Projekt na Inżynierię Oprogramowania UJ 2021

Temat: Gra platformowa
Technologia: Python z użyciem PyGame

Wzorce: MVVM i Singleton

Jako projekt zdecydowaliśmy się na stworzenie gry, w której gracz musi zebrać **6 monet**, a następnie jak najdłużej utrzymać się przy życiu aby wygrać. Gra posiada dwa poziomy oraz dużo pułapek.

Na samym początku zdecydowaliśmy się na konkretne wzorce projektowe, tak aby łatwiej nam się kolaborowało oraz aby spełnić dane wymagania na temat ich obecności.

Używamy **MVVM** dla gracza: w warstwie **MODEL** importujemy potrzebne do działania assety gracza oraz ustawiamy jego początkową pozycję.

```
self.vectX = 70
self.vectY = 300
self.velocityY = -5
self.playerSprite = pygame.image.load('assets/heroes/3 Dude_Monster/idle/1.png').convert_alpha()
self.playerSprite = pygame.transform.scale(self.playerSprite, (45, 69)).convert_alpha()
self.playerRect = self.playerSprite.get_rect()
self.playerRect.topleft = (70, 300)

self.playerRun1 = pygame.image.load('assets/heroes/3 Dude_Monster/run/1.png').convert_alpha()
self.playerRun1 = pygame.transform.scale(self.playerRun1, (52, 69)).convert_alpha()
self.playerRun2 = pygame.image.load('assets/heroes/3 Dude_Monster/run/2.png').convert_alpha()
self.playerRun2 = pygame.transform.scale(self.playerRun2, (52, 69)).convert_alpha()
self.playerRun3 = pygame.image.load('assets/heroes/3 Dude_Monster/run/3.png').convert_alpha()
self.playerRun3 = pygame.transform.scale(self.playerRun3, (52, 69)).convert_alpha()
self.playerRun4 = pygame.image.load('assets/heroes/3 Dude_Monster/run/4.png').convert_alpha()
self.playerRun4 = pygame.transform.scale(self.playerRun4, (52, 69)).convert_alpha()
self.playerRun5 = pygame.image.load('assets/heroes/3 Dude_Monster/run/5.png').convert_alpha()
self.playerRun5 = pygame.transform.scale(self.playerRun5, (52, 69)).convert_alpha()
self.playerRun6 = pygame.image.load('assets/heroes/3 Dude_Monster/run/6.png').convert_alpha()
self.playerRun6 = pygame.transform.scale(self.playerRun6, (52, 69)).convert_alpha()
```

W warstwie **VIEW-MODEL** obliczamy poruszanie się gracza, trzymamy warunki na skok, spadanie i bieganie.

```
def playerMovement(self, didJump):

    keys = pygame.key.get_pressed()

    if keys[pygame.K_a] and self.player.vectX > 0:
        self.player.vectX -= 5
        self.player.playerRect.x -= 5
        self.player.spriteItr += 1

        if not game.turnedLeft:
            game.turnedLeft = True
```

Ostatnia warstwa, czyli **VIEW** jest odpowiedzialna za wyświetlanie gracza na ekranie komputera.

```
class VIEW_PLAYER:
    def __init__(self, p):
        self.viewModelPlayer = p

    def drawPlayer(self, sc):
        self.screen = sc
        self.screen.blit(self.viewModelPlayer.getPlayerSpriteRun(),
                         (self.viewModelPlayer.getPlayerX(), self.viewModelPlayer.getPlayerY()))
```

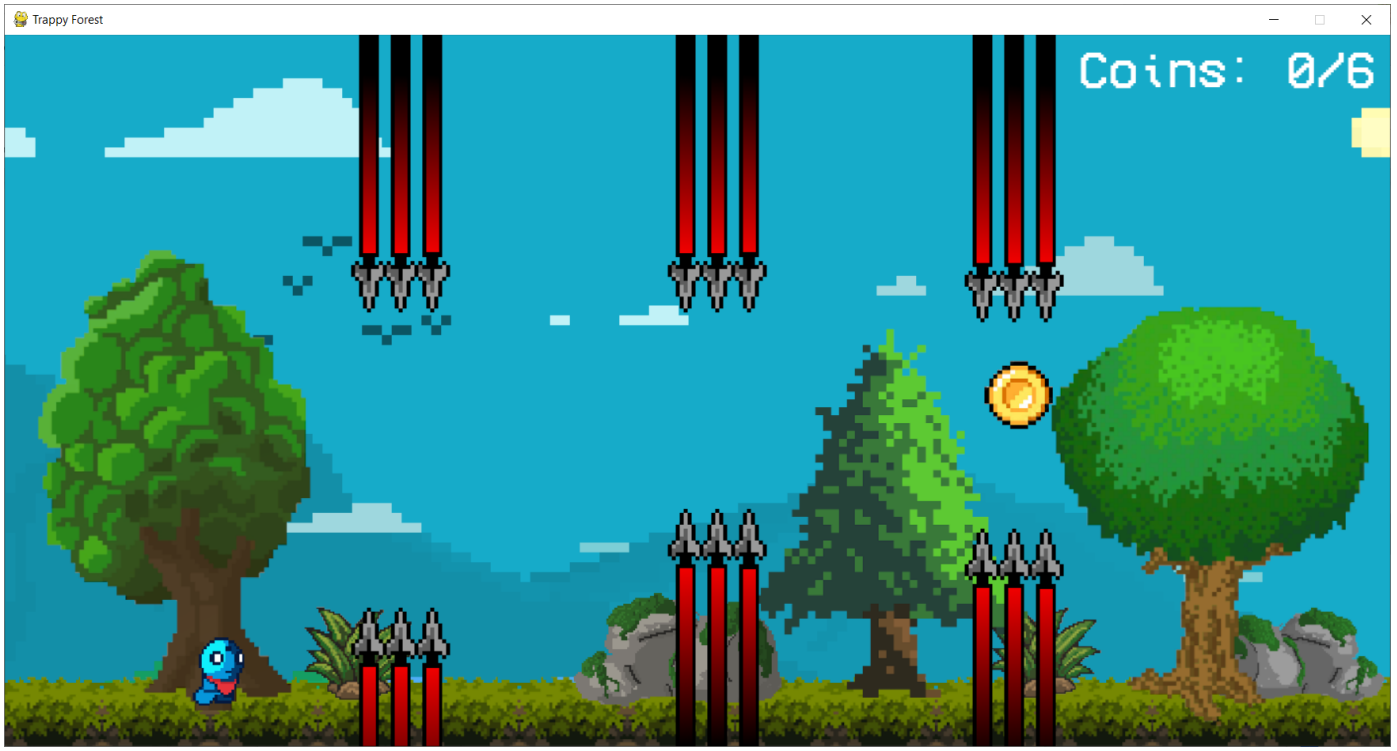
Drugim wzorcem, którego użyliśmy jest **Singleton**. Jest to dosyć łatwy koncepcyjnie wzorzec projektowy oraz przyjemnie się na nim pracuje podczas robienia gry. Stworzyliśmy jedną klasę o nazwie **GAME**, której zapewniliśmy możliwość obecności tylko jednej instancji i w niej zawarliśmy resztę gry.

W Singletonie stworzyliśmy miejsce na **INIT**, czyli inicjalizację potrzebnych zmiennych, assetów i innych składowych oraz stworzyliśmy **MAIN GAME LOOP**, czyli pętlę główną gry, w której wywołujemy wszystkie rzeczy i rysujemy wszystko na ekranie.

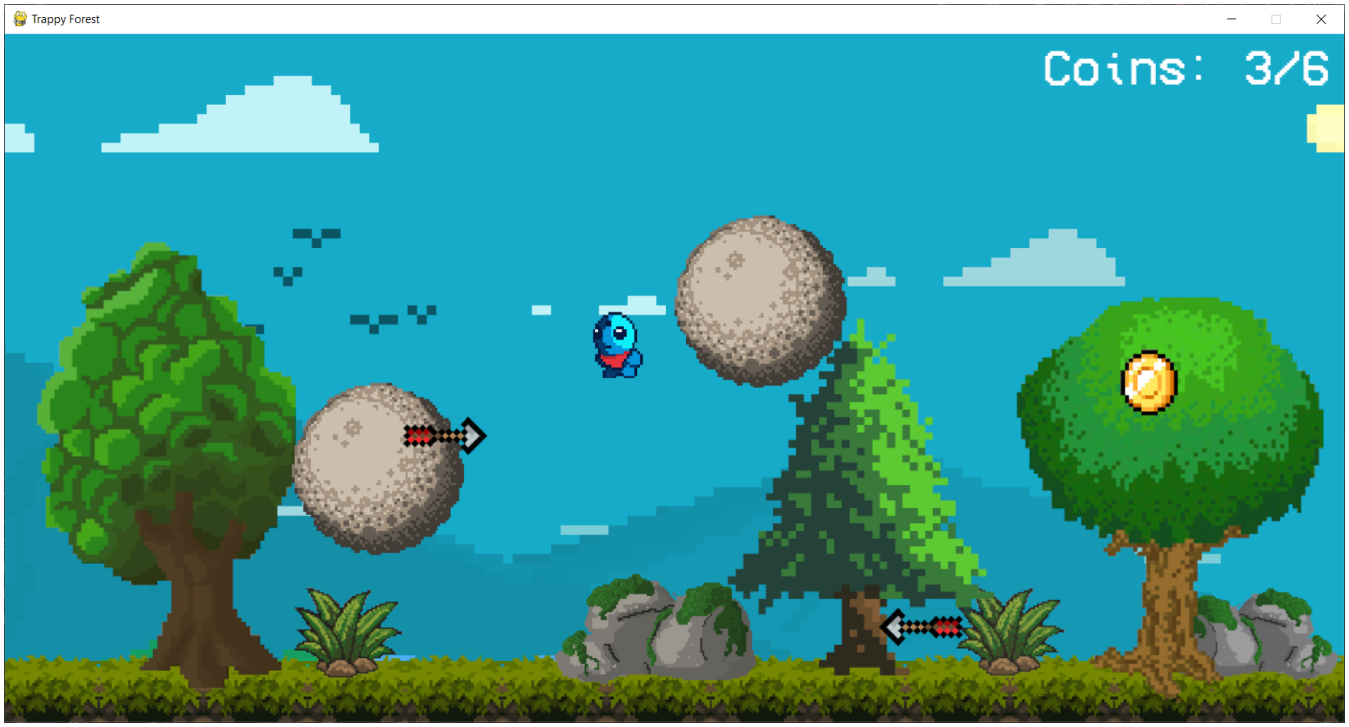
```
# GAME INIT:
pygame.init()
pygame.display.set_caption('Trappy Forest')
width = 1400
height = 720
screen = pygame.display.set_mode((width, height))
clock = pygame.time.Clock()
mixer.init()
soundtrack = mixer.Sound('assets/music/soundtrack/soundtrack_op.wav')
soundtrack.play(-1)
```

```
############################################################
# GAME MAIN LOOP:
############################################################

def playGame(self):
    while True:
        self.clock.tick(120)
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN and not self.gameActive and not self.didWin:
                    self.gameActive = True
                    self.didLose = False
                    self.turnedLeft = False
                    self.initPlayerAndAscreen()
                    self.setBouldersLevel2Rects()
                    self.setArrowsLevel2Rects()

                if event.key == pygame.K_ESCAPE:
                    pygame.quit()
                    sys.exit()
                if event.key == pygame.K_SPACE and self.gameActive and self.didJump == False:
                    if self.player.vectY > 0:
                        self.jumpSound.play()
                        self.didJump = True
```

Singleton zawiera przede wszystkim dużo różnych funkcji, które odpowiedzialne są za poprawne działanie gry w sensie logiki i wizualizacji graficznej.
Są tam funkcje rysujące ekran powitalny, końcowy i po śmierci gracza. Są również funkcje ustawiające obiekty w określonych miejscach (np. kolce, monety do zbierania), generujące poruszanie się przeszkód, które zabijają gracza (spadające skały, strzały) czy obliczające wynik gracza na koniec gry (ilość zebranych monet i przetrwany czas na koniec gry).



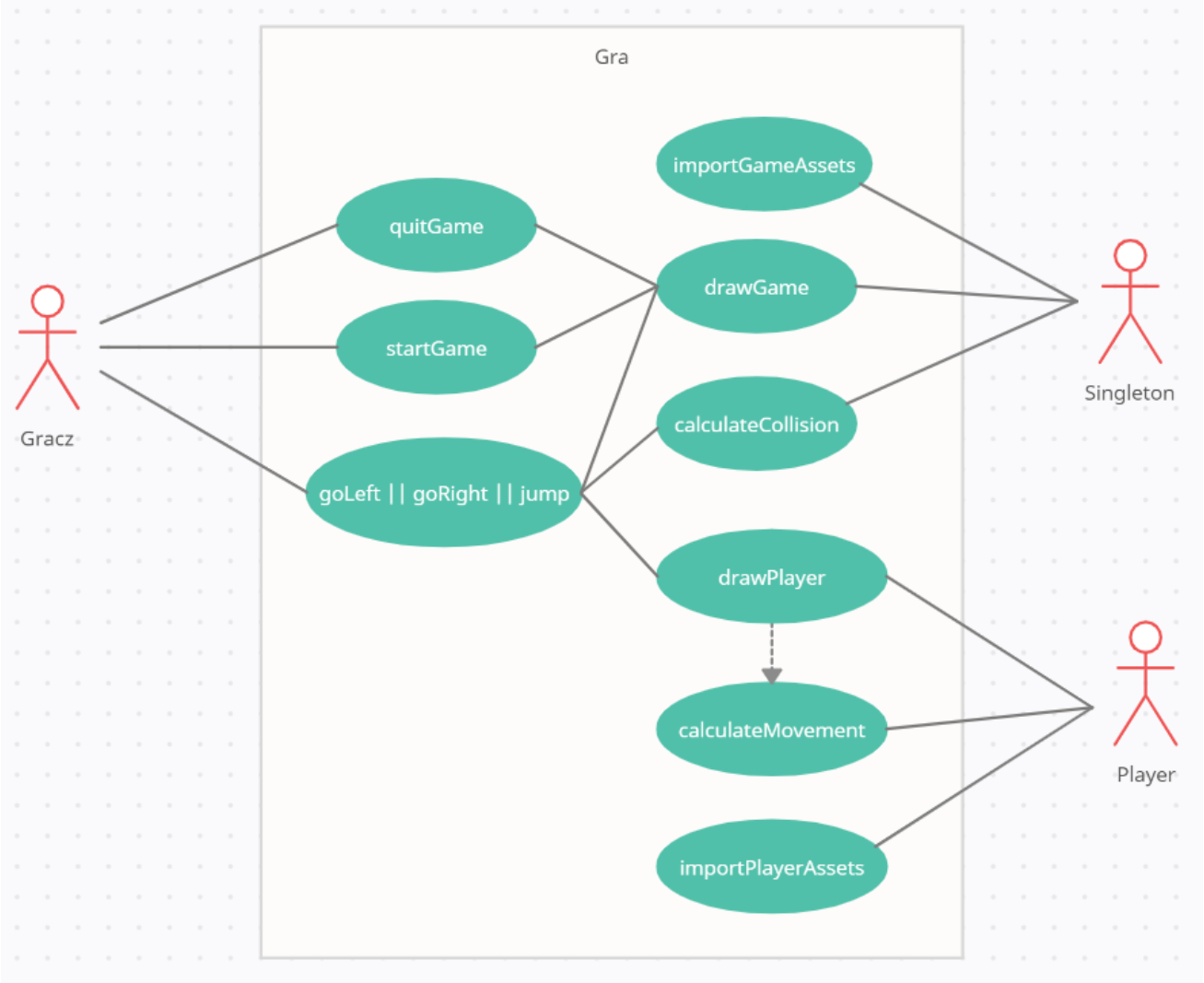
Na ekranie komputera z częstotliwością 120 klatek na sekundę rysowane jest widoczne powyżej tło, kolce, gracz, monety oraz wynik. Wszystko odświeżane jest w czasie rzeczywistym, więc każda zmiana jest rejestrowana na ekranie (np. jeśli gracz zbierze monetę, to moneta znika). Celem gracza jest zdobycie łącznie **6 monet** (3 na pierwszym poziomie oraz 3 na drugim) i jak najdłuższe przetrwanie. Jeśli gracz dotknie kolców, da się trafić strzałą lub spadnie na niego głaz - przegrywa. Po śmierci, gracz może ponownie zacząć grę od poziomu na którym zginął. Liczba monet resetuje się po śmierci.



Kolizje w PyGame są realizowane na podstawie nałożenia się na siebie dwóch prostokątów, które są “narysowane” dookoła danego obiektu. Poniżej przykład kolizji gracza z kolcami.

Użyliśmy środowiska PyCharm do wygenerowania **UML** [LINK].

Poniżej widnieje diagram przypadków użycia.
Aktor **Gracz** może jedynie włączyć grę, wyłączyć grę lub poruszać się postacią. Wszystko co związane z graczem, czyli wszelkie obliczenia i importy wykonywane są przez aktora **Player**.
Singleton natomiast zajmuje się rysowaniem całej gry, importami związanymi z wszystkim poza **Playerem** oraz obliczaniem kolizji z przeszkodami.



Projekt wykonali: Jan Skwarczek, Igor Łonak, Lena Kaczanowska