Technologie IoT - Programowanie Python
Klon gry Asteroids

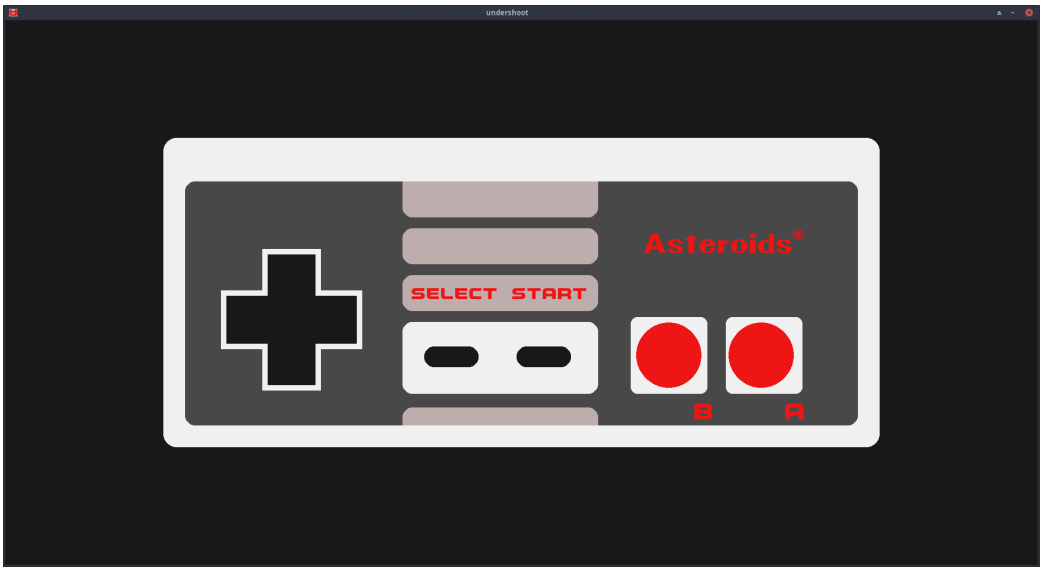
My Name, w12345

1 Opis projektu

Celem projektu jest wykonanie gry zainspirowanej grą Asteroids [1] w bibliotece Pygame [7]. Motywem gry jest tester kontrolerów, włączając grę widzimy narysowany kontroller do gier na wzór kontrolera do konsoli NES.

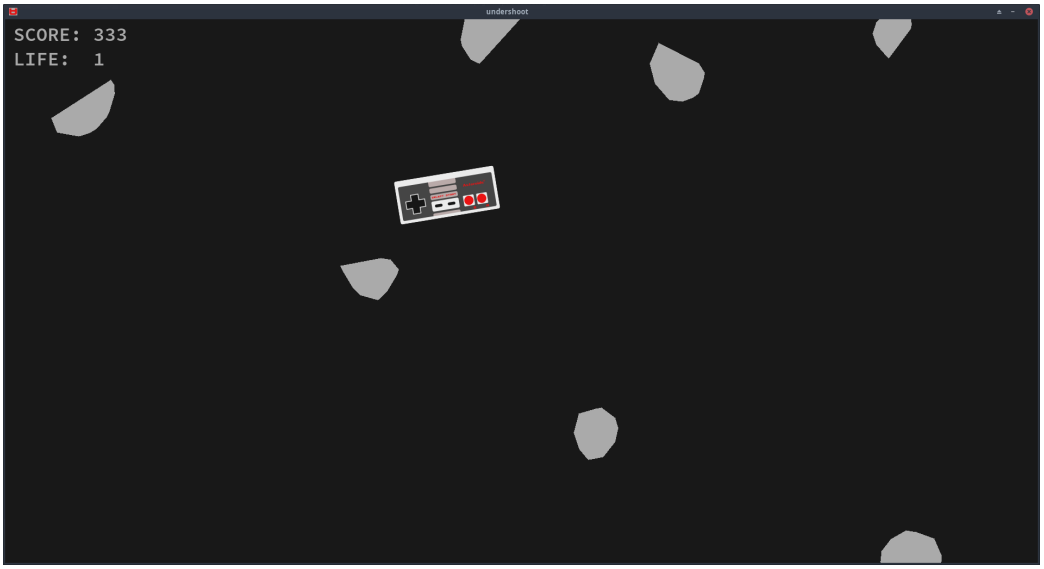
Rysunek 1: Nintendo Entertainment System

Wszystkie przyciski widoczne w menu gry podswietlają się w momencie wcisnięcia ich na kontrolerze, brakuje tu gałek analogowych i triggerów z typowych kontrolerów, jednak kontroler konsoli NES ich nie posiadał.



Rysunek 2: Menu gry

Rozgrywka rozpoczyna się po wcisnięciu przycisku START, zostaje wtedy wykonana animacja oddalająca kontroler, pojawiają się napisy informujące o punktach oraz życiu gracza, asteroidy zaczynają się pojawiać, a nam zostaje dana możliwość kontroli naszego 'statku'.



Rysunek 3: Rozgrywka

Naszym zadaniem jest strzelanie do i unikanie asteroid, jeżeli asteroida uderzy w nas 3 razy, pokazuje się ekran informujący o przegranej oraz o zdobytej liczbie punktów.

Rysunek 4: Przegrana

2 Struktura projektu

2.1 Lista plików

```
Projekt
├── conf.py
├── dpad.py
├── entity.py
├── lost_screen.py
├── misc.py
├── polygon.py
├── separation_axis_theorem.py
├── tester.py
├── undershoot
├── fonts/
│   ├── Need Every Sound.ttf
│   ├── NintendBoldRM8E.ttf
│   └── SourceCodePro.ttf
└── images/
    └── icon.png
```

Plik **undershoot** (plik .py) jest plikiem głównym, w nim znajduje się główna pętla gry oraz inicjalizacja wszystkich systemów i importowanie reszty plików.

Plik **conf.py** zawiera większość zmiennych konfiguracyjnych, dużo plików importuje jego zawartość.

Pliki **tester.py** oraz **dpad.py** są odpowiedzialne za tworzenie powierzchni (surface) z narysowanym kontrolerem, **lost_screen.py** zawiera funkcję tworzącą powierzchnię widoczną po przegraniu.

enitity.py i **polygon.py** to obiekty gry, **separation_axis_theorem.py** jest importowane przez **polygon.py** i jest zewnętrzną biblioteką używaną do sprawdzenia kolizji między dwoma wielokątami.

Reszta plików to czcionki oraz ikona gry.

2.2 Inicjalizacja i pętla gry

Uproszczona struktura programu wygląda w następujący sposób:

```
# import and initialize everything...
player = Player()
asteroids, projectiles = [], []

delta_time = get_dt()
running = True
while running:
    buttons = handle_input_events()

    maybe_spawn_asteroid(asteroids, delta_time)

    player.update(buttons, delta_time)
    for asteroid in asteroids:
        asteroid.update(delta_time)
        if asteroid.collides(player):
            # handle collision
            player.kill_or_something()
    for projectile in projectiles:
        projectile.update(delta_time)
        # ...

    screen.clear()
    for entity in entity_list:
        entity.draw()
    screen.flip()

    delta_time = get_dt()
```

Nie została tu pokazana pętla menu głownego. Dodatkowo, w rzeczywistości program jest dużo bardziej rozbudowany. Wiele logiki, takiej jak sprawdzanie kolizji i wykonanie jej skutków, znajduje się w pierwszej warstwie pętli (zamiast być głebiej ukryte, za kilkoma wywołaniami funkcji). W przypadku tak prostej i krótkiej gry, taka struktura dużo upraszcza.

2.3 Abstrakcje

Główne abstrakcje wykorzystywane w programie to:

- klasa `Entity` i klasy dziedziczące,
- klasa `Polygon`,
- funkcja `make_controller_surface()`

Entity zawiera właściwości fizyczne, tj. pozycję, orientację, ruch oraz przyśpieszenie. Każde Entity musi posiadać hitbox - obiekt klasy Polygon wykorzystywany w metodzie Entity.collides(). Dodatkowo, każde Entity może implementować metody .update() oraz .draw(). Z Entity dziedziczą klasy Player, Asteroid i Projectile.

Polygon jest wykorzystywany jedynie jako hitbox należący do klasy Entity.

Player, klasa dziedzicząca z Entity, wykorzystuje make_controller_surface() przy użyciu metody .draw(). Zwrócona powierzchnia kontrolera z narysowanymi aktualnie wciśniętymi przyciskami jest transformowana(rotacja, skalowanie) i rysowana w pozycji pozycji obiektu Player.

3 Działanie różnych systemów

3.1 Delta time

Delta time [4] to czas od ostatniej iteracji pętli, przekazywanie jej do funkcji zmieniającej stan gry jest konieczne, aby zachować tą samą prędkość fizyki gry, przy różnych ilościach klatek na sekundę.

3.2 Symulacja fizyki

Poruszanie się obiektów w grze jest wykonane w bardzo prosty sposób, każdy obiekt posiada wektory pozycji, przyśpieszenia oraz ruchu. Przy każdym użyciu metody .update() następuje:

```
def update(self):
    self.velocity += self.acceleration
    self.position += self.velocity
```

Takie operacje są lepiej opisane w książce The Nature of Code, dział 1.10 'Interactivity with Acceleration' [6].

Następnie występuje rotacja obiektów o 'rotation_velocity', w tym wypadku hitbox obiektu musi zostać obrócony. Metoda .rotate() w klasie Polygon:

```
def rotate(self, angle):
    self.points = [ (p - self.middle).rotate(angle) + self.middle
                                                for p in self.points ]
```

środek (middle) to punkt równy średniej wszystkich innych punktów wielokąta.

Kolizje między obiektami są sprawdzane przy pomocy dwóch funkcji, na początku zostaje wywołane maybe_collides():

```
def maybe_collides(self, polygon):
    x1, x2, y1, y2 = self.rect_coords
    for x, y in polygon.rect_points:
        if x1 <= x <= x2 and y1 <= y <= y2:
            return True
    return False
```

rect_coords to współrzędne a rect_points to punkty prostokąta, który jest wrysowany sprawdzany wielokąt. Są one aktualizowane przy użyciu metod .move() oraz .rotate(). Użycie tej metody jest bardzo szybkie, dlatego dopiero w sytuacji kiedy maybe_collides() zwraca True, używana jest metoda collides().

collides() korzysta z algorytmu Separating Axis Theorem [2] [3], do którego została użyta zewnętrza biblioteka [5]

3.3 Generacja asteroid

Generacja asteroid jest niekompletna - nie możemy ustawić rozmiaru asteroidy i generowane są niedokładne, mają zbyt małą lub zbyt dużą ilość kątów.

```
points = [ Vector2(position) ]
start = Vector2(1, 0)

for _ in range(9):
    start = start.normalize() * random.randint(400, 2000) / 50
    start = start.rotate(random.randint(80, 600)/10)
    points.append(points[-1] + start)
```

możliwymi zmianami algorytmu jest liczenie długości boków, sumy kątów, zmiana zakresu liczb użytego w funkcjach losowych (prawdopodobnie w zależności od chcianego rozmiaru), sprawdzanie pozycji pierwszego punktu wobec pierwszego punktu.

3.4 Tworzenie nowych asteroid

Nowe asteroidy pojawiają się na mapie losowo, z szansą wynoszącą 10 + 5k na jedną minutę, gdzie k=liczba minut.

```
rate = 10 + 5*minutes
chance = int(rate * delta_time*100)
if random.randint(0, 60*1000*100) < chance:
    asteroids.append(Asteroid(player=p))
```

mnożenie delta_time (liczba ms) przez 100 służy dokładności, mnożenie przez 1000 jest konieczne, ponieważ ten kod wykonuje się wiele razy w ciągu jednej sekundy.

Bibliografia

[1] *Asteroids (video game)*. URL: https://en.wikipedia.org/wiki/Asteroids_(video_game).

[2] *Collision tutorial (adobe flash emulation necessary)*. URL: https://www.metanetsoftware.com/technique/tutorialA.html/.

[3] *David Eberly, Magic Software, Inc. - Method Of Separating Axes*. URL: https://web.archive.org/web/20050130112914/http://www.magic-software.com/Documentation/MethodOfSeparatingAxes.pdf/.

[4] *Delta Time*. URL: https://gamedev.stackexchange.com/questions/13008/how-to-get-and-use-delta-time.

[5] *Juan Antonio Aldea, SAT Library*. URL: https://github.com/JuantAldea/Separating-Axis-Theorem.

[6] *Nature of Code*. URL: https://natureofcode.com/book/chapter-1-vectors/.

[7] *Pygame documentation*. URL: https://www.pygame.org/docs/.