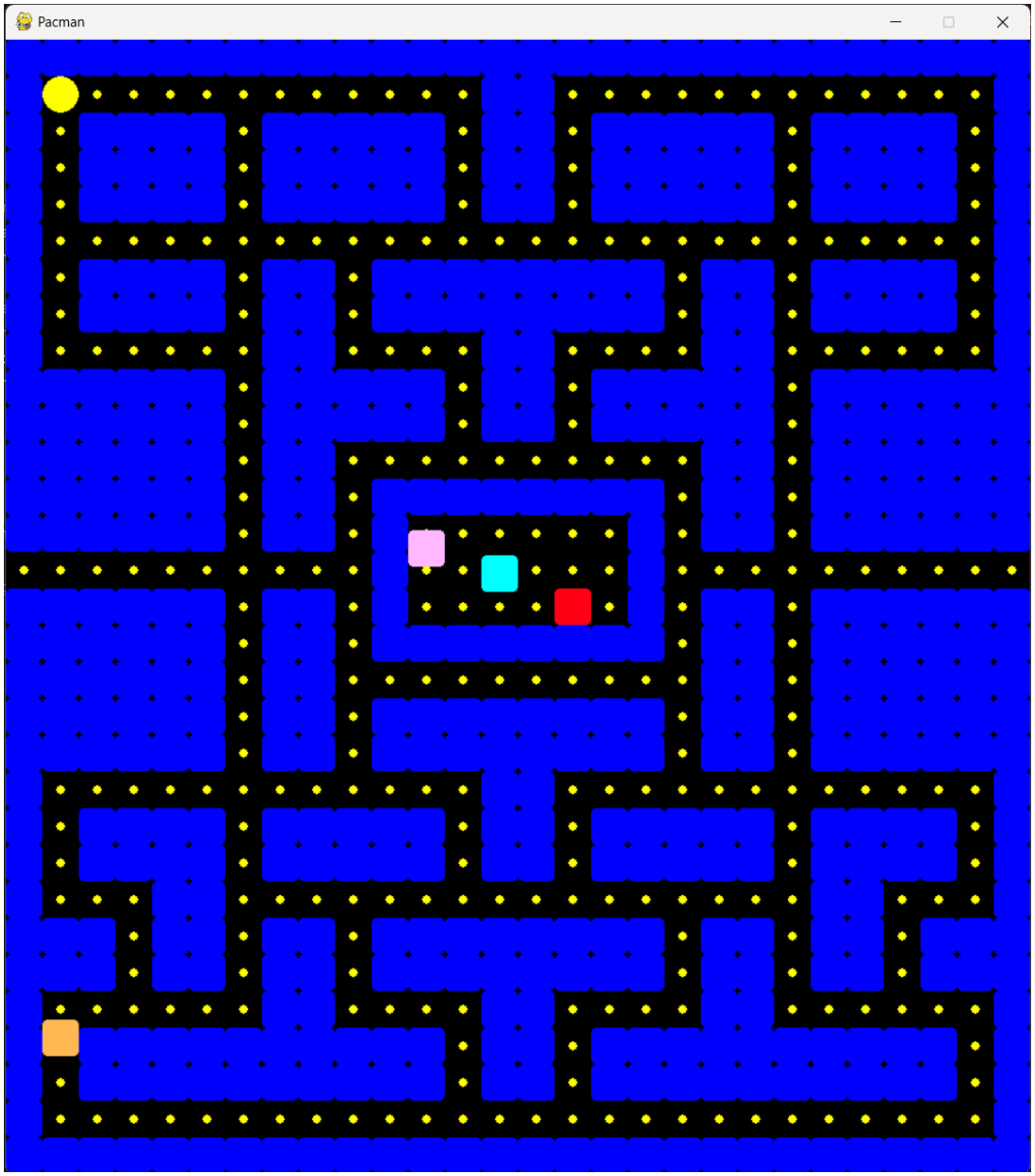
Проверили:
пр. Черников Сергей Викторович

Белгород 2024 г.

**Лабораторная работа №15**
«Знакомство с библиотеками языка Python. PyGame.»
Вариант 10

**Цель работы:** приобретение практических навыков создания приложений на языке Python, быстрая разработка 2d игр.

PacMan.




```
import pygame
import numpy as np
import tcod
import random
from enum import Enum


class Direction(Enum):
    LEFT = 0
    UP = 1
    RIGHT = 2
    DOWN = 3,
    NONE = 4

def translate_screen_to_maze(in_coords, in_size=32):
    return int(in_coords[0] / in_size), int(in_coords[1] / in_size)

def translate_maze_to_screen(in_coords, in_size=32):
    return in_coords[0] * in_size, in_coords[1] * in_size
```

```
class GameObject:
    def __init__(self, in_surface, x, y,
                 in_size: int, in_color=(255, 0, 0),
                 is_circle: bool = False):
        self._size = in_size
        self._renderer: GameRenderer = in_surface
        self._surface = in_surface._screen
        self.y = y
        self.x = x
        self._color = in_color
        self._circle = is_circle
        self._shape = pygame.Rect(self.x, self.y, in_size, in_size)

    def draw(self):
        if self._circle:
            pygame.draw.circle(self._surface,
                               self._color,
                               (self.x, self.y),
                               self._size)
        else:
            rect_object = pygame.Rect(self.x, self.y, self._size, self._size)
            pygame.draw.rect(self._surface,
                             self._color,
                             rect_object,
                             border_radius=4)

    def tick(self):
        pass

    def get_shape(self):
        return self._shape

    def set_position(self, in_x, in_y):
        self.x = in_x
        self.y = in_y

    def get_position(self):
        return (self.x, self.y)


class Wall(GameObject):
    def __init__(self, in_surface, x, y, in_size: int, in_color=(0, 0, 255)):
        super().__init__(in_surface, x * in_size, y * in_size, in_size, in_color)


class GameRenderer:
    def __init__(self, in_width: int, in_height: int):
        pygame.init()
        self._width = in_width
        self._height = in_height
        self._screen = pygame.display.set_mode((in_width, in_height))
        pygame.display.set_caption('Pacman')
        self._clock = pygame.time.Clock()
        self._done = False
```

```
        self._game_objects = []
        self._walls = []
        self._cookies = []
        self._hero: Hero = None
        self.gameover = False
        self.gameoversprite = pygame.transform.scale(pygame.image.load("gameover.png").convert_alpha(), (300, 200))

    def tick(self, in_fps: int):
        black = (0, 0, 0)
        while not self._done:
            for game_object in self._game_objects:
                if not self.gameover:
                    game_object.tick()
                game_object.draw()

            if self.gameover:
                self._screen.blit(self.gameoversprite, (self._screen.get_size()[0] / 2 -
                ↪  self.gameoversprite.get_size()[0] / 2,
                                                        self._screen.get_size()[1] / 2 -
                                                        ↪  self.gameoversprite.get_size()[1] / 2))

            pygame.display.flip()
            self._clock.tick(in_fps)
            self._screen.fill(black)
            self._handle_events()
        print("Game over")

    def add_game_object(self, obj: GameObject):
        self._game_objects.append(obj)

    def add_cookie(self, obj: GameObject):
        self._game_objects.append(obj)
        self._cookies.append(obj)

    def add_wall(self, obj: Wall):
        self.add_game_object(obj)
        self._walls.append(obj)

    def get_walls(self):
        return self._walls

    def get_cookies(self):
        return self._cookies

    def get_game_objects(self):
        return self._game_objects

    def add_hero(self, in_hero):
        self.add_game_object(in_hero)
        self._hero = in_hero

    def _handle_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
```

```
                self._done = True

        pressed = pygame.key.get_pressed()
        if pressed[pygame.K_UP]:
            self._hero.set_direction(Direction.UP)
        elif pressed[pygame.K_LEFT]:
            self._hero.set_direction(Direction.LEFT)
        elif pressed[pygame.K_DOWN]:
            self._hero.set_direction(Direction.DOWN)
        elif pressed[pygame.K_RIGHT]:
            self._hero.set_direction(Direction.RIGHT)


class MovableObject(GameObject):
    def __init__(self, in_surface, x, y, in_size: int, in_color=(255, 0, 0), is_circle: bool = False):
        super().__init__(in_surface, x, y, in_size, in_color, is_circle)
        self.current_direction = Direction.NONE
        self.direction_buffer = Direction.NONE
        self.last_working_direction = Direction.NONE
        self.location_queue = []
        self.next_target = None

    def get_next_location(self):
        return None if len(self.location_queue) == 0 else self.location_queue.pop(0)

    def set_direction(self, in_direction):
        self.current_direction = in_direction
        self.direction_buffer = in_direction

    def collides_with_wall(self, in_position):
        collision_rect = pygame.Rect(in_position[0], in_position[1], self._size, self._size)
        collides = False
        walls = self._renderer.get_walls()
        for wall in walls:
            collides = collision_rect.colliderect(wall.get_shape())
            if collides: break
        return collides

    def check_collision_in_direction(self, in_direction: Direction):
        desired_position = (0, 0)
        if in_direction == Direction.NONE: return False, desired_position
        if in_direction == Direction.UP:
            desired_position = (self.x, self.y - 1)
        elif in_direction == Direction.DOWN:
            desired_position = (self.x, self.y + 1)
        elif in_direction == Direction.LEFT:
            desired_position = (self.x - 1, self.y)
        elif in_direction == Direction.RIGHT:
            desired_position = (self.x + 1, self.y)

        return self.collides_with_wall(desired_position), desired_position

    def automatic_move(self, in_direction: Direction):
        pass
```

```
    def tick(self):
        self.reached_target()
        self.automatic_move(self.current_direction)

    def reached_target(self):
        pass


class Hero(MovableObject):
    def __init__(self, in_surface, x, y, in_size: int):
        super().__init__(in_surface, x, y, in_size, (255, 255, 0), False)
        self.last_non_colliding_position = (0, 0)

    def tick(self):
        # TELEPORT
        if self.x < 0:
            self.x = self._renderer._width

        if self.x > self._renderer._width:
            self.x = 0

        self.last_non_colliding_position = self.get_position()

        if self.check_collision_in_direction(self.direction_buffer)[0]:
            self.automatic_move(self.current_direction)
        else:
            self.automatic_move(self.direction_buffer)
            self.current_direction = self.direction_buffer

        if self.collides_with_wall((self.x, self.y)):
            self.set_position(self.last_non_colliding_position[0], self.last_non_colliding_position[1])

        self.handle_cookie_pickup()

        collision_rect = pygame.Rect(self.x, self.y, self._size, self._size)

        for ghost in self._renderer.get_game_objects():
            if not isinstance(ghost, Ghost):
                continue

            collides = collision_rect.colliderect(pygame.Rect(ghost.x, ghost.y, ghost._size, ghost._size))
            if collides:
                self._renderer.gameover = True

    def automatic_move(self, in_direction: Direction):
        collision_result = self.check_collision_in_direction(in_direction)

        desired_position_collides = collision_result[0]
        if not desired_position_collides:
            self.last_working_direction = self.current_direction
            desired_position = collision_result[1]
            self.set_position(desired_position[0], desired_position[1])
        else:
```

```
            self.current_direction = self.last_working_direction

    def handle_cookie_pickup(self):
        collision_rect = pygame.Rect(self.x, self.y, self._size, self._size)
        cookies = self._renderer.get_cookies()
        game_objects = self._renderer.get_game_objects()
        for cookie in cookies:
            collides = collision_rect.colliderect(cookie.get_shape())
            if collides and cookie in game_objects:
                game_objects.remove(cookie)

    def draw(self):
        half_size = self._size / 2
        pygame.draw.circle(self._surface, self._color, (self.x + half_size, self.y + half_size), half_size)


class Ghost(MovableObject):
    def __init__(self, in_surface, x, y, in_size: int, in_game_controller, in_color=(255, 0, 0)):
        super().__init__(in_surface, x, y, in_size, in_color, False)
        self.game_controller = in_game_controller

    def reached_target(self):
        if (self.x, self.y) == self.next_target:
            self.next_target = self.get_next_location()
        self.current_direction = self.calculate_direction_to_next_target()

    def set_new_path(self, in_path):
        for item in in_path:
            self.location_queue.append(item)
        self.next_target = self.get_next_location()

    def calculate_direction_to_next_target(self) -> Direction:
        if self.next_target is None:
            self.game_controller.request_new_random_path(self)
            return Direction.NONE
        diff_x = self.next_target[0] - self.x
        diff_y = self.next_target[1] - self.y
        if diff_x == 0:
            return Direction.DOWN if diff_y > 0 else Direction.UP
        if diff_y == 0:
            return Direction.LEFT if diff_x < 0 else Direction.RIGHT
        self.game_controller.request_new_random_path(self)
        return Direction.NONE

    def automatic_move(self, in_direction: Direction):
        if in_direction == Direction.UP:
            self.set_position(self.x, self.y - 1)
        elif in_direction == Direction.DOWN:
            self.set_position(self.x, self.y + 1)
        elif in_direction == Direction.LEFT:
            self.set_position(self.x - 1, self.y)
        elif in_direction == Direction.RIGHT:
            self.set_position(self.x + 1, self.y)
```

```
class Cookie(GameObject):
    def __init__(self, in_surface, x, y):
        super().__init__(in_surface, x, y, 4, (255, 255, 0), True)


class Pathfinder:
    def __init__(self, in_arr):
        cost = np.array(in_arr, dtype=np.bool_).tolist()
        self.pf = tcod.path.AStar(cost=cost, diagonal=0)

    def get_path(self, from_x, from_y, to_x, to_y) -> object:
        res = self.pf.get_path(from_x, from_y, to_x, to_y)
        return [(sub[1], sub[0]) for sub in res]


class PacmanGameController:
    def __init__(self):
        self.ascii_maze = [
            "XXXXXXXXXXXXXXXXXXXXXXXXXXXX",
            "XP           XX            X",
            "X XXXX XXXXX XX XXXXX XXXX X",
            "X XXXX XXXXX XX XXXXX XXXX X",
            "X XXXX XXXXX XX XXXXX XXXX X",
            "X                          X",
            "X XXXX XX XXXXXXXX XX XXXX X",
            "X XXXX XX XXXXXXXX XX XXXX X",
            "X      XX    XX    XX      X",
            "XXXXXX XXXXX XX XXXXX XXXXXX",
            "XXXXXX XXXXX XX XXXXX XXXXXX",
            "XXXXXX XX          XX XXXXXX",
            "XXXXXX XX XXXXXXXX XX XXXXXX",
            "XXXXXX XX X   G  X XX XXXXXX",
            "          X G    X          ",
            "XXXXXX XX X   G  X XX XXXXXX",
            "XXXXXX XX XXXXXXXX XX XXXXXX",
            "XXXXXX XX          XX XXXXXX",
            "XXXXXX XX XXXXXXXX XX XXXXXX",
            "XXXXXX XX XXXXXXXX XX XXXXXX",
            "X            XX            X",
            "X XXXX XXXXX XX XXXXX XXXX X",
            "X XXXX XXXXX XX XXXXX XXXX X",
            "X   XX       G        XX   X",
            "XXX XX XX XXXXXXXX XX XX XXX",
            "XXX XX XX XXXXXXXX XX XX XXX",
            "X      XX    XX    XX      X",
            "X XXXXXXXXXX XX XXXXXXXXXX X",
            "X XXXXXXXXXX XX XXXXXXXXXX X",
            "X                          X",
            "XXXXXXXXXXXXXXXXXXXXXXXXXXXX",
        ]

        self.numpy_maze = []
        self.cookie_spaces = []
```

```
        self.reachable_spaces = []
        self.ghost_spawns = []
        self.ghost_colors = [
            (255, 184, 255),
            (255, 0, 20),
            (0, 255, 255),
            (255, 184, 82)
        ]
        self.size = (0, 0)
        self.convert_maze_to_numpy()
        self.p = Pathfinder(self.numpy_maze)

    def request_new_random_path(self, in_ghost: Ghost):
        random_space = random.choice(self.reachable_spaces)
        current_maze_coord = translate_screen_to_maze(in_ghost.get_position())

        path = self.p.get_path(current_maze_coord[1], current_maze_coord[0], random_space[1],
                               random_space[0])
        test_path = [translate_maze_to_screen(item) for item in path]
        in_ghost.set_new_path(test_path)

    def convert_maze_to_numpy(self):
        for x, row in enumerate(self.ascii_maze):
            self.size = (len(row), x + 1)
            binary_row = []
            for y, column in enumerate(row):
                if column == "G":
                    self.ghost_spawns.append((y, x))

                if column == "X":
                    binary_row.append(0)
                else:
                    binary_row.append(1)
                    self.cookie_spaces.append((y, x))
                    self.reachable_spaces.append((y, x))
            self.numpy_maze.append(binary_row)


if __name__ == "__main__":
    unified_size = 32
    pacman_game = PacmanGameController()
    size = pacman_game.size
    game_renderer = GameRenderer(size[0] * unified_size, size[1] * unified_size)

    for y, row in enumerate(pacman_game.numpy_maze):
        for x, column in enumerate(row):
            if column == 0:
                game_renderer.add_wall(Wall(game_renderer, x, y, unified_size))

    for cookie_space in pacman_game.cookie_spaces:
        translated = translate_maze_to_screen(cookie_space)
        cookie = Cookie(game_renderer, translated[0] + unified_size / 2, translated[1] + unified_size / 2)
        game_renderer.add_cookie(cookie)
```

```
    for i, ghost_spawn in enumerate(pacman_game.ghost_spawns):
        translated = translate_maze_to_screen(ghost_spawn)
        ghost = Ghost(game_renderer, translated[0], translated[1], unified_size, pacman_game,
                      pacman_game.ghost_colors[i % 4])
        game_renderer.add_game_object(ghost)

    pacman = Hero(game_renderer, unified_size, unified_size, unified_size)
    game_renderer.add_hero(pacman)
    game_renderer.tick(170)
```

Ссылка на репозиторий

**Вывод:** в ходе лабораторной работы приобрели практические навыки создания приложений на языке Python, быстрая разработка 2d игр.