**Постановка задачи**

Создать игру Pac-Man. Отобразить лабиринт на экране с использованием графической библиотеки Pygame. Реализовать перемещение игрока в лабиринте с помощью клавиш управления.  
  
**Код программы  
  
import pygame # import packages (install with "pip install pygame" in cmd)**

**import numpy as np**

**import tcod**

**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**

**def tick(self, in\_fps: int):**

**black = (0, 0, 0)**

**while not self.\_done:**

**for game\_object in self.\_game\_objects:**

**game\_object.tick()**

**game\_object.draw()**

**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\_wall(self, obj: Wall):**

**self.add\_game\_object(obj)**

**self.\_walls.append(obj)**

**def \_handle\_events(self):**

**pass # we'll implement this later**

**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**

**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 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",**

**# shortened for article, full ascii on my github**

**"XXXXXXXXXXXXXXXXXXXXXXXXXXXX",**

**]**

**self.numpy\_maze = []**

**self.cookie\_spaces = []**

**self.reachable\_spaces = []**

**self.ghost\_spawns = []**

**self.size = (0, 0)**

**self.convert\_maze\_to\_numpy()**

**#self.p = Pathfinder(self.numpy\_maze) # use later**

**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))**

**game\_renderer.tick(120)**

**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**

**# inPacmanGameController**

**self.ghost\_colors = [**

**(255, 184, 255),**

**(255, 0, 20),**

**(0, 255, 255),**

**(255, 184, 82)**

**]**

**# in main**

**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)**

**# General functions for coordinate conversion, place at the beginning of the code**

**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 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]**

**# draw path - optional**

**red = (255, 0, 0)**

**green = (0, 255, 0)**

**\_from = (1, 1)**

**\_to = (24, 24)**

**path\_array = pacman\_game.p.get\_path(\_from[1], \_from[0], \_to[1], \_to[0])**

**print(path\_array)**

**# [(1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5), (6, 6), (6, 7) ...**

**white = (255, 255, 255)**

**for path in path\_array:**

**game\_renderer.add\_game\_object(Wall(game\_renderer, path[0], path[1], unified\_size, white))**

**from\_translated = translate\_maze\_to\_screen(\_from)**

**game\_renderer.add\_game\_object(**

**GameObject(game\_renderer, from\_translated[0], from\_translated[1], unified\_size, red))**

**to\_translated = translate\_maze\_to\_screen(\_to)**

**game\_renderer.add\_game\_object(**

**GameObject(game\_renderer, to\_translated[0], to\_translated[1], unified\_size, green))**

**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 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 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()**

**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)**

**Анализ результатов**