Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 4

з дисципліни «Технології та засоби розробки комп’ютерної графіки та мультимедіа»

Тема: «Переміщення зображення»

Виконала:
студентка групи ІС-34
Ященко Олександра.
Дата здачі 12.11.25
Захищено з балом _____

Перевірила:
ст. вик. кафедри ІСТ
Хмелюк Марина Сергіївна

Київ 2025

Завдання

Розробити програму з графічним інтерфейсом, яка реалізує:

- Малювання або підключення об'єктів (зображень)
- Безперервну траєкторію руху зображення за допомогою таймера
- Розташування об'єктів-перешкод та відбивання від них
- Управління переміщенням за допомогою клавіатури
- Управління переміщенням за допомогою маніпулятора миші
- Перемикання між об'єктами та режимами управління

Опис реалізації

Концепція програми

Програма реалізована у вигляді мінімалістичної гри "DODGE". Гравець керує білим колом, уникаючи червоних квадратів-ворогів, які автоматично рухаються по екрану та відбиваються від перешкод. Гра демонструє всі необхідні аспекти лабораторної роботи: автоматичний рух об'єктів, управління клавіатурою та мишею, систему зіткнень з перешкодами.

Структура програми

Програма побудована за об'єктно-орієнтованим принципом і складається з чотирьох основних класів:

1. Клас Player

```
class Player:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.size = 25

    def get_rect(self):
            return QRectF(self.x - self.size/2, self.y - self.size/2,
self.size, self.size)
```

Клас Player відповідає за представлення гравця на екрані. Він зберігає поточні координати центру об'єкта (x, y) та його розмір. Метод get_rect() повертає прямокутник QRectF, який використовується для перевірки зіткнень з іншими об'єктами.

2. Клас Enemy

```
class Enemy:
    def __init__(self, x, y, speed):
        self.x = x
        self.y = y
        self.size = 30
```

```
        self.vx = speed * random.choice([-1, 1])
        self.vy = speed * random.choice([-1, 1])

    def get_rect(self):
        return QRectF(self.x, self.y, self.size, self.size)
```

Клас Enemy представляє рухомих ворогів. На відміну від гравця, вороги мають вектор швидкості (vx, vy), який визначає напрямок та швидкість їх автоматичного руху. При ініціалізації швидкість випадково розподіляється між осями X та Y з випадковим напрямком (1 або -1), що створює різноманітні траєкторії руху.

3. Клас Obstacle

```
class Obstacle:
    def __init__(self, x, y, w, h):
        self.rect = QRectF(x, y, w, h)
```

Клас Obstacle — найпростіший з усіх, він просто зберігає прямокутну область, яка представляє перешкоду на ігровому полі.

4. Клас Game

Це головний клас програми, який наслідується від QWidget та керує всією логікою гри.

```
class Game(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("DODGE")
        self.setFixedSize(800, 600)
        self.setStyleSheet("background-color: black;")

        self.player = Player(400, 300)
        self.enemies = []
        self.obstacles = []
        self.score = 0
        self.game_over = False

        self.init_game()
```

У конструкторі встановлюються основні параметри вікна: заголовок, фіксований розмір 800×600 пікселів та чорний фон. Створюються списки для зберігання всіх ігрових об'єктів.

Технічні рішення

Система таймерів

Для реалізації безперервної анімації використовуються два таймери з бібліотеки PyQt5:

```
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_game)
        self.timer.start(16)
```

```
        self.score_timer = QTimer()
        self.score_timer.timeout.connect(self.increase_score)
        self.score_timer.start(100)
```

Перший таймер (timer) викликає метод update_game() кожні 16 мілісекунд, що забезпечує частоту оновлення приблизно 60 кадрів на секунду (1000ms / 16ms ≈ 62.5 FPS). Це стандартна частота для плавної анімації.

Другий таймер (score_timer) викликає метод increase_score() кожні 100 мілісекунд для збільшення рахунку гравця та періодичного додавання нових ворогів.

Ініціалізація ігрових об'єктів

```
    def init_game(self):
        self.obstacles = [
            Obstacle(150, 100, 80, 80),
            Obstacle(550, 400, 100, 60),
            Obstacle(300, 350, 60, 100)
        ]

        for i in range(3):
            x = random.randint(50, 700)
            y = random.randint(50, 500)
            while self.check_spawn_collision(x, y):
                x = random.randint(50, 700)
                y = random.randint(50, 500)
            speed = random.choice([2, 3, 4])
            self.enemies.append(Enemy(x, y, speed))
```

Метод init_game() створює три статичні перешкоди в різних частинах екрану та генерує три початкових вороги з випадковими позиціями. Важливо, що перед створенням ворога перевіряється, чи не перетинається його початкова позиція з гравцем або перешкодами за допомогою методу check_spawn_collision(). Це гарантує чесний старт гри.

Автоматичний рух ворогів

Ключовий метод update_game() відповідає за оновлення позицій всіх ворогів на кожному кадрі:

```
    def update_game(self):
        if self.game_over:
            return

        for enemy in self.enemies:
            new_x = enemy.x + enemy.vx
            new_y = enemy.y + enemy.vy

            if new_x <= 0 or new_x + enemy.size >= self.width():
                enemy.vx = -enemy.vx
                new_x = enemy.x + enemy.vx

            if new_y <= 0 or new_y + enemy.size >= self.height():
                enemy.vy = -enemy.vy
                new_y = enemy.y + enemy.vy
```

Принцип роботи:

1. Обчислюється нова позиція ворога на основі поточної позиції та вектора швидкості
2. Перевіряються межі екрану — якщо об'єкт виходить за ліву/праву межу, інвертується компонента vx
3. Якщо об'єкт виходить за верхню/нижню межу, інвертується компонента vy
4. Після інверсії швидкості позиція перераховується заново

Алгоритм визначення зіткнення з перешкодами

Для коректного відбивання від перешкод реалізовано складніший алгоритм, який визначає, з якої сторони відбулося зіткнення:

```
            test_enemy = Enemy(new_x, new_y, 0)
            test_enemy.size = enemy.size
            collision = False

            for obstacle in self.obstacles:
                if test_enemy.get_rect().intersects(obstacle.rect):
                    collision = True
                    enemy_rect = enemy.get_rect()
                    obs_rect = obstacle.rect

                    dx_left = abs(enemy_rect.right() - obs_rect.left())
                    dx_right = abs(enemy_rect.left() - obs_rect.right())
                    dy_top = abs(enemy_rect.bottom() - obs_rect.top())
                    dy_bottom = abs(enemy_rect.top() - obs_rect.bottom())

                    min_dist = min(dx_left, dx_right, dy_top, dy_bottom)

                    if min_dist in (dx_left, dx_right):
                        enemy.vx = -enemy.vx
                    else:
                        enemy.vy = -enemy.vy
                    break
```

Логіка алгоритму:

1. Створюється тестовий об'єкт з новими координатами для перевірки зіткнення
2. Якщо виявлено перетин з перешкодою методом intersects(), обчислюються чотири відстані:
    - dx_left — відстань між правим краєм ворога та лівим краєм перешкоди
    - dx_right — відстань між лівим краєм ворога та правим краєм перешкоди

- dy_top — відстань між нижнім краєм ворога та верхнім краєм перешкоди
- dy_bottom — відстань між верхнім краєм ворога та нижнім краєм перешкоди

3. Визначається мінімальна з чотирьох відстаней — вона вказує на сторону зіткнення
4. Якщо мінімум серед горизонтальних відстаней — інвертується vx (горизонтальне відбивання)
5. Якщо мінімум серед вертикальних відстаней — інвертується vy (вертикальне відбивання)

Цей підхід забезпечує реалістичне фізичне відбивання від перешкод під будь-яким кутом.

Перевірка зіткнення гравця з ворогами

```
            if enemy.get_rect().intersects(self.player.get_rect()):
                self.game_over = True
```

Після оновлення позиції кожного ворога перевіряється, чи не перетинається він з гравцем. Якщо так — гра завершується встановленням прапорця game_over.

Управління клавіатурою

Обробка натискання клавіш реалізована через перевизначення методу keyPressEvent():

```
    def keyPressEvent(self, event):
        if event.key() == Qt.Key_Escape:
            self.close()
            return

        if self.game_over:
            if event.key() == Qt.Key_Space:
                self.player = Player(400, 300)
                self.enemies = []
                self.obstacles = []
                self.score = 0
                self.game_over = False
                self.init_game()
            return

        old_x, old_y = self.player.x, self.player.y
        step = 15

        if event.key() == Qt.Key_Left:
            self.player.x -= step
        elif event.key() == Qt.Key_Right:
            self.player.x += step
        elif event.key() == Qt.Key_Up:
```

```
            self.player.y -= step
        elif event.key() == Qt.Key_Down:
            self.player.y += step
        else:
            return
```

Особливості реалізації:

1. Клавіша ESC завершує програму
2. Клавіша SPACE перезапускає гру після завершення
3. Перед переміщенням зберігаються старі координати (old_x, old_y)
4. Крок переміщення становить 15 пікселів
5. Стрілки змінюють відповідні координати

Перевірка меж екрану:

```
        if self.player.x - self.player.size/2 < 0:
            self.player.x = self.player.size/2
        elif self.player.x + self.player.size/2 > self.width():
            self.player.x = self.width() - self.player.size/2

        if self.player.y - self.player.size/2 < 0:
            self.player.y = self.player.size/2
        elif self.player.y + self.player.size/2 > self.height():
            self.player.y = self.height() - self.player.size/2
```

Після переміщення координати обмежуються так, щоб гравець не міг вийти за межі екрану. Враховується, що координати (x, y) вказують на центр об'єкта, тому використовується розмір об'єкта поділений навпіл.

Перевірка зіткнення з перешкодами:

```
        for obstacle in self.obstacles:
            if self.player.get_rect().intersects(obstacle.rect):
                self.player.x, self.player.y = old_x, old_y
                break
```

Якщо після переміщення гравець перетинається з будь-якою перешкодою, його координати повертаються до попередніх значень, ефективно блокуючи рух.

Управління мишею

Обробка кліків миші реалізована через метод mousePressEvent():

```
    def mousePressEvent(self, event):
        if self.game_over:
            return

        modifiers = QApplication.keyboardModifiers()
        if modifiers != Qt.ShiftModifier:
            return

        old_x, old_y = self.player.x, self.player.y

        self.player.x = event.pos().x()
        self.player.y = event.pos().y()
```

Особливості:

1. Миша працює тільки з утриманою клавішею SHIFT для уникнення випадкових кліків
2. Перевіряється, чи не завершена гра
3. Координати гравця встановлюються відразу в точку кліку
4. Застосовуються ті самі перевірки меж екрану та зіткнень з перешкодами

Це дозволяє гравцю швидко телепортуватися в будь-яку точку екрану, але не дає можливості пройти крізь перешкоди.

Система підрахунку очок та динамічна складність

```
    def increase_score(self):
        if not self.game_over:
            self.score += 1

            if self.score % 100 == 0 and self.score > 0:
                x = random.randint(50, 700)
                y = random.randint(50, 500)
                while self.check_spawn_collision(x, y):
                    x = random.randint(50, 700)
                    y = random.randint(50, 500)
                speed = random.randint(3, 5)
                self.enemies.append(Enemy(x, y, speed))
```

Кожні 100 мілісекунд рахунок збільшується на 1. Коли рахунок досягає кратного 100 (100, 200, 300...), додається новий ворог з випадковою позицією та трохи вищою швидкістю (3-5 замість початкових 2-4). Це створює прогресивне збільшення складності гри.

Візуалізація

Метод paintEvent() відповідає за відображення всіх графічних елементів:

```
    def paintEvent(self, event):
        painter = QPainter(self)
        painter.setRenderHint(QPainter.Antialiasing)

        painter.setPen(QPen(QColor(80, 80, 80), 2))
        painter.setBrush(QBrush(QColor(40, 40, 40)))
        for obstacle in self.obstacles:
            painter.drawRect(obstacle.rect)

        painter.setPen(QPen(QColor(255, 0, 0), 2))
        painter.setBrush(QBrush(QColor(200, 0, 0)))
        for enemy in self.enemies:
            painter.drawRect(enemy.get_rect())

        painter.setPen(QPen(QColor(255, 255, 255), 3))
        painter.setBrush(QBrush(QColor(255, 255, 255)))
        painter.drawEllipse(QPointF(self.player.x, self.player.y),
                            self.player.size/2, self.player.size/2)

        painter.setPen(QPen(QColor(255, 255, 255)))
        painter.setFont(QFont("Arial", 24, QFont.Bold))
```

```
        painter.drawText(20, 40, str(self.score))
```

Порядок малювання:

1. Увімкнення антиаліасингу для згладжування країв
2. Малювання перешкод (темно-сірі прямокутники з сірою рамкою)
3. Малювання ворогів (червоні квадрати з червоною рамкою)
4. Малювання гравця (біле коло з білою рамкою)
5. Відображення рахунку (білий текст у верхньому лівому куті)

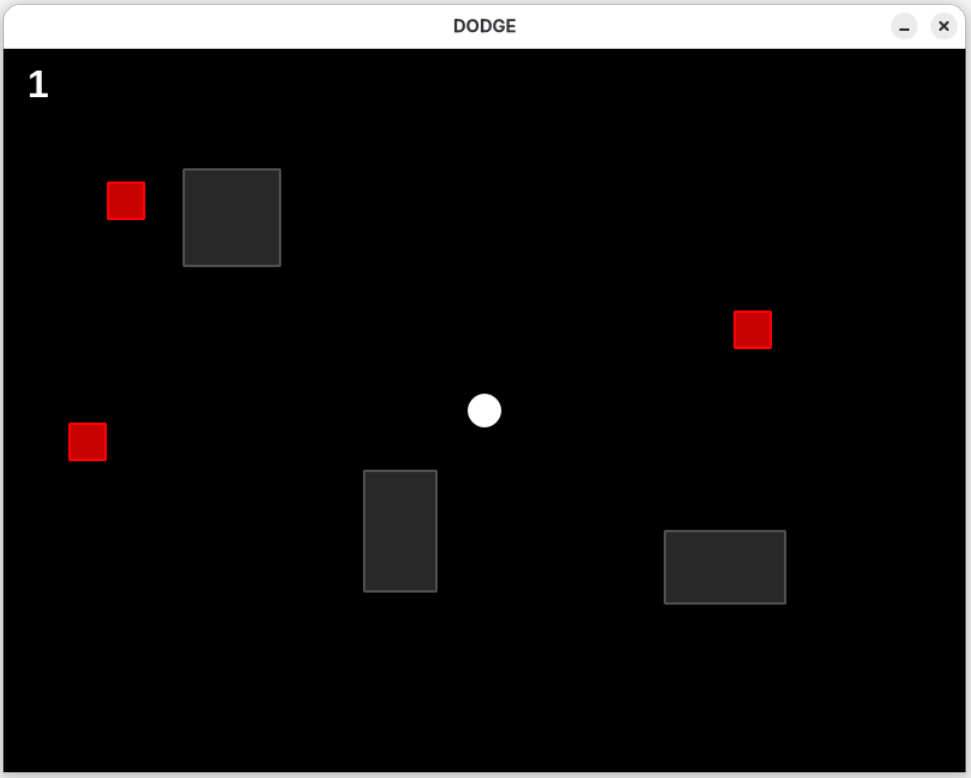
Екран завершення гри:

```
        if self.game_over:
            painter.fillRect(self.rect(), QColor(0, 0, 0, 200))

            painter.setPen(QPen(QColor(255, 0, 0)))
            painter.setFont(QFont("Arial", 60, QFont.Bold))
            painter.drawText(self.rect(), Qt.AlignCenter, "GAME OVER")

            painter.setPen(QPen(QColor(255, 255, 255)))
            painter.setFont(QFont("Arial", 24))
                    painter.drawText(self.rect().adjusted(0,  100,  0,  0),
Qt.AlignCenter,
                             f"SCORE: {self.score}")

            painter.setFont(QFont("Arial", 16))
                    painter.drawText(self.rect().adjusted(0,  150,  0,  0),
Qt.AlignCenter,
                             "PRESS SPACE TO RESTART")
```

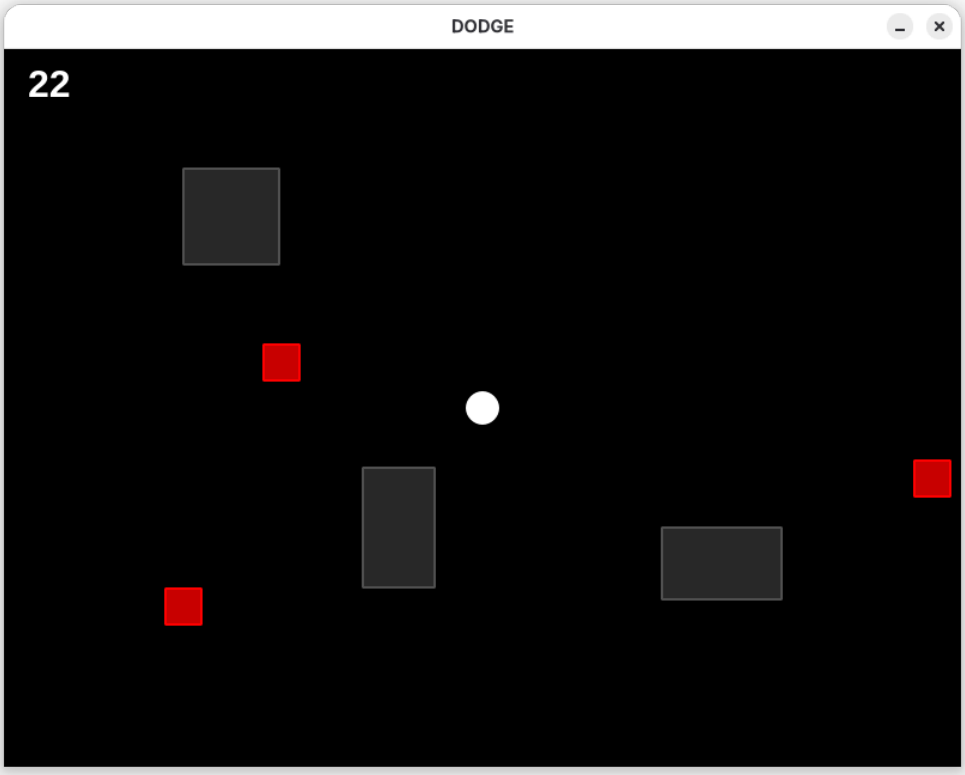
При завершенні гри поверх всього малюється напівпрозоре чорне затемнення (альфа-канал 200), великий червоний текст "GAME OVER", фінальний рахунок та підказка про перезапуск.

Результати

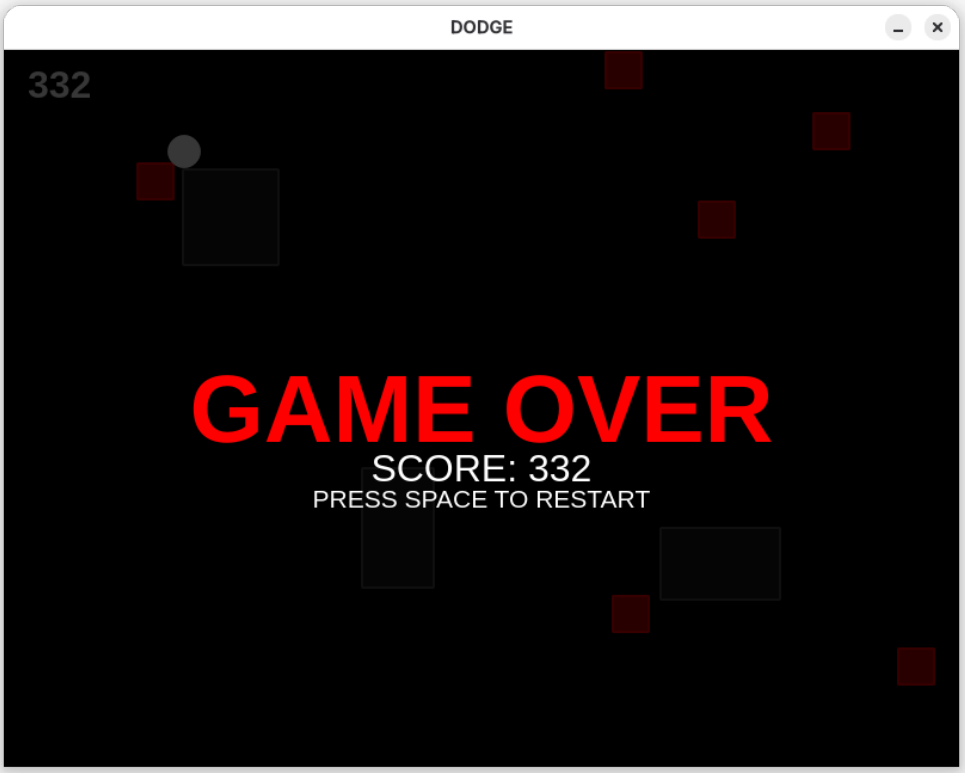
Скріншот 1: Початковий стан гри



Скріншот 2: Процес гри



Скріншот 3: Екран завершення гри



Висновки

В ході виконання лабораторної роботи було успішно освоєно та реалізовано:

Теоретичні аспекти:

- Принципи роботи з таймерами QTimer для створення анімації
- Обробку подій клавіатури через keyPressEvent()
- Обробку подій миші через mousePressEvent()
- Методику перемальовування графіки через paintEvent()
- Алгоритми виявлення зіткнень (collision detection)
- Структуру класів у PyQt5

Практичні навички:

- Створення плавної анімації з частотою 60 FPS
- Реалізація фізично коректного відбивання об'єктів від перешкод
- Розробка алгоритму визначення сторони зіткнення
- Комбінування різних методів управління (клавіатура + миша)
- Створення системи підрахунку очок та динамічної складності
- Проектування мінімалістичного та функціонального інтерфейсу

Виконання вимог завдання:

Програма повністю відповідає всім вимогам лабораторної роботи:

1. Малювання об'єктів — реалізовано через QPainter з використанням примітивів (коло, прямокутники)
2. Безперервна траєкторія руху — вороги рухаються автоматично завдяки таймеру з частотою оновлення 60 FPS
3. Об'єкти-перешкоди та відбивання — три статичні перешкоди, від яких коректно відбиваються рухомі об'єкти з визначенням сторони зіткнення
4. Управління клавіатурою — реалізовано плавне переміщення стрілками з перевіркою меж та зіткнень
5. Управління мишею — реалізовано телепортацію у точку кліку з тими самими перевірками
6. Перемикання між режимами — можливість використовувати як клавіатуру, так і мишу в будь-який момент гри

Особливу увагу було приділено:

- Фізичній коректності — об'єкти відбиваються під правильними кутами
- Плавності анімації — стабільні 60 FPS без затримок
- Зручності інтерфейсу — мінімалістичний дизайн з чіткою індикацією стану
- Ігровій механіці — прогресивне збільшення складності утримує інтерес

Реалізована програма демонструє комплексне розуміння роботи з графікою, подіями та анімацією в PyQt5. Об'єктно-орієнтований підхід дозволяє легко розширювати функціональність, додаючи нові типи об'єктів або механік.