Maciej Luśnia

Grupa I1S 1.5/9

Charakterystyka kodu autorskiej gry

Przedmiotem niniejszego referatu jest charakterystyka kodu autorskiej gry. „Miecze i kalosze”, bo tak się nazywa owa gra, powstała jako proste zajęcie w czasie wolnym służące do przećwiczenia programowania w języku Python, która ze zwykłej aplikacji konsolowej przekształciła się w pełnoprawną, okienkową grę graficzną. Dzięki pasji i zaangażowaniu, mogę przedstawić i scharakteryzować kod mojej pierwszej autorskiej gry.

Kod mojej gry rozpoczynam od zaimportowania bibliotek. Stworzenie okna z grą oraz wyświetlanie w nim generowanej grafiki jest możliwe dzięki bibliotece „pygame”, której elementy będę wykorzystywał na każdym kroku programu. Funkcja „exit” z biblioteki „sys” jest wykorzystana do prawidłowego zamknięcia programu, natomiast funkcje „sleep”, „randint” oraz „ceil” zostaną wykorzystane do różnych obliczeń w późniejszej części programu. Następnie dzięki trzem funkcjom z biblioteki pygame wywołane jest okno programu o wymiarach 1280x720 pikseli nazwane tytułem gry. Zmienna „czarny” zawiera odpowiednie temu kolorowi wartości w formacie RGB, które posłużą do generowania tekstu. Zmienna „tekst” zawiera specjalną czcionkę, która wpasowuje się w pikselowy styl graficzny gry. Jest przechowywana w zmiennej, aby nie było potrzeby ładować jej z pliku za każdym razem, gdy będzie potrzebna. W celu lepszej organizacji kodu, pozostałe zmienne inicjuję w klasach, dzięki czemu w łatwy sposób mogę przechować potrzebne obrazy, takie jak przyciski, tła grafiki gracza, dźwięki oraz

```python
import pygame
from pygame.locals import *
from sys import exit
from time import sleep
from random import randint
from math import ceil

pygame.init()
screen=pygame.display.set_mode((1280,720))
pygame.display.set_caption('Miecze i Kalosze')

czarny=(0,0,0)
tekst=pygame.font.Font('PixeloidMono.ttf',20)
```

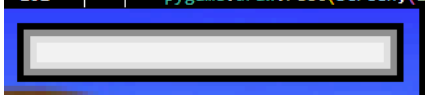
Rysunek 1 Inicjalizacja bibliotek i zmiennych roboczych

```python
class PNG():
    gracz=[pygame.image.load('gracz/idle1.png'), ···
    atak=[pygame.image.load('gracz/atak1.png'), ···
    trafienie=[pygame.image.load('atak/atak1.png'), ··
    class Przycisk(): ···
    przycisk=Przycisk()
    class Tlo(): ···
    tlo=Tlo()
    moneta=pygame.image.load('moneta.png')
    potka=pygame.image.load('potka.png')

class MP3():
    atak1=pygame.mixer.Sound("mp3/atak1.wav")
    atak2=pygame.mixer.Sound("mp3/atak2.wav")
    mikstura=pygame.mixer.Sound("mp3/mikstura.wav")
    przycisk=pygame.mixer.Sound("mp3/przycisk.wav")

class Gracz():
    hp=int(60)
    maxhp=int(60)
    zloto=int(20)
    poziom=int(1)
    exp=int(0)
    lvlup=int(125)
    potki=int(0)
    minatak=int(0)
    maxatak=int(7)
    pmiecza=int(0)
    pzbroi=int(0)
    szansa=int(100)

class stats_opp():
    hp=0
    maxhp=0
    atkmin=0
    atkmax=0
    potki=0
    szansa=0
    tura=0
    trudnosc=0
```

Rysunek 2 Kolejne zmienne robocze

statystyki gracza i przeciwnika, które zostaną wykorzystane w późniejszej części programu. Grafiki ładuję za pomocą funkcji „pygame.image.load()", natomiast odgłosy przez „pygame.mixer.Sound()". Zanim przejdę jednak do kodu, definiuję jeszcze jedną funkcję – textbox(). Posłuży mi ona do tworzania wizualnych okienek tekstowych poprzez wygenerowanie kilku prostokątów o różnych kolorach, tak jak na rysunku 3.

```
97  def textbox(x,y,a,b,w):
98      pygame.draw.rect(screen,(0,0,0),(x+w*0.5,y+w*0.5,a-w,b-w),w)
99      pygame.draw.rect(screen,(143,143,143),(x+w*1.5,y+w*1.5,a-w*3,b-w*3),w)
100     pygame.draw.rect(screen,(224,224,224),(x+w*2.5,y+w*2.5,a-w*5,b-w*5),w)
101     pygame.draw.rect(screen,(242,242,242),(x+w*3.5,y+w*3.5,a-w*7,b-w*7))
```



Rysunek 3 Funkcja textbox() oraz przykładowe wygenerowane przez nią okienko tekstowe

Po wszystkich przygotowaniach przejdę do kodu, który generuje obraz gry. Ten również został podzielony na klasy w celu zwiększenia czytelności kodu. Każdy obszar w grze ma swoją klasę, w której znajduje się cały kod odpowiedzialny za działanie danego obszaru. Na rysunku 4 została przedstawiona klasa Wstep(), która odpowiada za ekran tytułowy. Przy inicjalizacji każdej klasy tworzę zmienne, które przechowują współrzędne generowanych przycisków na

```
103 class Wstep():
104     def __init__(self):
105         self.przycisk_start=(550,560)
106
107     def render(self):
108         screen.fill((0,0,0))
109         screen.blit(grafika.tlo.menu,(0,0))
110         screen.blit(tekst.render('Witaj w Mieczach i Kaloszach!',True,czarny),(420,140))
111         screen.blit(tekst.render('Jesteś nowym gladiatorem.',True,czarny),(420,220))
112         screen.blit(tekst.render('Twoim zadaniem jest pokonać ',True,czarny),(420,270))
113         screen.blit(tekst.render('bossa areny. Zajdź do sklepu',True,czarny),(420,300))
114         screen.blit(tekst.render('po lepsze wyposażenie,',True,czarny),(420,330))
115         screen.blit(tekst.render('bez niego nie przetrwasz',True,czarny),(420,360))
116         screen.blit(tekst.render('na arenie.',True,czarny),(420,390))
117         screen.blit(tekst.render('Wychodzisz na główną ulicę',True,czarny),(420,440))
118         screen.blit(tekst.render('miasta, gotowy rozpocząć',True,czarny),(420,470))
119         screen.blit(tekst.render('swoją przygodę.',True,czarny),(420,500))
120         screen.blit(grafika.przycisk.start,self.przycisk_start)
121
122     def buttons(self):
123         global obszar
124         kursor=pygame.mouse.get_pos()
125         przycisk=(kursor[0]-self.przycisk_start[0],kursor[1]-self.przycisk_start[1])
126         if 0<=przycisk[0]<=grafika.przycisk.start.get_width() and 0<=przycisk[1]<=grafika.przycisk.start.get_height():
127             pygame.mixer.Sound.play(dzwiek.przycisk)
128             obszar='miasto'
```

Rysunek 4 Klasa Wstep()

danym obszarze. W taki sposób nie tylko zwiększa czytelność kodu, lecz także ułatwia aktualizowanie gry. Gdy w przyszłości przy aktualizacji gry będę potrzebował przenieść przycisk w inne miejsce na ekranie, wystarczy zmienić współrzędne w jednej zmiennej, aby przycisk został przeniesiony bez utraty funkcjonalności. W każdej klasie obsługa generowania grafiki i funkcjonalności przycisków została podzielona na oddzielne funkcje, gdyż w obszarach bez animacji grafikę wystarczy wygenerować tylko raz, a przyciski muszą działać dopóki nie zostaną wciśnięte, zatem w ten sposób zwiększa się prędkość działania gry, gdyż

stała grafika nie jest czyszczona i generowana w nieskończoność. Renderowanie zaczynam od metody „fill”, aby wyczyścić okno i zastąpić je czarną planszą. Następnie za pomocą metody „blit” wyświetlam tło, napisy oraz różne grafiki. Funkcje przycisków również są bardzo proste. Za pomocą metody pygame.mouse.get_pos() przypisuję do zmiennej kursor pozycję kursora na ekranie. Następnie w zmiennej przycisk obliczam pozycję kursora względem lewego górnego rogu przycisku, które zostają porównane z wymiarami przycisku. Jeżeli te wartości zawierają się w danych przedziałach, przycisk zostaje aktywowany. Większość przycisków, takich jak ten na ekranie tytułowym zmienia obszar gracza poprzez zmianę globalnej zmiennej „obszar”, wyrenderowanie obszaru, do którego przechodzi gracz, oraz odtworzenie odgłosu kliknięcia przy pomocy funkcji pygame.mixer.Sound.play(). Większość przycisków w grze ma identyczną funkcję wykonywaną w identyczny sposób, dlatego będę zwracał uwagę na te, które mają inne działanie niż zmiana obszaru.

Po ekranie tytułowym przechodzimy do miasta. Zanim jednak będę mógł opisać kod tego obszaru, musimy spojrzeć na główną pętlę gry, gdyż jest ona kluczowa dla działania całej gry. Na rysunku 5 przedstawione są zmienne, które są inicjowane przed samą petlą, czyli zmienne inicjujące klasy ze wszystkimi obszarami gry, zmienna „obszary”, która jest słownikiem wszystkich funkcji obsługujących przyciski, zmienna „obszar”, w której przechowywana jest nazwa obszaru, w której znajduje się gracz, oraz zmienne „anim” i „klatki”, które będą potrzebne do obsługi animacji w mieście. Przechodzimy teraz do głównej pętli, która jest przedstawiona na rysunku 6. Na początku zostaje jednorazowo wyrenderowany ekran tytułowy, po czym interpreter wchodzi do pętli. Funkcja pygame.Time.Clock().tick() ustala z jaką prędkością ma działać program, gdzie wartość w nawiasie to liczba klatek na sekundę, w tym przypadku jest to 120. Pętla for, która po niej występuje, wykorzystuje funkcję pygame.event.get(), która zawiera wszystkie akcje, jakie użytkownik podjął, między innymi zamknięcie programu czy kliknięcie przycisku myszy.

```
976  gracz=Gracz()
977  wstep=Wstep()
978  miasto=Miasto()
979  tawerna=Tawerna()
980  sklep=Sklep()
981  statystyki=Statystyki()
982  arena=Arena()
983  walka=Walka()
984  opp=stats_opp()
985  grafika=PNG()
986  dzwiek=MP3()

1001  obszar='wstep'
1002  anim=0
1003  klatki=0
```

```
987  obszary={'wstep': wstep.buttons,
988           'miasto': miasto.buttons,
989           'tawerna': tawerna.buttons,
990           'sklep': sklep.buttons,
991           'mikstura': sklep.mikstura_buttons,
992           'miecz': sklep.miecz_buttons,
993           'zbroja': sklep.zbroja_buttons,
994           'statystyki': statystyki.buttons,
995           'arena': arena.wejscie_buttons,
996           'start': arena.start_buttons,
997           'wygrana': walka.wygrana_buttons,
998           'koniec': walka.koniec_buttons,
999           'przegrana': walka.przegrana_buttons}
```

Rysunek 5 Zmienne inicjujące grę

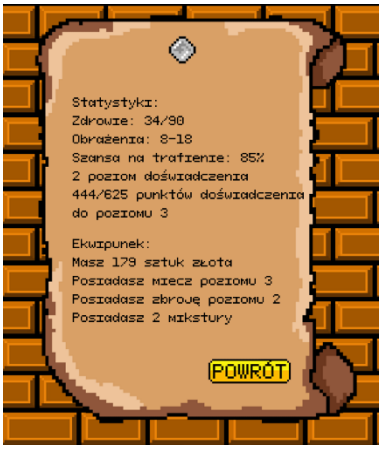
```
1005  wstep.render()
1006  while True:
1007      pygame.time.Clock().tick(120)
1008      for event in pygame.event.get():
1009          if event.type==QUIT:
1010              pygame.quit()
1011              exit()
1012
1013          if event.type==pygame.MOUSEBUTTONDOWN:
1014              obszary[obszar]()
1015
1016      if obszar=='miasto':
1017          miasto.render()
1018          if anim==len(grafika.gracz):
1019              anim=0
1020          sprite=pygame.transform.scale(pygame.transform.flip(grafika.gracz[anim],True,False),(230,230))
1021          screen.blit(sprite,(500,400))
1022          klatki+=1
1023          if klatki==8:
1024              anim+=1
1025              klatki=0
1026      pygame.display.update()
```

Rysunek 6 Główna pętla programu

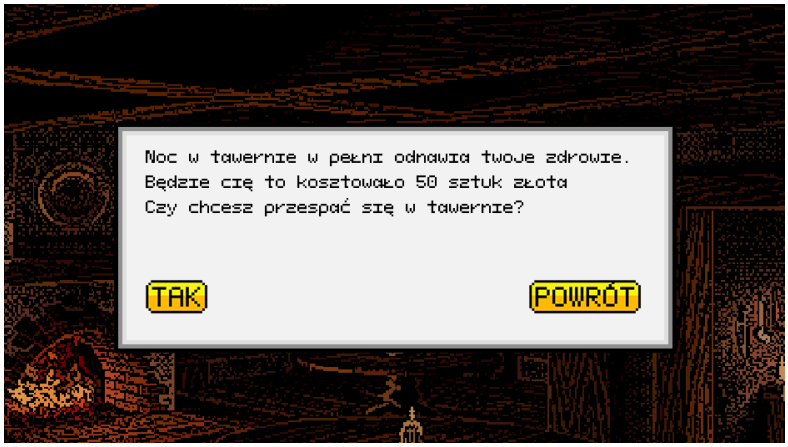
Dokładnie te akcje są sprawdzane w następnych instrukcjach warunkowych, aby prawidłowo wyłączyć grę przy zamknięciu okna, lub też prawidłowo obsłużyć odpowiedni przycisk przy kliknięciu myszą. Wykorzystywany jest tu zainicjowany wcześniej słownik „obszary", aby wywołać odpowiednią funkcję przycisków w zależności od obszaru. W pętli jest również renderowany obszar miasta, gdyż jest to jeden z dwóch obszarów w grze gdzie występują animacje. Na początku program wyświetla tło i grafiki przycisków, po czym przechodzi do obsługi animacji. Animacje polegają na wyświetlaniu po sobie wielu zdjęć nieznacząco różniących się od siebie w krótkich odstępach czasu, co dla ludzkiego oka sprawia wrażenie ruchu. Zmienna „anim" przechowuje indeks zdjęcia do wyświetlenia, a zmienna „klatki" z drugą zmienną warunkową sprawia, że co 8 klatek wyświetlane jest inne zdjęcie. Sama grafika gracza jest jednak wcześniej zmieniona przez dwie metody: „pygame.transform.flip", która obraca grafikę w pionie, oraz „pygame.transform.scale", która ustala odpowiedni rozmiar zdjęcia. Ostatnim krokiem w pętli jest funkcja „pygame.display.update", która wyświetla wszystkie załadowane grafiki w oknie gry. Takim sposobem otrzymujemy obraz miasta przedstawiony na rysunku 7. Podczas renderowania miasta wykorzystuję jeszcze dwie funkcje: „pygame.mixer.music.load" oraz „pygame.mixer.music.play".

Rysunek 7 Obszar miasto

Dzięki nim, gdy gracz przebywa w mieście odtwarzana jest muzyka, a dokładnie dźwięk tłumu w mieście. Przy naciśnięciu któregokolwiek z przycisków muzyka jest zatrzymywana przy pomocy funkcji „pygame.mixer.music.stop”. Z miasta użytkownik może przejść do kilku różnych obszarów: ekranu statystyk, gdzie wyświetlane są statsytyki postaci gracza (rysunek 8), tawerny, gdzie możliwe jest uleczenie postaci (rysunek 9), sklepu, gdzie można kupić lepsze wyposażenie (rysunek 10) lub areny, gdzie można walczyć z innymi wojownikami (rysunek 11).



Rysunek 8 Ekran statystyk postaci



Rysunek 9 Wygląd tawerny

Rysunek 10 Wygląd sklepu

Rysunek 11 Wygląd wejścia na arenę

Ekran statystyk działa w identyczny sposób jak ekran tytułowy, więc nie będę się na nim zatrzymywał. Tawerna działa jednak inaczej niż poprzednie obszary. Tutaj pierwszy raz wykorzystuję przywołaną wcześniej funkcję „textbox()”, dzięki której mogę wyrożnić tekst na każdym tle za pomocą okna tekstowego. Przycisk „TAK” na tym ekranie również ma nową funkcjonalność. Po kliknięciu go, sprawdza czy gracz ma maksymalną wartość zdrowia.

```
200                 if gracz.hp==gracz.maxhp:
201                     screen.blit(tekst.render('Jesteś już w pełni uzdrowiony!',True,czarny),(340,280))
202                 elif gracz.zloto<50:
203                     screen.blit(tekst.render('Nie masz wystarczająco dużo złota!',True,czarny),(340,280))
204                 else:
205                     gracz.hp=gracz.maxhp
206                     gracz.zloto-=50
207                     screen.blit(tekst.render('Zostałeś w pełni uzdrowiony!',True,czarny),(340,280))
208                     pygame.mixer.Sound.play(dzwiek.mikstura)
209                 pygame.display.update()
210                 sleep(1)
211                 tawerna.render()
```

Rysunek 12 Przycisk „TAK” w obszarze Tawerna

Jeśli tak, powstrzymuje go przed wydaniem złota z odpowiednim komunikatem. Przy niewystarczającej ilości złota gracz również dostaje taki komunikat. Dopiero, gdy nie ma tych dwóch przeszkód, program zmienia odpowiednie wartości w zmiennej gracz, po czym daje komunikat o powodzeniu akcji i odtwarza dźwięk leczenia. Niezależnie od wyniku komunikatu, który zostaje wyświetlony, zostaje wykonana funkcja „sleep” z biblioteki „time”, która sprawia, że interpreter czeka odpowiednią ilość czasu z dalszym wykonywaniem programu, w tym wypadku sekundę. Dzięki temu gracz ma odpowiednią ilość czasu, żeby przeczytać wyświetlony komunikat. Po upłynięciu sekundy wykonana zostaje funkcja, która renderuje pierwotny ekran tawerny, tj. bez komunikatu. Warto zwrócić uwagę na to, że renderowanie grafiki w obszarach bez animacji odbywa się jednorazowo przy kliknięciu przycisku zmiany obszaru, a nie jak w przypadku sklepu w pętli.

Kolejnym obszarem, do którego gracz może zajść, jest sklep. Renderowanie tego ekranu odbywa się w dwóch etapach. Pierwszym etapem jest „podstawa”, czyli podstawowe elementy, które są renderowane w każdym podobszarze sklepu. Zalicza się do nich tło, „textbox” będący podstawą pod każdy tekst, oraz grafika informująca o posiadanej ilości złota. Przyciski wyboru przedmiotu oraz powrotu działają tak jak na ekranie tytułowym.

```
234     def render_baza(self):
235         screen.fill((0,0,0))
236         screen.blit(grafika.tlo.sklep,(0,0))
237         textbox(310,110,530,330,5)
238         textbox(10,660,110,55,5)
239         screen.blit(grafika.moneta,(29,679))
240         screen.blit(tekst.render(f'{gracz.zloto}',True,czarny),(50,675))
```

Rysunek 13 Podstawa grafiki sklepu

Kupowanie mikstur również ma podobny kod do przycisku w tawernie, jedyną różnicą jest zmiana zmiennej z ilością mikstur zamiast ze zdrowiem. Kupowanie miecza lub zbroi działa jednak w trochę inny sposób. Ustaliłem maksymalny poziom tych przedmiotów na 4. Jeżeli gracz posiada miecz lub zbroję niższego poziomu niż maksymalny, pokaże się komunikat o statystykach i cenie przedmiotu poziomu wyższego. Pokaże się również przycisk z opcją zakupu. Jeśli gracz ma jednak dany przedmiot poziomu 4, pokaże się jedynie komunikat o maksymalnym poziomie i przycisk powrotu do wyboru przedmiotu. Jest to możliwe dzięki

zastosowaniu instrukcji warunkowej if sprawdzającej poziom danego przedmiotu. Na rysunku 14 przedstawiony jest kod obsługujący przycisk do zakupu miecza.

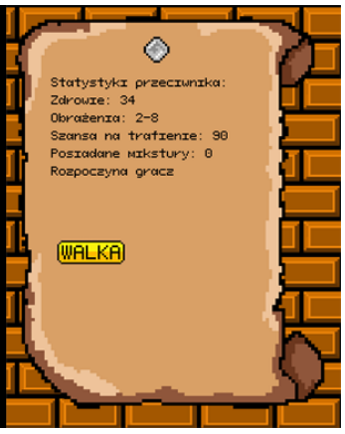
```
331             if gracz.pmiecza<4:
332                 przycisk=(kursor[0]-self.miecz_przycisk_tak[0],kursor[1]-self.miecz_przycisk_tak[1])
333                 if 0<=przycisk[0]<=grafika.przycisk.tak.get_width() and 0<=przycisk[1]<=grafika.przyc
334                     if gracz.zloto<self.miecz_koszt:
335                         screen.blit(tekst.render('Nie masz wystarczająco dużo złota!',True,czarny),(3
336                     else:
337                         gracz.zloto-=self.miecz_koszt
338                         gracz.pmiecza+=1
339                         gracz.minatak+=2
340                         gracz.maxatak+=3
341                         gracz.szansa-=5
342                         self.miecz_koszt=int(125*pow(2,gracz.pmiecza))
343                         self.miecz_render()
344                         screen.blit(tekst.render(f'Kupiłeś miecz poziomu {gracz.pmiecza}!',True,czarn
345                         pygame.mixer.Sound.play(dzwiek.mikstura)
346                     pygame.display.update()
347                     sleep(1)
348                     sklep.miecz_render()
```

Rysunek 14 Kod obsługujący przycisk do zakupu miecza

Jak można zauważyć, jeśli poziom miecza gracza jest maksymalny, przycisk ten jest zupełnie pomijany przez interpreter. Warto zwrócić również uwagę na zmienną ustalającą koszt miecza następnego poziomu w linijce 342. Wartość tą ustaliłem na 10 przy inicjacji klasy „Miecz”, aby gracz był w stanie kupić przed pierwszą walką miecz i zbroję. Koszt następnych poziomów zwiększa się wykładniczo, aby osiągnięcie maksymalnego poziomu nie stało się zbyt szybko i dawało satysfakcję. Komunikat przy zakupie a następnie renderowanie sklepu działa identycznie jak w tawernie. Identyczny kod jest też przy zakupie zbroi, dlatego przejdę teraz do bardziej skomplikowanego obszaru.

Miejscem, któremu jest poświęcona największa część kodu, jest arena. Po wejściu na nią pokazuje się lista dostępnych przeciwników. W grze są trzy stopnie trudności przeciwników: normalny, trudny, którego pokonanie daje większe nagrody, oraz „boss”, którego pokonanie jest celem gry. Każdy stopień trudności jest dostępny na konretnych poziomach: normalny na poziomach 1-2, trudny – 2-3, boss – 3-4. Po wybraniu przeciwnika odpowiednim przyciskiem gra losuje statystyki przeciwnika według wybranej trudności dzięki funkcji „randint” z biblioteki „random”. Wyjątkiem jest „boss”, którego statystyki są z góry ustalone. Zmienna „tura” przetrzymuje informację o tym, czy aktualnie jest tura gracza, czy przeciwnika, gdzie 1 oznacza przeciwnika, a 0 gracza. Na początku jest losowana, aby ani gracz, ani przeciwnik nie miał przewagi. Po ustaleniu statystyk program wyświetla je graczowi. Znowu wykorzystana jest metoda blit(), tak jak do większości tekstów

```
493             if opp.trudnosc==1:
494                 opp.hp=randint(30,40)
495                 opp.atkmin=randint(1,2)
496                 opp.atkmax=randint(7,9)
497                 opp.potki=randint(0,1)
498                 opp.szansa=randint(80,90)
499             elif opp.trudnosc==2:
500                 opp.hp=randint(80,100)
501                 opp.atkmin=randint(8,10)
502                 opp.atkmax=randint(16,18)
503                 opp.potki=randint(0,2)
504                 opp.szansa=randint(70,80)
505             else:
506                 opp.hp=125
507                 opp.atkmin=18
508                 opp.atkmax=28
509                 opp.potki=4
510                 opp.szansa=75
511             opp.maxhp=opp.hp
512             opp.tura=randint(0,1)
```



Rysunek 15 Kod ustalający statystyki przeciwnika oraz ekran wyświetlający je graczowi

w grze. Graczowi pokazuje się również przycisk „WALKA”, po klinięciu którego gracz zostaje przeniesiony na pole bitwy.

Ekran walki jest najbardziej dynamicznym obszarem w całej grze. Tło areny jest niezmienne, jednak przez cały czas trwania walki postacie gracza i przeciwnika są animowane. Zanim jednak przejdziemy do samej walki, potrzeba jeszcze zainicjować kilka zmiennych. Pierwsza to „self.czcionka”, w której przechowywana jest czcionka w trzech rozmiarach, która zostanie wykorzystana do renderowania punktów obrażeń lub leczenia podczas walki. Słowo „self” przed nazwą zmiennej potrzebne jest do tego, aby zainicjowana w obrębie klasy zmienna mogła zostać później wykorzystana w różnych funkcjach. W samej funkcji obsługującej walkę powołuję zmienne „animacje” oraz „klatki”, które zostaną wykorzystane w ten sam sposób jak przy renderowaniu miasta, tj. zmienna „animacje” będzie przechowywała indeksy aktualnych zdjęć z animacji do wyświetlenia, natomiast zmienna „klatki” przechowuje liczbę klatek od ostatniej zmiany zdjęcia. Tym razem jednak zmienna „animacje” jest listą dwóch wartości, gdyż poza graczem animowana jest też postać przeciwnika. Podczas walki, tak jak w mieście, odtwarzany jest dźwięk tłumu przy użyciu metod „load” oraz „play”. Argument -1 oznacza, że muzyka będzie odtwarzana w nieskończoność. Po zainicjowaniu zmiennych gra rozpoczyna pętlę walki. Pierwszy krok pętli to sprawdzenie, czy aktualnie jest tura przeciwnika, czy gracza z wcześniej wylosowanej zmiennej „opp.tura”. Zacznę od omówienia kodu przeciwnika. Cała jego logika jest oparta na losowości i generowaniu liczb. Pierwsza instrukcja warunkowa odpowiada za leczenie przeciwnika. Jeżeli przeciwnik ma poniżej 50% zdrowia oraz posiada w ekwipunku miksturę leczenia, to ma 20% szans na to, aby jej użyć. Jak uda mu się spełnić te warunki, to wartość leczenia jest losowana z przedziału od 40% do 60% maksymalnego zdrowia przeciwnika, po czym jest

```
549             self.czcionka=[pygame.font.Font('PixeloidMono.ttf',80),
550                            pygame.font.Font('PixeloidMono.ttf',70),
551                            pygame.font.Font('PixeloidMono.ttf',60)]
552
553         def walka(self):
554             animacje=[randint(0,3),randint(0,3)]
555             klatki=0
556             pygame.mixer.music.load("mp3/tlum.wav")
557             pygame.mixer.music.play(-1)
```

Rysunek 16 Inicjowanie zmiennych używanych podczas wyświetlania walki

```
while True:
    if opp.tura:
        if opp.hp/opp.maxhp<0.5 and opp.potki>0 and randint(1,10)<2:
            leczenie=int(ceil(opp.maxhp*(40+randint(0,20))/100))
            if leczenie>opp.maxhp-opp.hp:
                leczenie=opp.maxhp-opp.hp
            opp.potki-=1
            self.render_leczenie(1,leczenie)
        else:
            akcja=randint(1,3)
            if akcja==1:
                moc_ataku=randint(70,90)/100
                szansa_ataku=randint(15,25)
            elif akcja==2:
                moc_ataku=1
                szansa_ataku=0
            else:
                moc_ataku=randint(110,130)/100
                szansa_ataku=-randint(15,25)
            atak=randint(ceil(opp.atkmin*moc_ataku),ceil(opp.atkmax*moc_ataku))
            if opp.szansa+szansa_ataku<randint(1,100):
                atak=0
            self.render_atak(opp.tura,atak)
        opp.tura=0
```

Rysunek 17 Logika przeciwnika podczas walki

zaokrąglona w górę do liczby całkowitej dzięki funkcji „ceil” z biblioteki „math”. Następnie sprawdzane jest, czy wartość leczenia nie jest większa od maksymalnej liczby punktów zdrowia przeciwnika, aby uniknąć sytuacji, gdzie przeciwnik ma więcej zdrowia niż maksymalna wartość. Po tej instrukcji liczba mikstur przeciwnika jest zmniejszana o 1 i następuje renderowanie leczenia przeciwnika. Jest to możliwe dzięki metodzie „render_leczenie”. Aby uniknąć powtarzania tego samego kodu, stworzyłem również metodę „render_baza”, która wyświetla tło areny, liczbę posiadanych mikstur oraz paski zdrowia. Aktualizują się one przez pomnożenie długości paska zdrowia przez % posiadanego zdrowia przez odpowiednio gracza i przeciwnika. Będzie ona używana przez całą walkę, przez co utworzenie metody bardzo skraca i porządkuje kod. Teraz przejdę do samej metody leczenia. Zaczyna się ona od pętli for, która po renderowaniu bazy wyświetla krótką animację używania mikstury. Następnie zmienione są wartości zdrowia odpowiedniej postaci, po czym zainicjowane są dwie zmienne z wartościami RGB dla kolorów czarnego i zielonego. Zmienne te są następnie wykorzystane

```
def render_baza(self):
    screen.fill((0,0,0))
    screen.blit(grafika.tlo.arena,(0,0))
    screen.blit(grafika.potka,(30,540))
    screen.blit(tekst.render(f'{gracz.potki}',True,czarny),(70,630))
    pygame.draw.rect(screen,czarny,(160,650,400,40),3)
    pygame.draw.rect(screen,(255,0,0),(163,653,394*float(gracz.hp)/float(gracz.maxhp),34))
    screen.blit(tekst.render(f'{gracz.hp}/{gracz.maxhp}',True,czarny),(350,660))

    screen.blit(grafika.potka,(1150,540))
    screen.blit(tekst.render(f'{opp.potki}',True,czarny),(1190,630))
    pygame.draw.rect(screen,czarny,(712,650,400,40),3)
    pygame.draw.rect(screen,(255,0,0),(714,652,394*float(opp.hp)/float(opp.maxhp),36))
    screen.blit(tekst.render(f'{opp.hp}/{opp.maxhp}',True,czarny),(902,660))
```

Rysunek 18 Metoda „render_baza”

```
def render_leczenie(self,leczenie):
    for i in range(4):
        self.render_baza()
        pygame.time.Clock().tick(10)
        if opp.tura:
            screen.blit(pygame.transform.flip(grafika.atak[i],True,False),(680,80))
            screen.blit(grafika.gracz[i%4],(80,80))
        else:
            screen.blit(grafika.atak[i],(80,80))
            screen.blit(pygame.transform.flip(grafika.gracz[i%4],True,False),(680,80))
        pygame.display.update()
    if opp.tura:
        opp.hp+=leczenie
    else:
        gracz.hp+=leczenie
    pygame.mixer.Sound.play(dzwiek.mikstura)
    kolor1=czarny
    kolor2=(0,255,0)
    for i in range(16):
        self.render_baza()
        pygame.time.Clock().tick(10)
        if opp.tura:
            screen.blit(grafika.gracz[i%4],(80,80))
            if i<12:
                screen.blit(pygame.transform.flip(grafika.atak[3],True,False),(680,80))
            else:
                screen.blit(pygame.transform.flip(grafika.gracz[15-i],True,False),(680,80))
        else:
            screen.blit(pygame.transform.flip(grafika.gracz[i%4],True,False),(680,80))
            if i<12:
                screen.blit(grafika.atak[3],(80,80))
            else:
                screen.blit(grafika.atak[15-i],(80,80))
        if opp.tura:
            screen.blit(self.czcionka[0].render(f'{leczenie}',True,kolor1),(900,160))
            screen.blit(self.czcionka[1].render(f'{leczenie}',True,kolor2),(904,167))
            screen.blit(self.czcionka[2].render(f'{leczenie}',True,kolor1),(908,174))
        else:
            screen.blit(self.czcionka[0].render(f'{leczenie}',True,kolor1),(300,160))
            screen.blit(self.czcionka[1].render(f'{leczenie}',True,kolor2),(304,167))
            screen.blit(self.czcionka[2].render(f'{leczenie}',True,kolor1),(308,174))

        kolor1,kolor2=kolor2,kolor1
        pygame.display.update()
```

Rysunek 19 Metoda „render_leczenie”

w krótkiej animacji, która wyświetla nad leczonym graczem wartość uleczonych punktów zdrowia. Zostało to przedstawione na rysunku 20. Wracając do kodu przeciwnika, jeśli nie uda się spełnić przyciwnikowi wszystkich warunków do leczenia, przystępuje do ataku. W takiej sytuacji losuje on liczbę od 1 do 3 oznaczającą moc ataku. 1 to atak szybki, 2 – normalny, a 3 – mocny. Szybki atak zadaje około 80% obrażeń, ale ma około 20% więcej szansy na trafienie, normalny atak wykorzystuje podstawowe statystyki przeciwnika, a atak mocny zadaje około 120% obrażeń, ale ma 20% mniej szansy na trafienie. Wartości dla szybkiego i mocnego ataku są przybliżone, gdyż w rzeczywistości też są losowane. Moc ataku może mieć +- 10% od podanej wartości, a szansa na atak +-5%. Dzięki temu gra jest mniej przewidywalna, a co za tym idzie ciekawsza. Po wylosowaniu rodzaju ataku program losuje wartość ataku przez wylosowanie liczby z przedziału liczb wylosowanych w statystykach przeciwnika, które są uprzednio pomnożone przez mnożnik rodzaju ataku. Następnie sprawdzane jest, czy atak trafił przeciwnika poprzez wylosowanie liczby z przedziału 1-100 i porównanie jej z szansą przeciwnika na trafienie. Jeśli liczba jest większa od szansy na trafienie, atak jest chybiony i wartość ataku zostaje wyzerowana. Po tej czynności następuje wyrenderowanie ataku. Działa to w taki sam sposób jak leczenie, jedyne różnice to zmienione animacje i kolory, jednak kod działa tak samo. Po zakończeniu tury przeciwnika wartość zmiennej opp.tura zostaje zmieniona na 0, co oznacza kolej gracza. Kod obsługujący turę gracza wygląda niemalże identycznie jak u przeciwnika, jedyną różnicą jest wybór akcji, która już nie jest losowana. Ponieważ walka jest zagnieżdżona w osobnej nieskończonej pętli, nie ma dostępu do głównej pętli gry, zatem w celu obsługi przycisków wymagane jest wywołanie kolejnej

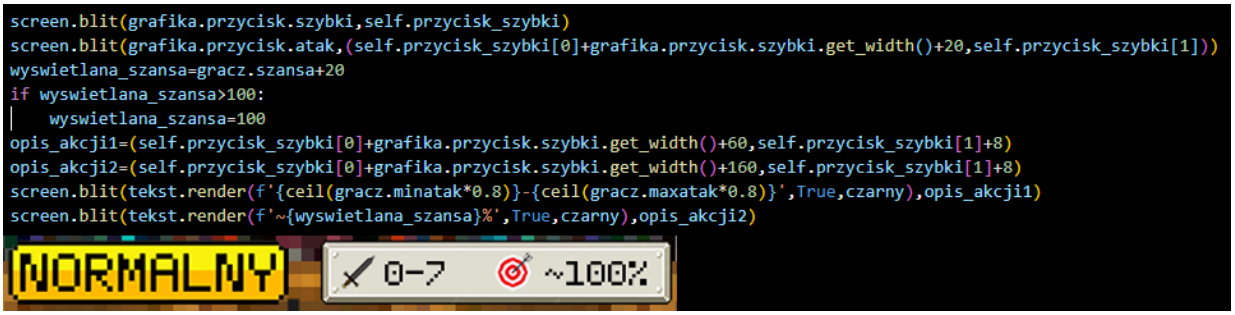
Rysunek 20 Walka w trakcie animacji leczenia

Rysunek 21 Postać gracza podczas animacji ataku

```
else:
    while True:
        if animacje[0]==len(grafika.gracz):
            animacje[0]=0
        if animacje[1]==len(grafika.gracz):
            animacje[1]=0
        if klatki==7:
            animacje[0]+=1
            animacje[1]+=1
            klatki=0
        klatki+=1
        akcja=0
        self.render_gracz(animacje)
        for event in pygame.event.get():
            if event.type==pygame.MOUSEBUTTONDOWN:
                akcja=self.buttons()
        if akcja!=0:
            break
```

Rysunek 22 Pętla odpowiedzialna za pobranie akcji od gracza

nieskończonej pętli. Rozpoczyna się ona od obsługi zmiennych odpowiedzialnych za animacje, tak jak miało to miejsce w mieście, po czym zostaje wykonana metoda „render_gracz”, która poza renderowaniem tła i pasków zdrowia, renderuje też animowane grafiki postaci oraz przyciski. Poza zwykłymi przyciskami, dla ataków generowana jest też grafika przedstawiająca graczowi obrażenia danego ataku oraz szansę na trafienie, dzięki czemu może on podjąć lepszą decyzję. Warto zwrócić uwagę na to, że dzięki zastosowaniu instrukcji warunkowej przycisk



Rysunek 23 Jeden z przycisków z metody "render_gracz" oraz przykładowy wygenerowany przycisk podczas walki

do użycia mikstury nie renderuje się jeśli gracz nie posiada żadnych mikstur. Po wybraniu dowolnej akcji program wychodzi z pętli i wykonuje odpowiednie komendy, identycznie jak podczas ruchu przeciwnika. Po każdym ruchu, niezależnie od aktualnej postaci, dwie instrukcje warunkowe sprawdzają, czy zdrowie którejś z postaci po danej turze jest niedodatnie. Jeśli tak, wyświetlany jest odpowiedni komunikat i program wychodzi z pętli walki. Przywołując odpowiednie funkcje, program może oznajmić o wygranej lub przegranej. Zacznę od tego drugiego. Gdy zdrowie gracza spadnie do 0, renderowana jest animacja podniesienia przez przeciwnika miecza w geście zwycięstwa, po czym pokazuje się okno tekstowe mówiące o przegranej. Użytkownik ma wtedy dwie opcje do wyboru: restart gry, co jest osiągnięte przez przypisanie graczowi początkowych statystyk i wyrenderowanie ekranu startowego, lub wyjście z programu. Komunikat o wygranej jest natomiast bardziej skomplikowany.

Rysunek 24 Komunikat o przegranej

```
if opp.trudnosc==3:
    self.koniec()
else:
    if opp.trudnosc==2:
        mnoznik=3
    else:
        mnoznik=1
    ZdobytyExp=mnoznik*randint(100,200)
    ZdobyteZloto=mnoznik*randint(150,250)
    gracz.exp+=ZdobytyExp
    gracz.zloto+=ZdobyteZloto
    if gracz.exp<gracz.lvlup:
        textbox(700,100,480,250,5)
        screen.blit(tekst.render('Wygrywasz!',True,czarny),(720,120))
        screen.blit(tekst.render(f'Otrzymujesz {ZdobytyExp} expa',True,czarny),(720,150))
        screen.blit(tekst.render(f'oraz {ZdobyteZloto} złota!',True,czarny),(720,180))
        screen.blit(tekst.render(f'Masz {gracz.exp} punktów doświadczenia.',True,czarny),(720,210))
        screen.blit(tekst.render(f'Brakuje ci {gracz.lvlup-gracz.exp} expa do poziomu {gracz.poziom+1}.',True,czarny),(720,240))
        screen.blit(grafika.przycisk.powrot,self.przycisk_powrot)
    else:
        if gracz.poziom==1:
            gracz.lvlup=625
        elif gracz.poziom==2:
            gracz.lvlup=2125
        gracz.poziom+=1
        gracz.maxhp+=10
        gracz.hp=gracz.maxhp
        gracz.exp=0
        gracz.minatak+=2
        gracz.maxatak+=2
        textbox(580,95,630,420,5)
        screen.blit(tekst.render('Wygrywasz!',True,czarny),(620,120))
        screen.blit(tekst.render(f'Otrzymujesz {ZdobytyExp} expa oraz {ZdobyteZloto} złota!',True,czarny),(620,150))
        screen.blit(tekst.render(f'Gratulacje!',True,czarny),(620,200))
        screen.blit(tekst.render(f'Osiągnąłeś {gracz.poziom} poziom doświadczenia!',True,czarny),(620,230))
        screen.blit(tekst.render(f'Twoje maksymalne zdrowie wzrasta o 10!',True,czarny),(620,260))
        screen.blit(tekst.render(f'Aktualnie masz go {gracz.maxhp}.',True,czarny),(620,290))
        screen.blit(tekst.render(f'Twoje ataki zadają 2 punkty obrażeń więcej!',True,czarny),(620,320))
        screen.blit(tekst.render(f'Aktualnie {gracz.minatak}-{gracz.maxatak}!',True,czarny),(620,350))
        screen.blit(tekst.render(f'Zyskujesz na celności!',True,czarny),(620,380))
        screen.blit(tekst.render(f'Masz {gracz.szansa}% szans na trafienie!',True,czarny),(620,410))
        screen.blit(grafika.przycisk.powrot,self.przycisk_w_powrot)
    global obszar
    obszar='wygrana'
```

Rysunek 25 Kod obsługujący komunikaty o wygranej

Odtwarzana jest identyczna animacja jak przy przegranej, tylko że tym razem to postać gracza podnosi miecz. Następnie wyświetlany jest komunikat z dodatkowymi informacjami. Jeśli przeciwnik był bossem, wyświetlana jest informacja o ukończeniu gry. Użytkownik dostaje wtedy opcję resetu gry lub wyjścia z programu, tak jak przy przegranej. W innym przypadku program losuje wartości zdobytego złota oraz punktów doświadczenia w zależności od trudności przeciwnika. Poźniej program sprawdza, czy gracz osiągnął kolejny poziom doświadczenia przez porównanie zmiennej z posiadanymi punktami doświadczenia do zmiennej z ilością punktów doświadczenia wymaganą do osiągnięcia kolejnego poziomu. Warto zauważyć, że nie jest wymagane sprawdzenie, czy gracz ma maksymalny poziom, gdyż postać na 4 poziomie doświadczenia może zawalczyć jedynie z „bossem”. Jeśli gracz osiągnął wymaganą ilość, jego statystyki zwiększają się, po czym zostaje powiadomiony o ich aktualnych wartościach. W przeciwnym wypadku dostaje informację o tym, ile punktów brakuje mu do kolejnego poziomu. W obu przypadkach zostaje wyrenderowany przycisk powrotu do miasta, po kliknięciu którego gracz wraca do głównej pętli gry.

Rysunek 26 Przykładowy komunikat o zwycięstwie po osiągnięciu kolejnego poziomu doświadczenia

W ten sposób przedstawiłem dogłębnie całe działanie mojej autorskiej gry. Nie jest to skomplikowany program, mimo to tworzenie go było bardzo satysfakcjonujące. Wiele razy zagrałem w tę grę i dobrze się bawiłem. Chcę tym referatem zachęcić wszystkich zainteresowanych Pythonem do eksperymentowania i poświęcenia części wolnego czasu na naukę. Programowanie nie jest trudne, tylko czasochłonne, jednak kiedy spędzi się przy nim kilka godzin, każdy może stworzyć coś wyjątkowego.