Raport z projektu 2 (Python)

Piotr Sieński 184297

Wstęp

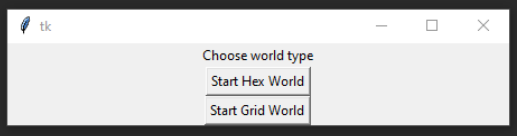
Projekt jest podzielony na klasy świata: klasę abstrakcyją World oraz dziedziczące po niej klasy HexWorld i GridWorld. Istnieje klasa abstrakcyjna Organism, dziedziczące po niej klasy Plant oraz Animal i implementujące poszczególne organizmy klasy dziedziczące po Plant lub Animal.

Klasa World zawiera ogólne metody potrzebne do funkcjonowania świata zaś klasy dziedziczące implementują metody abstrakcyjne których działanie zależy od typu symulowanego świata. W klasie Organism zaimplementowane są dwie różne metody wyświetlania organizmu na planszy. Zależnie czy świat jest złożony z kwadratów czy sześciokątów wywołanie metody display() organizmu wywołuje odpowiednia metodę wyświetlania. Ruch zwierząt również jest zależny od kształtu pola w świecie - zależnie od rodzaju świata rózni się ilość możliwych ruchów dla zwierzęcia - są to 4 kierunki ruchu dla świata kwadratowego oraz 6 dla sześciokątnego. Poprzez implementację poruszania się człowieka na czterech strzałkach człowiek porusza się niezależnie od rodzaju świata w czterech kierunkach.

GUI

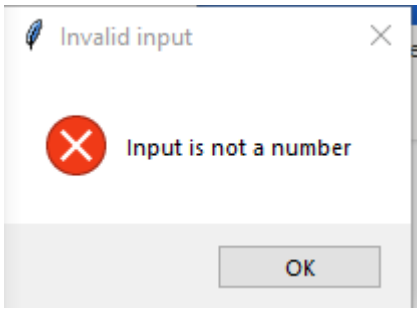
W pliku main.py znajduje się skrypt inicjujący symulację. Wyświetlane jest okno pozwalające na wybór czy świat ma składać się z sześciokątów czy kwadratów, następnie tworzony jest odpowiedni świat

```python
def start(t, window):
    window.quit()
    window.destroy()
    if t == "Hex":
        HexWorld()
    else:
        GridWorld()


def display_start_window():
    win = tk.Tk()
    label = tk.Label(master=win, text="Choose world type")
    btn_hex = tk.Button(master=win, text="Start Hex World", command=lambda: start("Hex", win))
    btn_grid = tk.Button(master=win, text="Start Grid World", command=lambda: start("Grid", win))
    label.pack()
    btn_hex.pack()
    btn_grid.pack()
    win.mainloop()


if __name__ == '__main__':
    display_start_window()
```



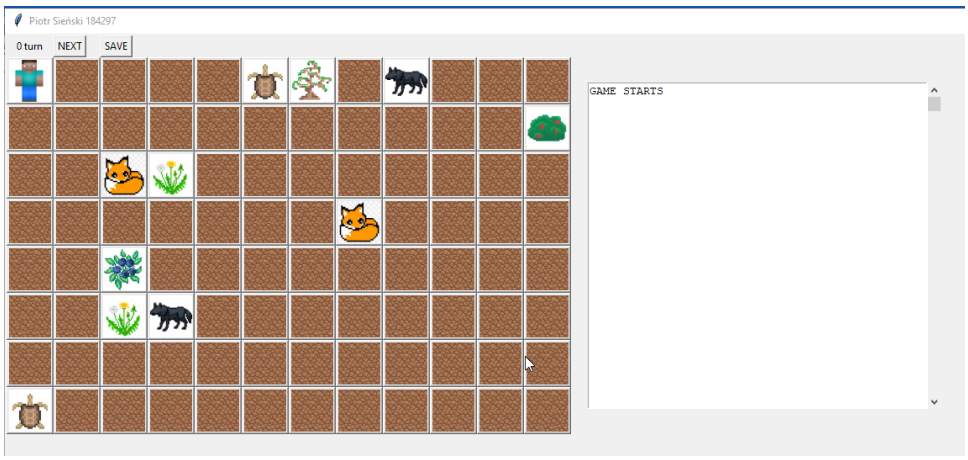
Następnie w metodzie klasy World get_info() wyświetlane jest okno pozwalające wybrać sposób generacji świata oraz jego parametry.

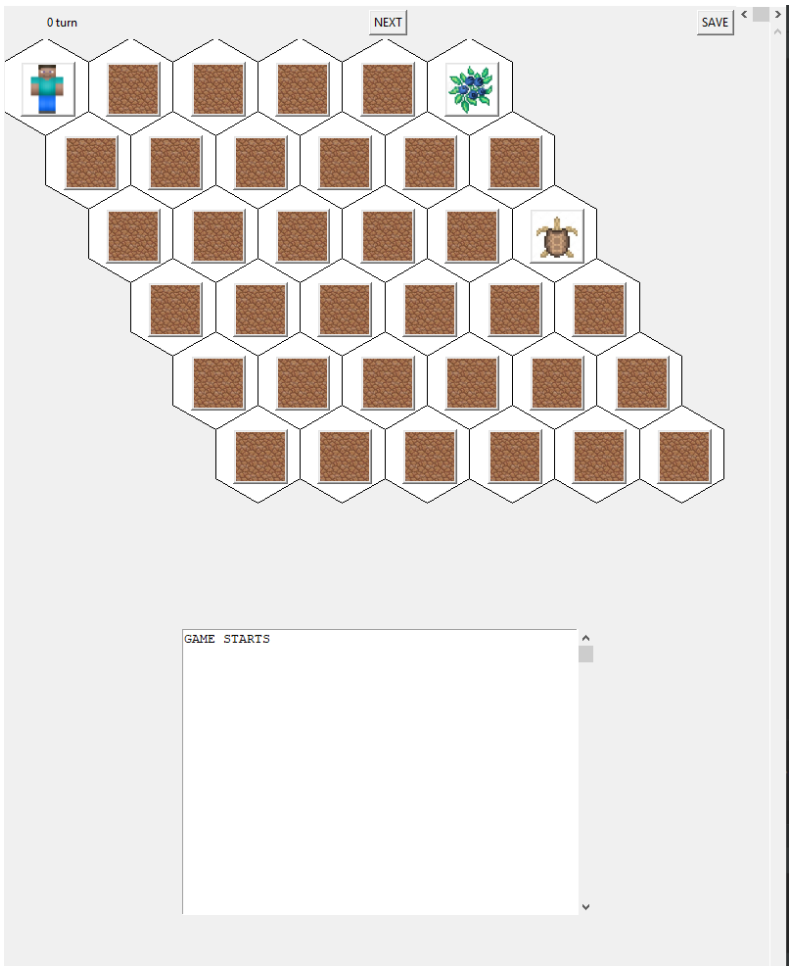
W przypadku podania nieprawidłowych danych wyświetlana jest informacja o błędzie nie pozwalająca przejść dalej.



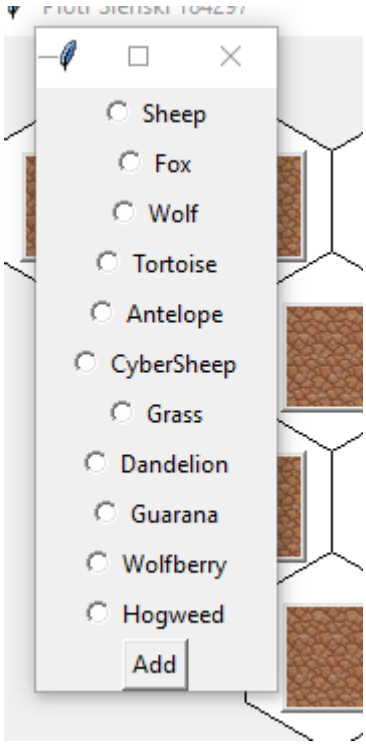
Losowa generacja świata wymaga podania wymiarów oraz ilości organizmów (bez człowieka). Wypełnienie pozostałych pól nie jest potrzebne przy wyborze załadowania świata z pliku.

W głównym oknie aplikacji wyświetlany jest numer tury i przyciski pozwalające na przejście do kolejnej tury bez ruchu człowieka lub zapisanie stanu gry. Wyświetlane są również logi oraz plansza.





Po kliknięciu na pole nie zawierające żadnego organizmu wyświetlane jest okno pozwalające na dodanie dowolnego organizmu w tym miejscu (oprócz człowieka)



Przejście do kolejnej tury może również odbywać się poprzez kliknięcie strzałek na klawiaturze, spowoduje to ruch człowieka w zadaną stronę (jeśli człowiek żyje lub po prostu przejście do kolejnej tury w przeciwnym wypadku)

W przypadku dużych światów możliwe jest przewijanie treści wyświetlanej na ekranie w obie strony.

Generacja świata

Jako że świat w przypadku planszy kwadratowej jak i sześciokątnej jest reprezentowany przez 2 wymiarową tablicę (listę) możliwa jest implementacja obu metod generacji świata w klasie World, gdyż mogą być takie same dla obu typów świata

Losowa generacja świata

Generacja przebiega w następujacy sposób:

- Dodawany jest człowiek na pozycji (0, 0)
- Dla wszystkich klas implementujących organizmy obliczana jest suma prawdopodobieństw generacji każdego organizmu (podstawowo wartość ustawiona na 1 dla każdego organizmu z wyjątkiem człowieka dla którego jest to 0 - każdy organizm ma równą szansę na generację)

- Po obliczeniu sumy generowane jest n losowych organizmów (n podane przy tworzeniu świata)
    - Odbywa się to poprzez wylosowanie liczby z przedziału (0, suma prawdopodobieństw) i iterację po tablicy wszystkich organizmów zwiększając licznik (który startuje od 0), jeśli wykryte zostanie że licznik jest równy bądź większy od wylosowanej liczby generowany jest organizm, którego prawdopodobieństwo generacji zostało ostatnio dodane do licznika.
- Każdy z wygenerowanych organizmów jest dodawany na losowej wolnej pozycji
    - Wykorzystywana jest funkcja find_free_adjacent z ostatnim parametrem True, co oznacza szukanie wolnego miejsca na wszystkich polach dookoła wylosowanego pola do momentu aż nie zostanie znalezione pole lub wyszukiwanie nie napotka na kraniec mapy

```python
def _populate_world(self, n):
    # add
    human = Human((0, 0), self, self._turn_count)
    self._add_organism(human, (0, 0))
    probabilities_sum = 0
    # iterate through all existing organism subclasses and count sum of probablilities of generating each one
    for cls in Organism.__subclasses__():
        for sub in cls.__subclasses__():
            probabilities_sum += sub.generation_probability
    for i in range(n):
        self._generate_random_organism(probabilities_sum)

def _generate_random_organism(self, probabilities_sum):
    found = False
    count = 0
    # get random number in range 0 - sum of all generation probabilities
    # and decide which organism should be added based on that random number
    rand = random.randint(0, probabilities_sum)
    for cls in Organism.__subclasses__():
        for sub in cls.__subclasses__():
            count += sub.generation_probability

            if count >= rand:
                self._add_at_random_pos(sub)
                found = True
                break

        if found is True:
            break

def _add_at_random_pos(self, organism_class):
    x = random.randint(0, (self._size[0] - 1))
    y = random.randint(0, (self._size[1] - 1))
    if self._map[x][y] is None:
        self._add_organism(organism_class((x, y), self, self._turn_count), (x, y))
```

Ładowanie i Zapis świata z/do pliku

Plik zapisu świata wygląda w nastepujący sposób :

Save.txt:

ILOŚĆ ORGANIZMÓW

(SZEROKOŚĆ PLANSZY, WYSOKOŚĆ PLANSZY)

Lista organizmów w formacie:

(pozycja.x, pozycja.y) Nazwa klasy

```
12
(12, 8)
(0, 0) Human
(3, 4) Dandelion
(2, 1) Hogweed
(11, 6) Wolfberry
(3, 5) Fox
(10, 7) Wolf
(11, 2) Guarana
(11, 4) Fox
(8, 4) Dandelion
(6, 7) Fox
(1, 4) Dandelion
(7, 0) Wolfberry
```

Zapis i ładowanie świata realizują poniższe funkcje

```
def _save_world(self):
    file = open("save.txt", "wt")
    file.write(str(len(self._organisms)) + "\n")
    file.write(str(self._size) + "\n")
    for organism in self._organisms:
        file.write(str(organism._position) + " " + str(type(organism).__name__) + "\n")

    file.close()
```

```python
def _load_world(self):
    file = open("save.txt", "rt")
    count = int(file.readline())
    size_line = file.readline()
    # offset is set in order to correctly read 2 digit size
    offset = 0
    if size_line[2] != ',':
        offset += 1
    x = size_line[1:2+offset]

    if size_line[5+offset] != ')':
        y = size_line[4+offset:5+offset+1]
    else:
        y = size_line[4 + offset]

    self._size = (int(x), int(y))

    self._map = [[None for j in range(int(self._size[1]))] for i in range(int(self._size[0]))]

    offset = 0
    for i in range(count):
        o = file.readline()
        # format of data is always (x_pos, y_pos) class name       \n
        #                           ^       ^     ^               ^
        #                        o[1]     o[4]   begins with o[7]  o[-1]
        if o[2] == ',':
            x_pos = int(o[1])
        else:
            x_pos = int(o[1:2])
            offset += 1

        if o[5] == ')':
            y_pos = int(o[4+offset])
        else:
            y_pos = int(o[4+offset:5+offset])
            offset += 1

        position = (x_pos, y_pos)
        cls = o[7+offset:-1]
        instance = globals()[cls]
        self._add_organism(instance(position, self, self._turn_count), position)
        offset = 0
    file.close()
```

Przebieg tury

Przy każdym przejściu do kolejnej tury wywoływana jest metoda update(), w której sortowana jest lista organizmów, zmieniana jest wartość numeru tury wyświetlanego na górze interfejsu oraz wywoływana jest funkcja update_loop(), która wywołuje metodę action() dla każdego organizmu na planszy i zapisuje rezultat zwrócony przez metodę akcji.

Następnie rozpatrywany jest rezultat akcji:

- Jeśli aktualnie wywoływany organizm jest zwierzęciem i rezultat jest koordynatami, znaczy to że zwierzę wykonało ruch, jest on rozpatrywany w metodzie handle_move()
- Jeśli aktualnie wywoływany organizm jest rośliną istnieją dwie możliwości (oprócz zwrócenia None – nie wykonania akcji)
    - Zwrócona wartość jest krotką - tego typu wartość zwracana jest tylko przez Barszcz sosnowskiego, gdzie pierwsza wartość oznacza ile razy roślina rozmnożyła się, a druga jaki jest jej zasięg trucizny.

W zwróconym zasięgu zabijane są wszystkie organizmy w metodzie spread_poison. Dalej rozpatrywana jest możliwość rozmnożenia tak jak dla innych roślin.

- Zwrócona wartość jest liczbą - oznacza ona ile razy dana roślina rozmnożyła się - tyle razy wywoływana jest metoda multiply_organism()

Rozpatrywanie ruchu:

```
def _handle_move(self, organism, new_pos):
    # handle move and collisions occuring after this move
    self.log(organism.to_string() + " moves to " + str(new_pos))

    if self._map[new_pos[0]][new_pos[1]] is None:
        self._move_organism(new_pos, organism)
    else:
        collision_result = self._map[new_pos[0]][new_pos[1]].collision(organism)

        if collision_result == CollisionResult.DIED:
            self.handle_collision_death(new_pos, organism)

        elif collision_result == CollisionResult.KILLED_OPPONENT:
            self._handle_collision_kill(new_pos, organism)

        elif collision_result == CollisionResult.MULTIPLIED and organism.can_breed():
            self._handle_collision_multilpification(new_pos, organism)

        elif collision_result == CollisionResult.ESCAPED:
            self._handle_collision_escape(new_pos, organism)

        elif collision_result == CollisionResult.OPPONENT_ESCAPED:
            self._handle_collision_attacker_escape(new_pos, organism)

        elif collision_result == CollisionResult.BOTH_DIED:
            self._handle_collision_death_both(new_pos, organism)

        else:
            pass
```

Jeśli pole na które ruch zwróciło zwierzę z metody akcji jest puste – wystarczy wykonać ruch. W przeciwnym wypadku wywoływana jest metoda kolizji dla organizmu stojącego na danym polu i rozpatrywany jest wynik tej kolizji.

Wykonywanie tury realizują poniższe funkcje:

```python
def _update(self):
    self._organisms.sort(key=lambda o: (o.get_initiative(), -o.get_creation_turn()), reverse=True)
    self._turn_count += 1
    self._update_turn_display()

    self._update_loop()

    self._activated_human_ability = False
    self.log("---\n" + "Turn number " + str(self._turn_count))

def _update_loop(self):
    result = None
    for organism in self._organisms:
        result = organism.action()
        if isinstance(organism, Animal) and type(result) is tuple:
            self._handle_move(organism, result)

        elif isinstance(organism, Plant):
            if type(result) is tuple:
                if organism.is_poisonous():
                    poison_radius = result[1]
                    self._spread_poison(organism, poison_radius)

                result = result[0]

            for i in range(result):
                self.log(type(organism).__name__ + " spread")
                self._multiply_organisms(organism)

def _update_turn_display(self):
    info_str = str(self._turn_count) + " turn"
    lbl_info = tk.Label(master=self._window, text=info_str)
    lbl_info.grid(row=0, column=0)
```

Wyświetlanie świata

Świat jest wyświetlany w metodzie display() wywoływanej po pobraniu wszelkich potrzebnych danych dla generacji świata. Metoda ta wywoływana jest tylko raz, a zmiany na mapie realizowane są poprzez funkcję wyświetlania organizmów (np. Sheep.display()) oraz funkcję add_ground() dodającą puste pole we wskazanym miejscu.

Każde wyświetlane pole jest przyciskiem (tkinter.Button())

Najpierw tworzone jest okno i dodawane są scrollbary, następnie wywoływana jest metoda display_map() różna dla obu typów świata oraz dodawane jest okno wyświetlające logi.

```
def _display(self):

    self._crate_main_window()

    self._display_map()

    self._add_movement_binds()

    self._add_logs_frame()

    self._root_frame.pack(side="left", fill="both", expand=1)

    self._window.configure(width=self._root.winfo_reqwidth())

    self._root.mainloop()
```

W metodach display_map dodawane są przyciski przejścia do następnej tury i zapisu oraz rysowana jest plansza. Różni te metody sposób umieszczania elementów w oknie oraz miejsce gdzie wyświetlane są przyciski. W metodzie świata sześciokątnego dodatkowo jest dodawane tło reprezentujące planszę.

Rysowanie mapy w świecie sześciokątnym:

```
def _display_map(self):
    self._update_turn_display()
    btn_new_turn = tk.Button(master=self._window, text="NEXT", command=self._update)
    btn_new_turn.grid(row=0, column=1, sticky="W")
    btn_save = tk.Button(master=self._window, text="SAVE", command=self._save_world)
    btn_save.grid(row=0, column=2, sticky="W")
    y_range = self._size[1]
    x_range = self._size[0]
    for y in range(y_range):
        for x in range(x_range):
            if self._map[x][y] is None:
                self._add_ground((x, y))
            else:
                self._map[x][y].display(self._window)
```

Wyświetlanie organizmów:

```
def display(self, window):
    if self._world.is_hex():
        self._display_at_position(window)
    else:
        self._display_at_grid(window)


def _display_at_grid(self, window):
    # display method adds button at organism position
    x = self._position[0]
    y = self._position[1] + Constants.MAP_Y_OFFSET
    self.icon_button = tk.Button(master=window, image=self._icon)
    self.icon_button.grid(row=y, column=x)

def _display_at_position(self, window):
    x = self._position[0]
    y = self._position[1]
    center = self._world.get_hex_center((x, y))
    self.icon_button = tk.Button(master=window, image=self._icon)

    a = self._world.get_hex_width()/2
    rad = self._world.get_hex_radius()

    # calcualte where to place the button - left top corner of hex and in  a - rad / 2 distance from it
    point = (center[0] - a, center[1] - (0.5 * rad))
    self.icon_button.place(x=point[0] + a - rad / 2, y=point[1]+30)
```

Wyświetlanie w świecie sześciokątnym:

Obliczenia w dwóch ostatnich liniach _display_at_position wyznaczają punkt umieszczenia ikony. Point jest to wierzchołek sześciokata (lewy górny), a dalsze równanie wyznacza przesunięcie względem tego wierzchołka