

Разработка игры «Battle City» на языке Python с помощью нейросети

Ёлгин Роман Николаевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной статье рассматривается процесс разработки 2D-игры жанра аркадный танковый шутер по мотивам классической игры «Battle City» на языке Python с использованием генеративной нейросети в роли интеллектуального помощника программиста. Выполнен обзор актуальных исследований в области генерации программного кода и игровых объектов нейросетями, выбраны инструменты разработки (Python, библиотека Pygame и облачный сервис нейросети), определена архитектура игры, реализованы базовые игровые механики с привлечением нейросети для генерации и рефакторинга кода. Обсуждаются преимущества и ограничения использования нейросетей при создании небольших игр.

Ключевые слова: разработка игры, Python, Pygame, нейросеть, генерация кода, Battle City.

Development of “Battle City” game in python using a neural network

Yolgin Roman Nikolaevich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article discusses the process of developing a 2D arcade tank shooter game based on the classic Battle City game using Python and a generative neural network as an intelligent programmer assistant. The article provides an overview of current research in the field of neural network-based software code and game object generation, selects development tools (Python, the Pygame library, and a neural network cloud service), defines the game architecture, and implements basic game mechanics using a neural network for code generation and refactoring. The article also discusses the advantages and limitations of using neural networks for creating small games.

Keywords: game development, Python, Pygame, neural network, code generation, Battle City.

1 Введение

1.1 Актуальность исследования

Разработка простых 2D-игр широко применяется в обучении программированию, так как позволяет наглядно познакомиться с основами

алгоритмизации, объектно-ориентированного подхода и обработки событий в режиме реального времени. Язык Python благодаря простому синтаксису и большому количеству библиотек стал одним из стандартов для учебных проектов и прототипирования игр.

Параллельно стремительно развиваются генеративные нейросети, в частности большие языковые модели, способные создавать и дополнять программный код на различных языках. Такие системы уже интегрируются в среды разработки в виде интеллектуальных помощников и автодополнения кода. Для начинающих программистов это открывает возможность получать готовые фрагменты программы, рекомендации по архитектуре и пояснения к непонятным конструкциям.

Однако использование нейросетей в учебных игровых проектах пока недостаточно изучено: не до конца понятно, как именно они влияют на структуру кода, скорость разработки и формирование профессиональных навыков. Поэтому исследование процесса создания игры «Battle City» на Python с активным использованием нейросети в качестве помощника представляется актуальным и с практической, и с методической точек зрения.

1.2 Обзор исследований

Модель StarCoder, описанная R. Li представляет собой крупную языковую модель, обученную на репозиториях открытого программного кода и ориентированную на практическую поддержку разработчиков [1]. M. Chen представил модель Codex, показавшую высокое качество генерации программного кода на популярных языках, включая Python [2]. Исследование M. Vaithilingam посвящено оценке удобства использования инструментов генерации кода и анализу того, как они влияют на рабочие процессы программистов [3]. S. Barke в работе Grounded Copilot детально изучают стратегии взаимодействия программистов с моделями автогенерации кода и факторы доверия к их подсказкам [4]. В статье E. Nijkamp описана модель CodeGen, поддерживающая диалоговый многошаговый синтез программ и работу с несколькими языками программирования [5]. Технический отчёт по GPT-4 демонстрирует, что универсальная языковая модель общего назначения способна решать широкий спектр задач, включая написание и объяснение кода и обработку естественного языка [6]. Работа по MarioGPT показывает, как языковая модель может генерировать уровни платформера «Super Mario Bros.» по текстовому описанию, что иллюстрирует применение нейросетей для автоматизации геймдизайна [7].

1.3 Цель исследования

Цель исследования — разработать игру «Battle City» на языке Python с использованием нейросети в качестве интеллектуального помощника и оценить, как такое взаимодействие влияет на архитектуру проекта, скорость разработки и качество получившегося решения.

2 Материалы и методы

Разработка игры выполнялась на языке Python версии 3.x с использованием следующих инструментов:

- стандартная библиотека Python и официальная документация;
- библиотека Pygame для работы с 2D-графикой, обработкой событий и звуком;
- среда разработки PyCharm с поддержкой отладки;
- облачный сервис генеративной нейросети общего назначения (модель класса GPT-5 или аналогичная), доступный через веб-интерфейс или API, [lmarena.ai](#).

3 Результаты и обсуждение

Для разработки игры «Battle City» необходимо установить Python, а также среду разработки PyCharm. Зарегистрироваться в онлайн-сервисе [lmarena.ai](#), использовать в качестве основы нейросетевой модели GPT-5, позволяющем генерировать и дорабатывать программный код по текстовому описанию задачи.

Данный сервис предлагает простой и интуитивно понятный интерфейс (рис.1).

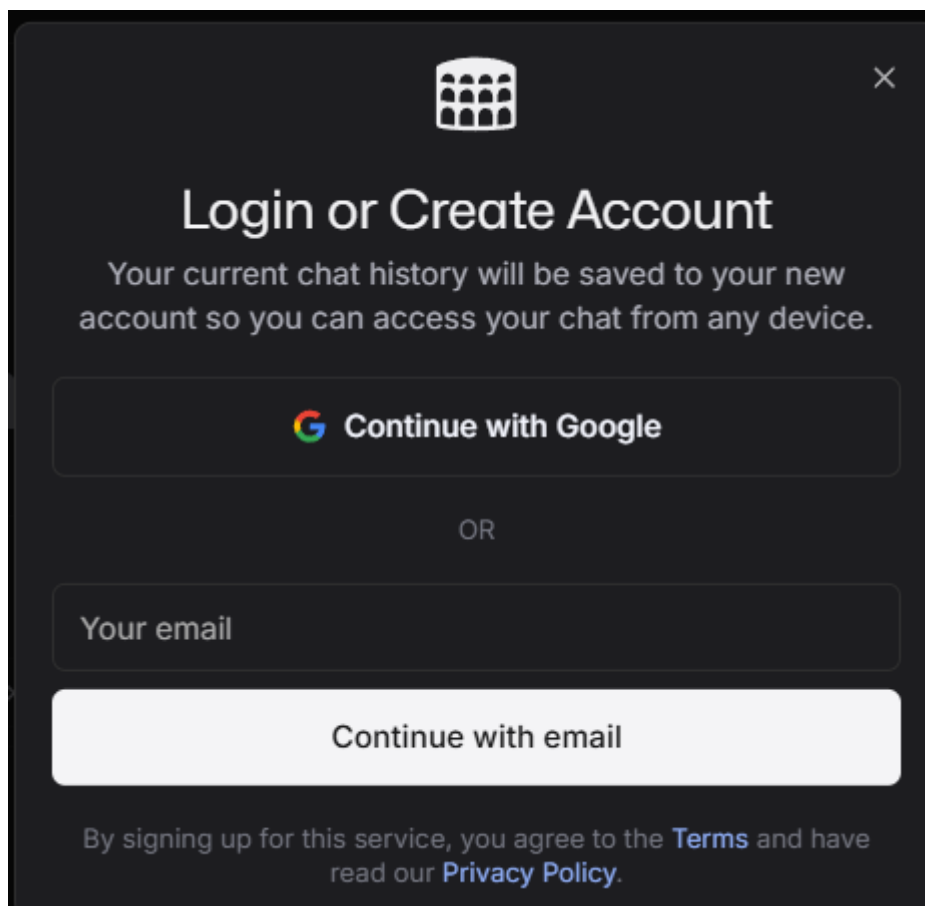


Рис. 1. Регистрация

После регистрации будет доступен ввод необходимого запроса, а также выбор нейросетевой модели (рис. 2).

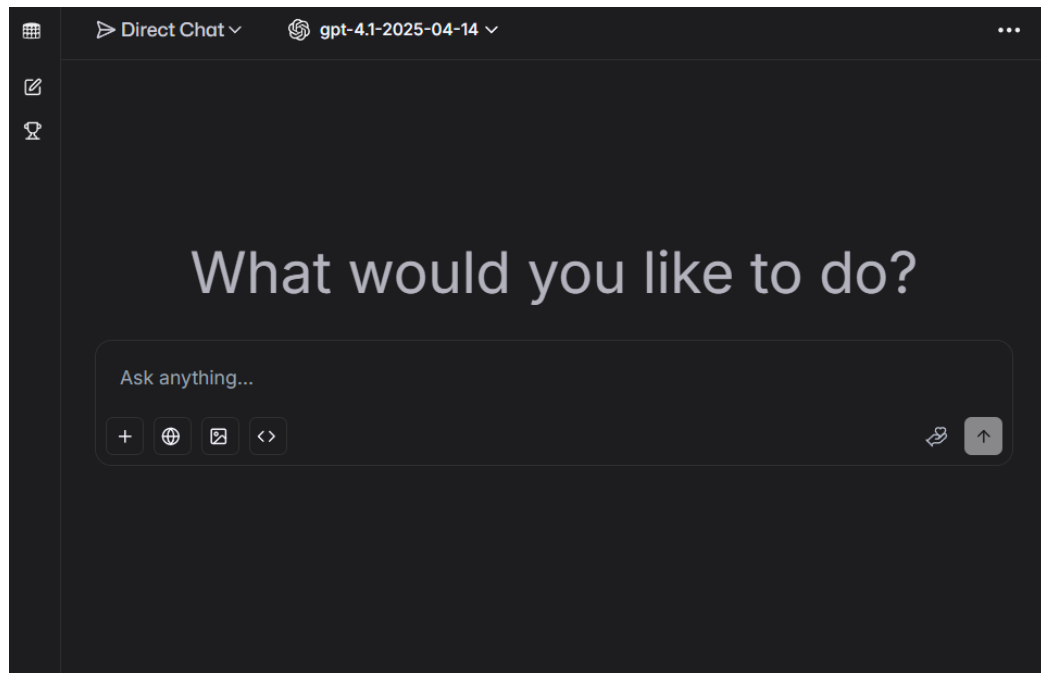


Рис. 2. Начало работы

На первом этапе был сформулирован текстовый запрос (промпт), в котором задавались основные требования к будущей игре: тип графики, управление, структура уровней, типы врагов и бонусы. Пример исходного промпта приведён ниже:

Разработай на языке Python с использованием библиотеки Pygame 2D-игру в стиле

«Battle City».

Требования к игре:

1. Игровое поле:

- вид сверху;*
- сетка из тайлов 32×32 , размер поля 13×13 клеток;*
- карта уровня задаётся списком строк, где символы обозначают*

объекты:

- '#' – разрушаемый кирпичный блок,*
- 'S' – неразрушаемая сталь,*
- 'W' – вода (танки не проезжают, пули пролетают),*
- 'G' – трава (декор, рисуется поверх танков),*
- 'B' – база игрока,*
- 'P' – позиция танка игрока,*
- 'E' – обычный враг,*
- 'F' – быстрый враг,*
- 'A' – бронированный враг,*
- '.' – пустая клетка.*

2. Танк игрока:

- движение по четырём направлениям (влево, вправо, вверх, вниз) стрелками клавиатуры, без диагонального движения;
- размер танка меньше тайла, чтобы он свободно проходил по коридорам;
- автоматическое «прилипание» к проходам: при движении по X танк выравнивается по центру клетки по Y и наоборот;
- стрельба пробелом, с задержкой между выстрелами;
- пули вылетают от края танка в направлении движения.

3. Противники:

- несколько типов врагов:
 - E – обычный (средняя скорость, 1 жизнь);
 - F – быстрый (повышенная скорость, 1 жизнь);
 - A – бронированный (медленный, несколько жизней);
- враги двигаются по карте, периодически меняют направление и иногда стреляют;
- при попадании пули игрока во врага он получает урон и при обнулении здоровья уничтожается.

4. Объекты окружения и столкновения:

- кирпичи разрушаются от выстрела;
- сталь неразрушаема;
- вода блокирует движение танков, но не останавливает пули;
- при попадании по базе игра завершается;
- реализовать столкновение пуль друг с другом: если пуля игрока встречает пулю врага, обе уничтожаются с небольшим эффектом взрыва.

5. Игровая логика:

- счёт очков (за уничтожение врагов и кирпичных блоков);
- ограниченное количество жизней игрока; при потере всех жизней – Game Over;
- несколько уровней (список карт); переход на следующий уровень после уничтожения всех врагов;
- главное меню (начать игру, выбор уровня, выход) и экран Game Over.

6. Бонусы:

- случайное выпадение бонуса при уничтожении врага;
- типы бонусов:
 - shield* – временный щит (неуязвимость);
 - power* – усиленная пушка (быстрая перезарядка и скорость пули);
 - speed* – ускорение танка;
- отображение активных бонусов в интерфейсе.

7. Звук и музыка:

- использование *pygame.mixer*;

- звуки выстрела, взрыва и подбора бонуса;
- фоновая музыка в цикле; при отсутствии файла – игра должна продолжать работать без звука.

Сгенерируй полный исходный код игры в одном файле `main.py`, с комментариями

к ключевым частям программы (инициализация, классы танка, пули, врагов, карта уровня, главное меню, бонусы).

Игровое окно создаётся функцией `main()`, которая инициализирует Pygame, задаёт размеры экрана и запускает главный цикл обработки событий:

```
import pygame

TILE = 32
MAP_W, MAP_H = 13, 13
WIDTH, HEIGHT = TILE * MAP_W, TILE * MAP_H
FPS = 60

def main():
    pygame.init()
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption("Battle City на Python")
    clock = pygame.time.Clock()

    running = True
    while running:
        dt = clock.tick(FPS)

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        # обновление объектов и отрисовка
        pygame.display.flip()

    pygame.quit()
```

Полученный код был перенесён в среду PyCharm и запущен для тестирования (Рис.3):

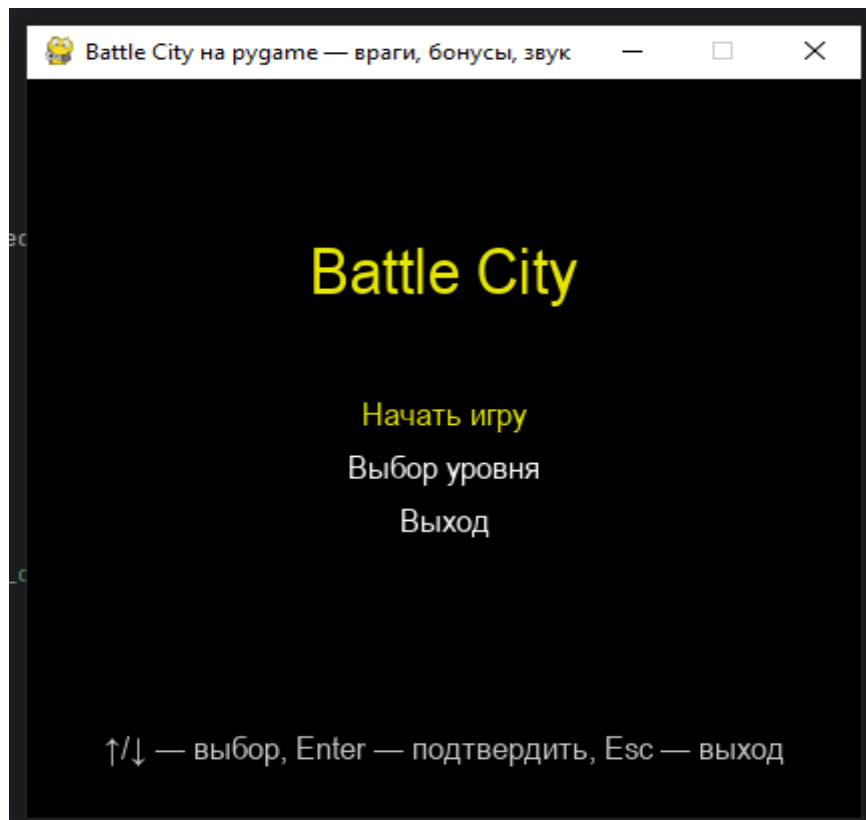


Рис. 3 Стартовое окно игры

Карта уровня описывается в виде массива строк, где каждый символ соответствует типу клетки (кирпич, сталь, вода, база, танк игрока, враг и т.п.). Нейросеть по текстовому описанию сгенерировала структуру данных и базовый класс `Block` для отрисовки тайлов:

```
LEVEL_1 = [
    "#####",
    "#P.....E#",
    "#.###.###...#",
    "#.....#",
    "#.###.###...#",
    "#.....B.....#",
    "#.###.###...#",
    "#E.....F#",
    "#####",
]

class Block(pygame.sprite.Sprite):
    def __init__(self, x, y, ch):
        super().__init__(all_sprites, walls)
        self.base = (ch == 'B')
        self.destructible = (ch == '#')
        self.blocks_tank = ch in ['#', 'S', 'W', 'B']
        self.blocks_bullet = ch in ['#', 'S', 'B']
```

```

if ch == '#':
    self.image = make_brick_surface()
elif ch == 'S':
    self.image = make_steel_surface()
elif ch == 'W':
    self.image = make_water_surface()
elif ch == 'B':
    self.image = make_base_surface()

self.rect = self.image.get_rect()
self.rect.topleft = (x * TILE, y * TILE)

```

Результат выполнения кода (Рис. 4):

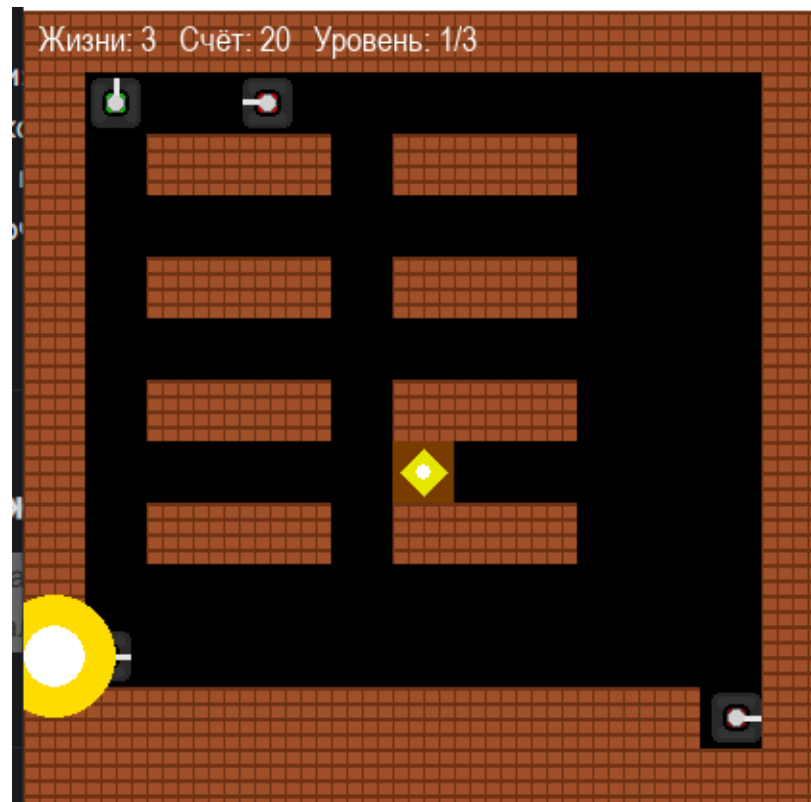


Рис. 4 Карта первого уровня

Управление танком игрока реализовано в виде отдельного класса PlayerTank, наследующего базовый класс Tank. Движение выполняется по сетке с учётом коллизий со стенами; при одновременном нажатии двух клавиш движение по диагонали блокируется, что приближает поведение к оригинальной игре:

```

class Tank(pygame.sprite.Sprite):
    def __init__(self, x, y, color):
        super().__init__(all_sprites, tanks)

```

```

self.base_image = create_tank_base_image(color)
self.image = self.base_image.copy()
self.rect = self.image.get_rect()
self.rect.center = (x * TILE + TILE // 2, y * TILE + TILE // 2)
self.pos = pygame.math.Vector2(self.rect.topleft)

```

```

self.direction = pygame.math.Vector2(0, -1)
self.speed = 2
self.shoot_cooldown = 400
self.last_shot = 0

```

```

def move(self, dx, dy, walls_group):
    if dx == 0 and dy == 0:
        return
    self.direction.update(dx, dy)
    # далее выполняется поосевое движение с проверкой столкновений

```

```

class PlayerTank(Tank):
    def __init__(self, x, y):
        super().__init__(x, y, color=(0, 200, 0))
        player_group.add(self)

    def update(self):
        keys = pygame.key.get_pressed()
        dx = dy = 0
        if keys[pygame.K_LEFT]:
            dx = -1
        elif keys[pygame.K_RIGHT]:
            dx = 1
        if keys[pygame.K_UP]:
            dy = -1
        elif keys[pygame.K_DOWN]:
            dy = 1
        if dx != 0 and dy != 0:
            dy = 0 # запрет диагонального движения
        self.move(dx, dy, walls)

```

Стрельба реализована с помощью класса Bullet. Снаряд создаётся в направлении текущего вектора движения танка, перемещается по прямой и обрабатывает столкновения со стенами, базой и танками. При использовании нейросети особое внимание уделялось корректной обработке коллизий и устранению ошибок, связанных с неправильным именем функции `spritecollideany`:

```

class Bullet(pygame.sprite.Sprite):

```

```
def __init__(self, tank):
    super().__init__(all_sprites, bullets)
    self.image = pygame.Surface((8, 8))
    self.image.fill((230, 230, 0))
    self.rect = self.image.get_rect()

    dir_vec = tank.direction.normalize()
    offset = dir_vec * (tank.rect.width // 2 + 2)
    self.pos = pygame.math.Vector2(tank.rect.center) + offset
    self.rect.center = self.pos

    self.vel = dir_vec * tank.bullet_speed
    self.owner = tank

def update(self):
    self.pos += self.vel
    self.rect.center = self.pos

    # столкновение с блоками
    hit_blocks = pygame.sprite.spritecollide(self, walls, False)
    for block in hit_blocks:
        if block.base:
            handle_base_destroyed(block)
        if block.destructible:
            handle_brick_destroyed(block)
            block.kill()
    self.kill()
    return
```

Этот код дает следующий результат (Рис. 5-6):

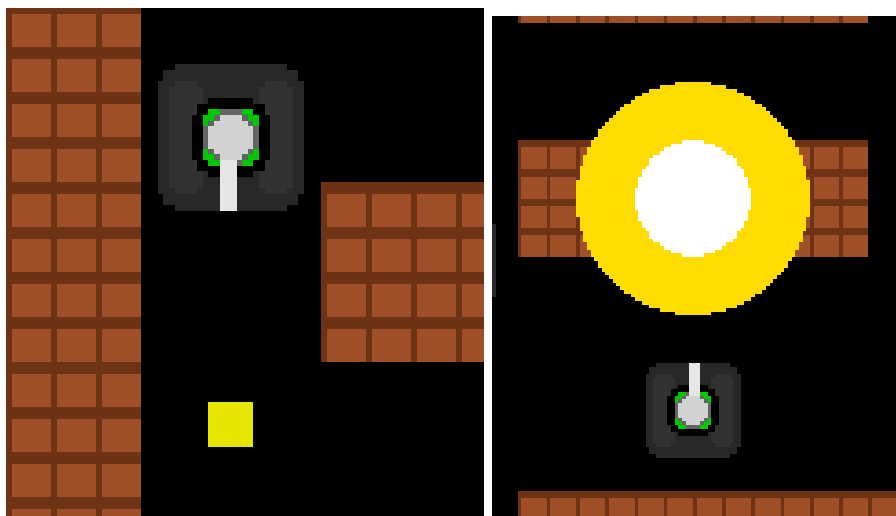


Рис 5-6. Стрельба и разрушение блоков

Для создания противников был разработан класс EnemyTank, параметры которого (скорость, количество «жизней», частота выстрелов) задаются в словаре ENEMY_STATS. Нейросеть по описанию задачи сгенерировала шаблон поведения: случайный выбор направления движения с периодической сменой, стрельба с заданной вероятностью, а также обработка получения урона:

```
ENEMY_STATS = {
    "normal": {"speed": 1.5, "hp": 1, "shoot_cooldown": 700},
    "fast": {"speed": 2.5, "hp": 1, "shoot_cooldown": 500},
    "armored": {"speed": 1.0, "hp": 3, "shoot_cooldown": 900},
}

class EnemyTank(Tank):
    def __init__(self, x, y, enemy_type="normal"):
        stats = ENEMY_STATS[enemy_type]
        super().__init__(x, y, color=(200, 0, 0))
        enemies.add(self)
        self.enemy_type = enemy_type
        self.speed = stats["speed"]
        self.hp = stats["hp"]
        self.shoot_cooldown = stats["shoot_cooldown"]

    def update(self):
        if game_state != "playing":
            return
        now = pygame.time.get_ticks()
        # смена направления и движение
        # случайная стрельба с небольшой вероятностью
```

Результат выполнения кода (Рис. 7):

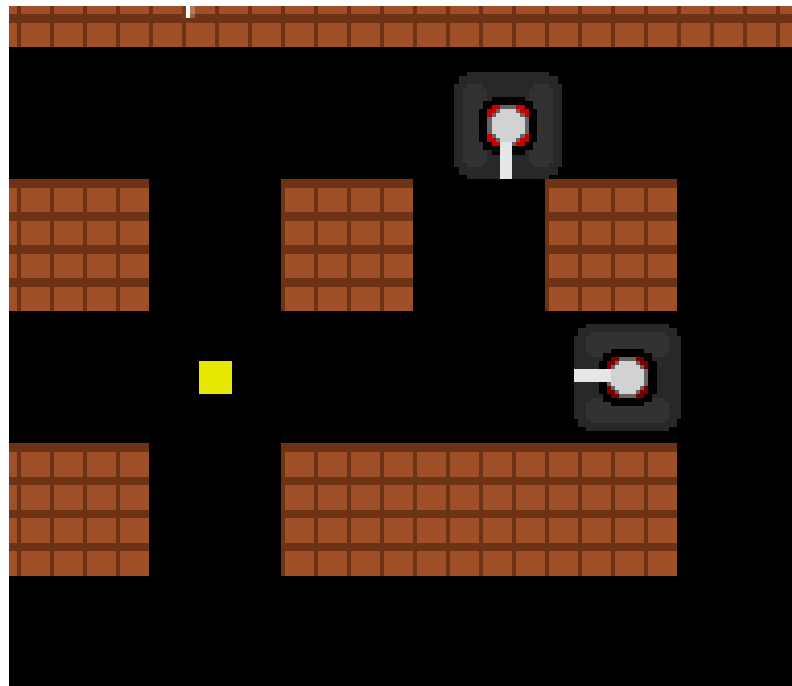


Рисунок 7. Движение и атака вражеских танков

Завершающим этапом стало создание главного меню, выбора уровня и экрана окончания игры. Для этого была введена простая система состояний `game_state` и отдельные функции отрисовки каждого экрана. Нейросеть сгенерировала шаблон обработчика событий и переключения состояний, который был адаптирован под структуру проекта:

```
game_state = "main_menu" # main_menu, level_select, playing, game_over
```

```
def draw_main_menu(screen):
    title = font_big.render("Battle City", True, (230, 230, 0))
    screen.blit(title, title.get_rect(center=(WIDTH // 2, HEIGHT // 2 - 100)))
    # отрисовка пунктов меню
```

```
def main():
    global game_state
    ...
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if game_state == "main_menu":
                    # выбор пункта меню, переход к игре или выбору уровня
                elif game_state == "game_over":
                    # перезапуск игры или возврат в меню
                ...
            if game_state == "main_menu":
```

```
draw_main_menu(screen)
elif game_state == "playing":
    draw_game(screen)
```

Этот код дает следующий результат (Рис. 8):

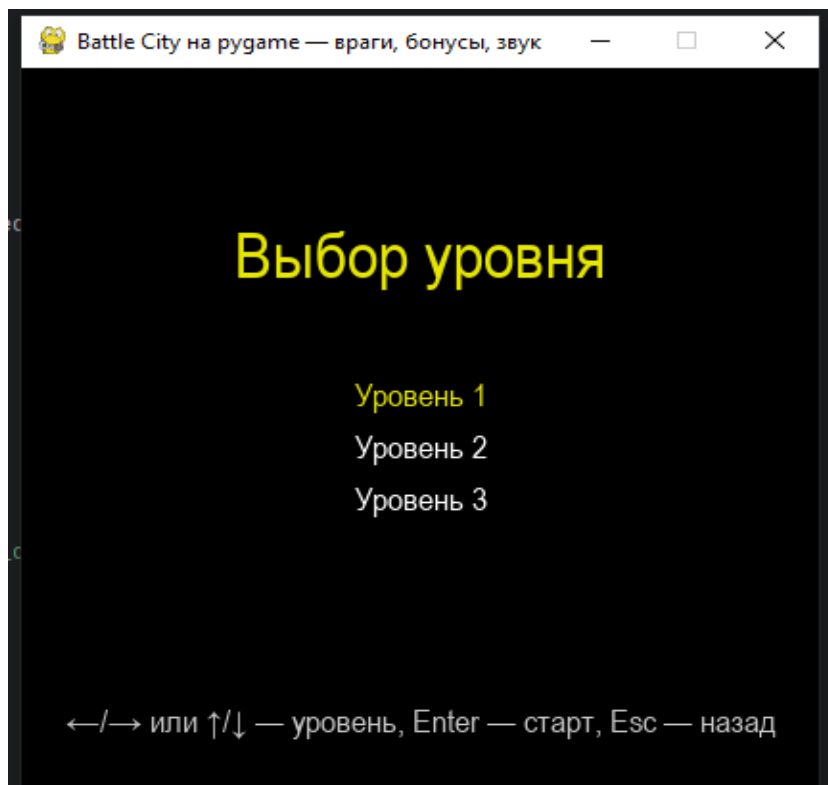


Рисунок 8. Главное меню и выбор уровня игры

4 Выводы

Разработка игры «Battle City» на языке Python показала, что современные генеративные нейросети могут эффективно использоваться как интеллектуальный помощник при создании учебных игровых проектов. Нейросеть облегчает подготовку черновых версий классов и функций, ускоряет поиск ошибок и позволяет получить дополнительные объяснения по незнакомым элементам языка и библиотеки Pygame.

В то же время использование нейросетей не отменяет необходимости владения базовыми навыками программирования: сгенерированный код требует проверки, адаптации и понимания логики работы. Для небольших игр на Python наиболее продуктивным оказывается комбинированный подход, когда общую архитектуру и ключевые решения разрабатывает человек, а нейросеть применяется для рутинных и повторяющихся фрагментов кода.

Полученные результаты могут быть использованы при организации учебных курсов по программированию и разработке игр, демонстрируя студентам как возможности, так и ограничения нейросетевых помощников.

Библиографический список

1. Li R., Allal L., Muennighoff N. et al. StarCoder: may the source be with you! // arXiv preprint arXiv:2305.06161. 2023.
2. Chen M., Tworek J., Jun H. et al. Evaluating large language models trained on code // arXiv preprint arXiv:2107.03374. 2021.
3. Vaithilingam P., Zhang T., Glassman E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models // Chi conference on human factors in computing systems extended abstracts. 2022. C. 1-7.
4. Barke S., James M. B., Polikarpova N. Grounded copilot: How programmers interact with code-generating models // Proceedings of the ACM on Programming Languages. 2023. T. 7. №. OOPSLA1. C. 85-111.
5. Nijkamp E., Pang B., Hayashi H. et al. CodeGen: an open large language model for code with multi-turn program synthesis // arXiv preprint arXiv:2203.13474. 2022.
6. OpenAI. GPT-4 technical report // arXiv preprint arXiv:2303.08774. 2023.
7. Sudhakaran S. et al. MarioGPT: Open-ended text2level generation through large language models // Advances in Neural Information Processing Systems. 2023. T. 36. C. 54213-54227.
8. Python Software Foundation. Python language reference. Version 3. URL: <https://docs.python.org/3/>.
9. Pygame Community. Pygame documentation. URL: <https://www.pygame.org/docs/>.