PyGame – Gra w Życie Conwaya

Opis implementacji: Używając biblioteki PyGame oraz języka Python, stworzymy prostą implementację gry w życie Conwaya.

Autorzy: Łukasz Zarzecki, Robert Bednarz

Czas realizacji: 90 min

Poziom trudności: Poziom 3

Biblioteka PyGame ułatwia tworzenie aplikacji multimedialnych, w tym gier.

I. Zmienne i plansza gry

Tworzymy plik **life.py** w terminalu lub w wybranym edytorze i zaczynamy od zdefiniowania zmiennych określających właściwości obiektów w naszej grze.

Kod I.1

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import pygame, sys, random
from pygame.locals import * #udostępnienie nazw metod z locals

# inicjacja modułu pygame
pygame.init()

# szerokość i wysokość okna gry
OKNOGRY_SZER = 800
OKNOGRY_WYS = 400

# przygotowanie powierzchni do rysowania, czyli inicjacja okna gry
OKNOGRY = pygame.display.set_mode((OKNOGRY_SZER, OKNOGRY_WYS), 0, 32)
# tytuł okna gry
pygame.display.set_caption('Gra o życie')

# rozmiar komórki
ROZ_KOM = 10
# ilość komórek w poziomie i pionie
KOM_POZIOM = OKNOGRY_SZER/ROZ_KOM
KOM_PION = OKNOGRY_WYS/ROZ_KOM

# wartości oznaczające komórki "martwe" i "żywe"
KOM_MARTWA = 0
KOM_ZYWA = 1

# lista opisująca stan pola gry, 0 - komórki martwe, 1 - komórki żywe
# na początku tworzymy listę zawierającą KOM_POZIOM zer
POLE_GRY = [KOM_MARTWA] * KOM_POZIOM
# rozszerzamy listę o listy zagnieżdżone, otrzymujemy więc listę dwuwymiarową
for i in range(KOM_POZIOM):
    POLE_GRY[i] = [KOM_MARTWA] * KOM_PION
```

W instrukcji **`pygame.display.set_mode()`** inicjalizujemy okno gry o rozmiarach 800x400 pikseli i 32 bitowej głębi kolorów. Tworzymy w ten sposób powierzchnię główną do rysowania zapisaną w zmiennej **`OKNOGRY`**. Ilość możliwych do narysowania komórek, reprezentowanych przez kwadraty o boku 10 pikseli, wyliczamy w zmiennych **`KOM_POZIOM`** i **`KOM_PION`**. Najważniejszą strukturą w naszej grze jest **`POLE_GRY`**, dwuwymiarowa lista elementów reprezentujących "żywe" i "martwe" komórki, czyli populację. Tworzymy ją w dwóch krokach, na początku inicjujemy zerami jednowymiarową listę o rozmiarze odpowiadającym ilości komórek w poziomie (**`POLE_GRY = [KOM_MARTWA] * KOM_POZIOM`**). Następnie do każdego elementu listy przypisujemy listę zawierającą tyle zer, ile jest komórek w pionie.

II. Populacja komórek

Kolejnym krokiem będzie zdefiniowanie funkcji przygotowującej i rysującej populację komórek.

Kod II.1

```
# przygotowanie następnej generacji komórek, czyli zaktualizowanego POLA_GRY
def przygotuj_populacje(polegry):
    # na początku tworzymy 2-wymiarową listę wypełnioną zerami
    nast_gen = [KOM_MARTWA] * KOM_POZIOM
    for i in range(KOM_POZIOM):
        nast_gen[i] = [KOM_MARTWA] * KOM_PION

    # iterujemy po wszystkich komórkach
    for y in range(KOM_PION):
        for x in range(KOM_POZIOM):

            # zlicz populację (żywych komórek) wokół komórki
            populacja = 0
            # wiersz 1
            try:
                if polegry[x-1][y-1] == KOM_ZYWA: populacja += 1
            except IndexError:pass
            try:
                if polegry[x][y-1] == KOM_ZYWA: populacja += 1
            except IndexError:pass
            try:
                if polegry[x+1][y-1] == KOM_ZYWA: populacja += 1
            except IndexError:pass

            # wiersz 2
            try:
                if polegry[x-1][y] == KOM_ZYWA: populacja += 1
            except IndexError:pass
            try:
                if polegry[x+1][y] == KOM_ZYWA: populacja += 1
            except IndexError:pass

            # wiersz 3
            try:
                if polegry[x-1][y+1] == KOM_ZYWA: populacja += 1
            except IndexError:pass
            try:
                if polegry[x][y+1] == KOM_ZYWA: populacja += 1
            except IndexError:pass
            try:
                if polegry[x+1][y+1] == KOM_ZYWA: populacja += 1
            except IndexError:pass

            # "niedoludnienie" lub przeludnienie = śmierć komórki
            if polegry[x][y] == KOM_ZYWA and (populacja < 2 or populacja > 3):
                nast_gen[x][y] = KOM_MARTWA
            # życie trwa
            elif polegry[x][y] == KOM_ZYWA and (populacja == 3 or populacja == 2):
                nast_gen[x][y] = KOM_ZYWA
            # nowe życie
            elif polegry[x][y] == KOM_MARTWA and populacja == 3:
                nast_gen[x][y] = KOM_ZYWA

    # zwróć nowe polegry z następną generacją komórek
    return nast_gen

# rysowanie komórek (kwadratów) żywych
def rysuj_populacje():
```

```
    for y in range(KOM_PION):
        for x in range(KOM_POZIOM):
            if POLE_GRY[x][y] == KOM_ZYWA:
                pygame.draw.rect(OKNOGRY, (255,255,255),
Rect((x*ROZ_KOM,y*ROZ_KOM),(ROZ_KOM,ROZ_KOM)),1)
```

Najważniejszym fragmentem kodu, implementującym logikę naszej gry, jest funkcja **`przygotuj_populacje()`**, która jako parametr przyjmuje omówioną wcześniej strukturę **`POLE_GRY`** (pod nazwą **`polegry`**). Funkcja sprawdza, jak rozwija się **`populacja`** komórek, według następujących zasad:

1. Jeżeli żywa komórka ma mniej niż 2 żywych sąsiadów, umiera z powodu samotności.
2. Jeżeli żywa komórka ma więcej niż 3 żywych sąsiadów, umiera z powodu przeludnienia.
3. Żywa komórka z 2 lub 3 sąsiadami żyje dalej.
4. Martwa komórka z 3 żywymi sąsiadami ożywa.

Funkcja iteruje po każdym elemencie **`POLA_GRY`** i sprawdza stan sąsiadów każdej komórki, w wierszu 1 powyżej komórki, w wierszu 2 na tym samym poziomie i w wierszu 3 poniżej. Konstrukcja **`try...except`** pozwala obsłużyć sytuacje wyjątkowe (błędy), a więc komórki skrajne, które nie mają sąsiadów u góry czy u dołu, z lewej bądź z prawej strony: w takim przypadku wywoływana jest instrukcja **`pass`**, czyli nie rób nic :-). Końcowa złożona instrukcja warunkowa **`if`** ożywia lub uśmierca sprawdzaną komórkę w zależności od stanu sąsiednich komórek (czyli zmiennej **`populacja`**).

Zadaniem funkcji **`rysuj_populacje()`** jest narysowanie kwadratów (obiekty **`Rect`**) o białych bokach w rozmiarze 10 pikseli dla pól (elementów), które w liście **`POLE_GRY`** są żywe (mają wartość 1).

III. Główna pętla programu

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w pętli, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (przesunięcia elementów, aktualizacja planszy),
3. aktualizacja wyświetlanego okna (narysowanie nowego stanu gry).

Dopisujemy więc do kodu główną pętlę wraz z obsługą zdarzeń:

Kod III.1

```
zycie_trwa = False
przycisk_wdol = False

# pętla główna programu
while True:
    # obsługa zdarzeń generowanych przez gracza
    for event in pygame.event.get():
        # przechwyć zamknięcie okna
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == KEYDOWN and event.key == K_RETURN:
            zycie_trwa = True

        if zycie_trwa == False:
```

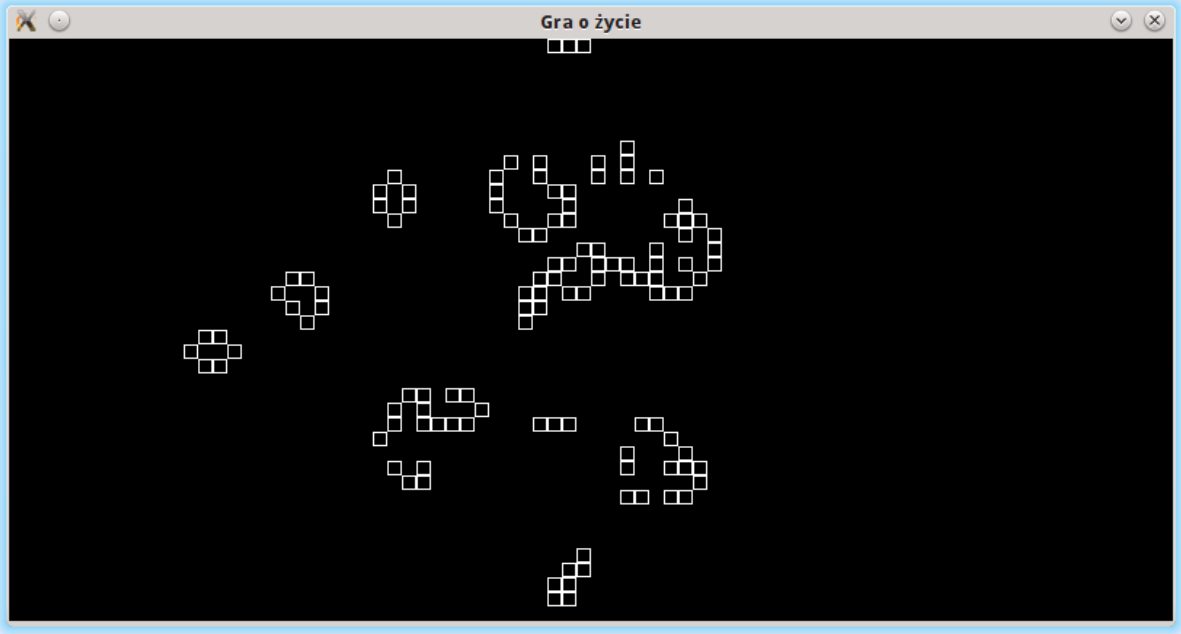
```
            if event.type == MOUSEBUTTONDOWN:
                przycisk_wdol = True
                przycisk_typ = event.button

            if event.type == MOUSEBUTTONUP:
                przycisk_wdol = False

            if przycisk_wdol:
                mouse_x, mouse_y = pygame.mouse.get_pos()
                mouse_x = mouse_x / ROZ_KOM
                mouse_y = mouse_y / ROZ_KOM
                # lewy przycisk myszy ożywia
                if przycisk_typ == 1: POLE_GRY[mouse_x][mouse_y] = KOM_ZYWA
                # prawy przycisk myszy uśmierca
                if przycisk_typ == 3: POLE_GRY[mouse_x][mouse_y] = KOM_MARTWA

    if zycie_trwa == True:
        POLE_GRY = przygotuj_populacje(POLE_GRY)

    OKNOGRY.fill((0,0,0)) # ustaw kolor okna gry
    rysuj_populacje()
    pygame.display.update()
    pygame.time.delay(100)
```

W obrębie głównej pętli programu pętla **`for`** odczytuje kolejne zdarzenia zwracane przez metodę **`pygame.event.get()`**. Jak widać, w pierwszej kolejności obsługujemy wydarzenie typu (właściwość **`.type`**) QUIT, czyli zakończenie aplikacji.

Jednak na początku gry gracz klika lewym lub prawym klawiszem myszy i ożywia lub uśmierca kliknięte komórki w obrębie okna gry. Dzieje się tak dopóty, dopóki zmienna **`zycie_trwa`** ma wartość **`False`**, a więc dopóki gracz nie naciśnie klawisza ENTER (**`if event.type == KEYDOWN and event.key == K_RETURN:`**). Każde kliknięcie myszą zostaje przechwycone (**`if event.type == MOUSEBUTTONDOWN:`**) i zapamiętane w zmiennej **`przycisk_wdol`**. Jeżeli zmienna ta ma wartość **`True`**, pobieramy współrzędne kursora myszy (**`mouse_x, mouse_y = pygame.mouse.get_pos()`**) i obliczamy indeksy elementu listy POLE_GRY odpowiadającego kliknietej komórce. Następnie sprawdzamy, który przycisk myszy został naciśnięty; informację tę zapisaliśmy wcześniej za pomocą funkcji **`event.button`** w zmiennej **`przycisk_typ`**, która przyjmuje wartość 1 (lewy) lub 3 (prawy przycisk myszy), w zależności od klikniętego przycisku ożywiamy lub uśmiercamy komórkę, zapisując odpowiedni stan w liście **`POLE_GRY`**.

Naciśnięcie klawisza ENTER uruchamia symulację rozwoju populacji. Zmienna **`zycie_trwa`** ustawiona zostaje na wartość **`True`** , co przerywa obsługę kliknięć myszą, i wywoływana jest funkcja **`przygotuj_populacje()`**, która przygotowuje kolejny stan populacji. Końcowe polecenia wypełniają okno gry kolorem (**`.fill()`**), wywołują funkcję rysującą planszę (**`rysuj_populacje()`**). Funkcja **`pygame.display.update()`**, która musi być wykonywana na końcu rysowania, aktualizuje obraz gry na ekranie. Ostatnie polecenie **`pygame.time.delay(100)`** dodaje 100-milisekundowe opóźnienie kolejnej aktualizacji stanu populacji. Dzięki temu możemy obserwować jej rozwój na planszy.

Grę możemy uruchomić poleceniem wpisanym w terminalu: **`python life.py`**.



Słownik

- **Kanał alfa (ang. alpha channel)** – w grafice komputerowej jest kanałem, który definiuje przezroczyste obszary grafiki. Jest on zapisywany dodatkowo wewnątrz grafiki razem z trzema wartościami barw składowych RGB.
- **Inicjalizacja** – proces wstępnego przypisania wartości zmiennym i obiektom. Każdy obiekt jest inicjalizowany różnymi sposobami zależnie od swojego typu.
- **Iteracja** – czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli. Mianem iteracji określa się także operacje wykonywane wewnątrz takiej pętli.
- **Zdarzenie (ang. event)** – zapis zajścia w systemie komputerowym określonej sytuacji, np. poruszenie myszką, kliknięcie, naciśnięcie klawisza.
- **pygame.display.set_mode()** – inicjuje okno lub ekran do wyświetlania, parametry: rozdzielczość w pikselach = (x,y), flagi, głębia koloru.
- **pygame.display.set_caption()** – ustawia tytuł okna, parametr: tekst tytułu.
- **pygame.Surface()** – obiekt reprezentujący dowolny obrazek (grafikę), który ma określoną rozdzielczość (szerokość i wysokość) oraz format pikseli (głębokość, przezroczystość); **SRCALPHA** – oznacza, że format pikseli będzie zawierać ustawienie alfa (przezroczystości); **.fill()** – wypełnia obrazek kolorem; **.get_rect()** – zwraca prostokąt zawierający obrazek, czyli obiekt **Rect**; **.convert_alpha()** – zmienia format pikseli, w tym przezroczystość; **.blit()** – rysuje jeden obrazek na drugim, parametry: źródło, cel.
- **pygame.draw.rect()** – rysuje prostokąt na wskazanej powierzchni, parametry: powierzchnia, kolor, obiekt **Rect**, grubość obramowania.
- **pygame.draw.ellipse()** – rysuje okrągły kształt wewnątrz prostokąta, parametry: przestrzeń, kolor, prostokąt.
- **pygame.font.Font()** – tworzy obiekt czcionki z podanego pliku; **.render()** – tworzy

nową powierzchnię z podanym tekstem, parametry: tekst, antyaliasing, kolor, tło.

- **pygame.event.get()** – pobiera zdarzenia z kolejki zdarzeń; **event.type()** – zwraca identyfikator SDL typu zdarzenia, np. KEYDOWN, KEYUP, MOUSEMOTION, MOUSEBUTTONDOWN, QUIT.
- **SDL** (Simple DirectMedia Layer) – międzyplatformowa biblioteka ułatwiająca tworzenie gier i programów multimedialnych.
- **pygame.Rect** – obiekt pygame przechowujący współrzędne prostokąta; **.centerx, .x, .y, .top, .bottom, .left, .right** – wirtualne własności obiektu prostokąta określające jego położenie; **.collliderect()** – metoda sprawdza czy dwa prostokąty nachodzą na siebie.

POĆWICZ SAM

Zmień grę tak, aby zaczynał ją komputer.

Dodaj do gry możliwość rozgrywki wielokrotnej bez konieczności ponownego uruchamiania skryptu.

Zmodyfikuj funkcję rysującą pole gry tak, aby komputer rysował krzyżyki, a nie kółka.