Gliwice, 26.05.2020

LABORATORIUM PROGRAMOWANIA KOMPUTERÓW 4

Informatyka AEiI, gr.6
sekcja 11

Platformówka 2D

Autor:

Jakub Cieślik

Temat projektu

Celem projektu było stworzenie gry platformowej, w której celem jest przechodzenie plansz i zdobywanie przy tym jak największej ilości punktów. Gracz porusza się w dwóch wymiarach – ma możliwość poruszania się w poziomie i skakania (z grawitacją). Poziomy gry zawierają przeszkody i przeciwników, utrudniających dotarcie do końca, ale też przedmioty do zebrania, uławiające dotarcie do celu.

Analiza

Do stworzenia warstwy wizualnej gry wykorzystana została biblioteka SFML. Dzięki niej możliwe jest proste korzystanie z okna aplikacji i wyświetlanie obiektów. Każdy obiekt w grze posiada swoją własną teksturę. Ze względu na swoją funkcjonalność, zdecydowałem podzielić ich klasy na trzy kategorie:
- Stałe bloki
- Poruszające się jednostki (przeciwnicy i gracz)
- Przedmioty do zebrania

Pierwsza z nich obejmuje obiekty, które mają ustaloną funkcję i pozycję w grze, takie jak stałe bloki z których zbudowane są poziomy, kolce (przeszkoda dla gracza) oraz meta. W drugiej kategorii znajduje się gracz oraz przeciwnicy, którzy posiadają różne poziomy sztucznej inteligencji oraz parametry takie jak prędkość czy wielkość, a także możliwość poruszania się po planszy. W trzeciej kategorii znajdują się przedmioty, które ułatwiają graczowi rozgrywkę, na przykład pozwalając mu na pokonywanie przeciwników czy szybsze poruszanie się.

W projekcie zaimplementowana została również klasa pocisku, z której korzysta gracz oraz jeden z typów przeciwników. Gracz ma możliwość strzelania, w celu pokonania przeciwników, a przeciwnicy starają się strzelać w postać użytkownika. Przy kolizji gracza z pociskiem przeciwnika, traci on punkt życia i zostaje przeniesiony na początek poziomu.

Reszta klas odpowiedzialna jest za właściwa logikę gry, co zostanie opisane w specyfikacji wewnętrznej.

Algorytmy

Najważniejszym algorytmem w projekcie jest sprawdzanie kolizji między obiektami. Ważna była możliwość sprawdzenia kierunku, z której nastąpiła kolizja, dlatego zastosowałem następujący algorytm:

```
direction Tile::collision_check(Unit* object)
{
    float dx = (this->get_position().x + this->size.x / 2) - (object->get_position().x + object->get_size().x / 2); //odleglosc w osi x miedzy obiektami
    float dy = (this->get_position().y + this->size.y / 2) - (object->get_position().y + object->get_size().y / 2); //odleglosc w osi y miedzy obiektami
    float width = (this->size.x + object->get_size().x) / 2;   // suma polowek szerokosci obiektow (odleglosc od srodkow w osi x)
    float height = (this->size.y + object->get_size().y) / 2;  // suma polowek wysokosci obiekow (odleglosc od srodkow w osi y)
    float crossWidth = width * dy;                             // szerokosc przeciecia miedzy obiektami
    float crossHeight = height * dx;                           // wysokosc przeciecia miedzy obiektami
    direction collision = direction::no_direction;
    if (abs(dx) <= width && abs(dy) <= height)      // test, czy obiekty sie przecinaja
    {
        if (crossWidth > crossHeight) // jezeli szerokosc przeciecia jest wieksza od wysokosci (wysokosc ujemna), to kolizja zaszla z dolu lub z lewej
            collision = (crossWidth > (-crossHeight)) ? direction::down : direction::left;
        else   // w przeciwnym razie, z prawej lub z gory
            collision = (crossWidth > (-crossHeight)) ? direction::right : direction::up;
    }
    return collision;
}
```

Dzięki zwracaniu kierunku kolizji można wykonać odpowiednią metodę odpowiedzialną za obsłużenie kolizji, na przykład w przypadku zderzenia z kolcami tylko kolizja od góry powoduje utratę życia gracza.

Podczas tworzenia gry ważną kwestią było dla mnie również utworzenie sztucznej inteligencji, która stawiała by jakieś wyzwanie dla gracza. W taki sposób rozwiązywałem więc poruszanie się jednego z przeciwników:

```
int x = int(object->get_position().x / 32.f);
int y = int(object->get_position().y / 32.f);
if (dynamic_cast<Skeleton*>(object) && object->get_velocity().y == 0.f) // jezeli to szkielet i nie spada w dol (zacinalby sie w scianie)
{
    if (object->get_direction() == direction::right) // jezeli idzie w prawo
    {
        if (objects.tilemap[make_pair(y, x + 1)] == nullptr && objects.tilemap[make_pair(y + 1, x + 1)] == nullptr
            && objects.tilemap[make_pair(y + 1, x)] != nullptr && objects.tilemap[make_pair(y + 2, x + 1)] == nullptr
            && objects.tilemap[make_pair(y + 2, x + 2)] == nullptr)
            object->change_direction_left(); // jezeli jest pod nim blok, a odpowiednich blokow po prawej nie ma, zawroc (jest przepasc)
        else if (objects.tilemap[make_pair(y - 1, x + 1)] == nullptr && objects.tilemap[make_pair(y - 1, x + 2)] == nullptr
            && objects.tilemap[make_pair(y, x + 2)] != nullptr && object->check_can_jump())
            object->jump(object->get_jump_height()); // jezeli jest przed nim przeszkoda i nie ma nad nim sufitu, przeskocz przeszkode
    }
    else // gdy idzie w lewo, sprawdza odpowiednie bloki po lewej
    {
        if (objects.tilemap[make_pair(y, x)] == nullptr && objects.tilemap[make_pair(y + 1, x)] == nullptr
            && objects.tilemap[make_pair(y + 1, x + 1)] != nullptr && objects.tilemap[make_pair(y + 2, x)] == nullptr
            && objects.tilemap[make_pair(y + 2, x - 1)] == nullptr)
                object->change_direction_right();
        else if (objects.tilemap[make_pair(y - 1, x)] == nullptr && objects.tilemap[make_pair(y - 1, x - 1)] == nullptr
            && objects.tilemap[make_pair(y, x - 1)] != nullptr && object->check_can_jump())
            object->jump(object->get_jump_height());
    }
}
```

Szkielet sprawdza czy może zeskoczyć z bloku bez spadnięcia w przepaść, a także czy jest przed nim jakaś przeszkoda, którą może przeskoczyć. Podobnie zachowuje się szczur, który wykonuje jednak tylko uproszczoną wersję tego algorytmu (zawraca w przypadku nawet jedno-blokowej przepaści).

Dzięki podzieleniu planszy na bloki o rozmiarze 32x32 piksele, łatwo jest określić pozycje danego obiektu względem stałych bloków, dzięki czemu nie trzeba sprawdzać kolizji z wszystkimi obiektami, a tylko z tymi wokół obiektu, dla którego sprawdzana jest kolizja.

```
direction collision_side = direction::no_direction;
int x = int(object->get_position().x / 32.f);
int y = int(object->get_position().y / 32.f);
for (int i = y - 1; i <= y + 1; i++)
{
    for (int j = x - 1; j <= x + 1; j++)
    {
        if (objects.tilemap[make_pair(i, j)] != nullptr) // jezeli w danym miejscu jest blok
        {
            collision_side = objects.tilemap[make_pair(i, j)]->collision_check(object); //sprawdz, z ktorej strony jest kolizja
            if (collision_side == direction::left)  // jezeli z lewej, obsluz kolizje z lewej
                objects.tilemap[make_pair(i, j)]->collision_left(object);
            else if (collision_side == direction::right)  // jezeli z prawej, obsluz kolizje z prawej
                objects.tilemap[make_pair(i, j)]->collision_right(object);
        }
    }
}
```

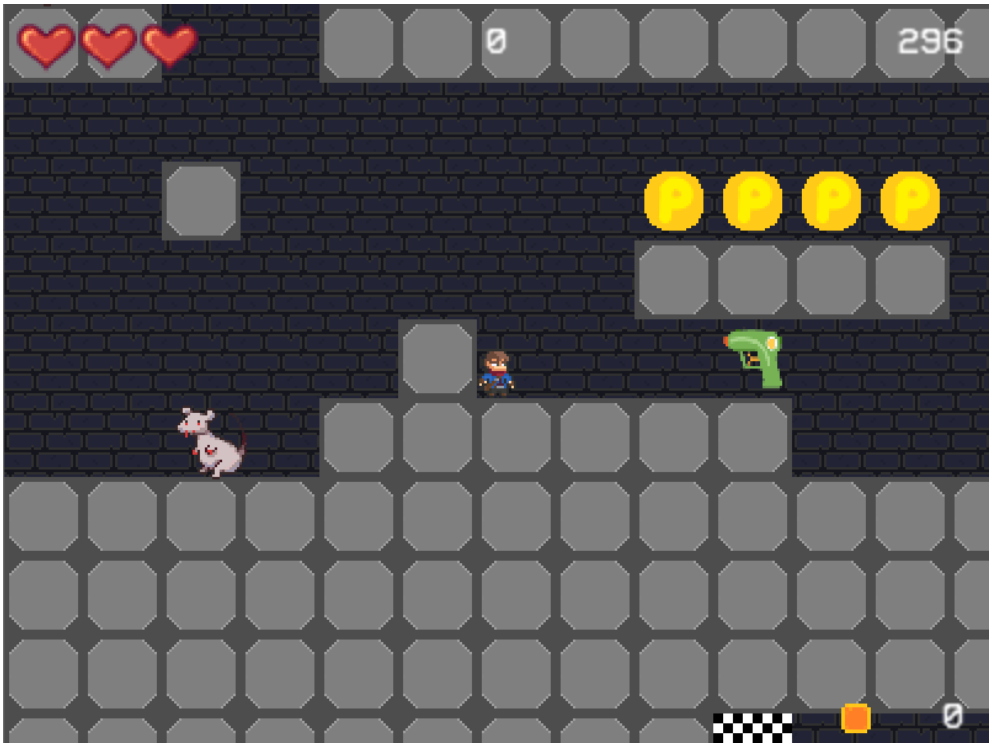
Ze względu na sposób w jaki obsługiwana jest kolizja, konieczne jest osobne sprawdzanie kolizji w poziomie, a następnie w pionie. Dzięki temu nie występują żadne błędy w kolizji, takie jak zacinanie się na ścianach.

Specyfikacja zewnętrzna

Po uruchomieniu programu użytkownikowi wyświetlane jest menu główne gry:

Po menu można poruszać się za pomocą strzałek, a wybraną opcje zatwierdzać za pomocą przycisku enter. Po wybraniu opcji najlepsze wyniki wyświetlana jest tabela:

W przypadku wyboru opcji „Rozpocznij grę”, zaczyna się faktyczna rozgrywka, której wygląd przedstawiony jest na poniższym zrzucie ekranu:

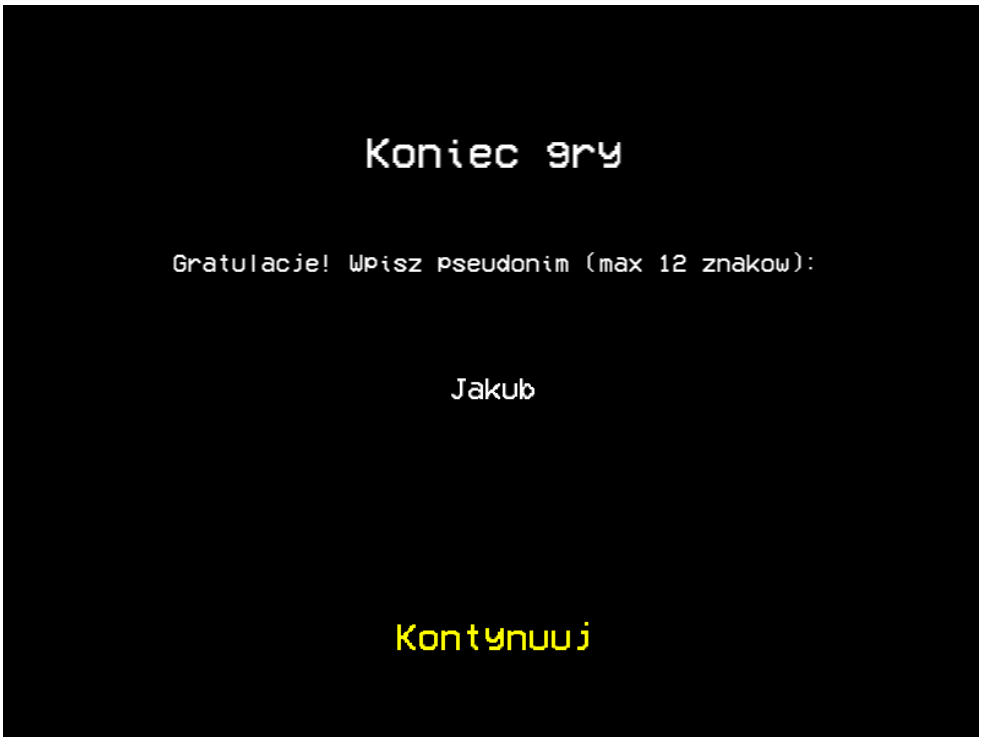


W centrum ekranu widoczna jest postać gracza. Poruszać się można za pomocą strzałek (lewo i prawo – poruszanie się w poziomie, góra – skok). Przycisk Z powoduje wystrzelenie pocisku (w prawym dolnym rogu interfejsu widoczna jest obecna ilość pocisków, które można wykorzystać).
Elementy interfejsu od lewej:
- ilość pozostałych żyć
- ilość zdobytych punktów
- czas pozostały na przejście poziomu

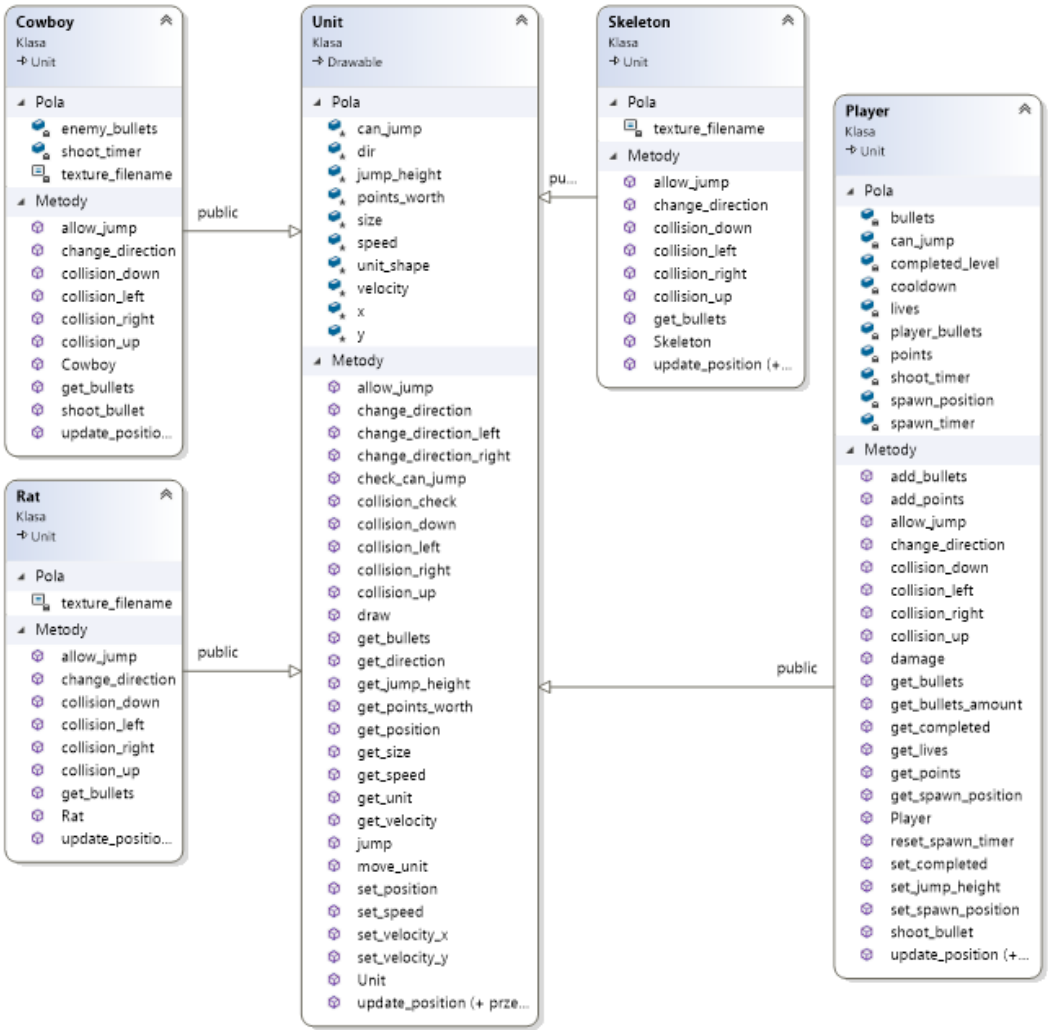
Możliwe jest również zapauzowanie gry klawiszem Escape, co powoduje wyświetlenie następującego ekranu:

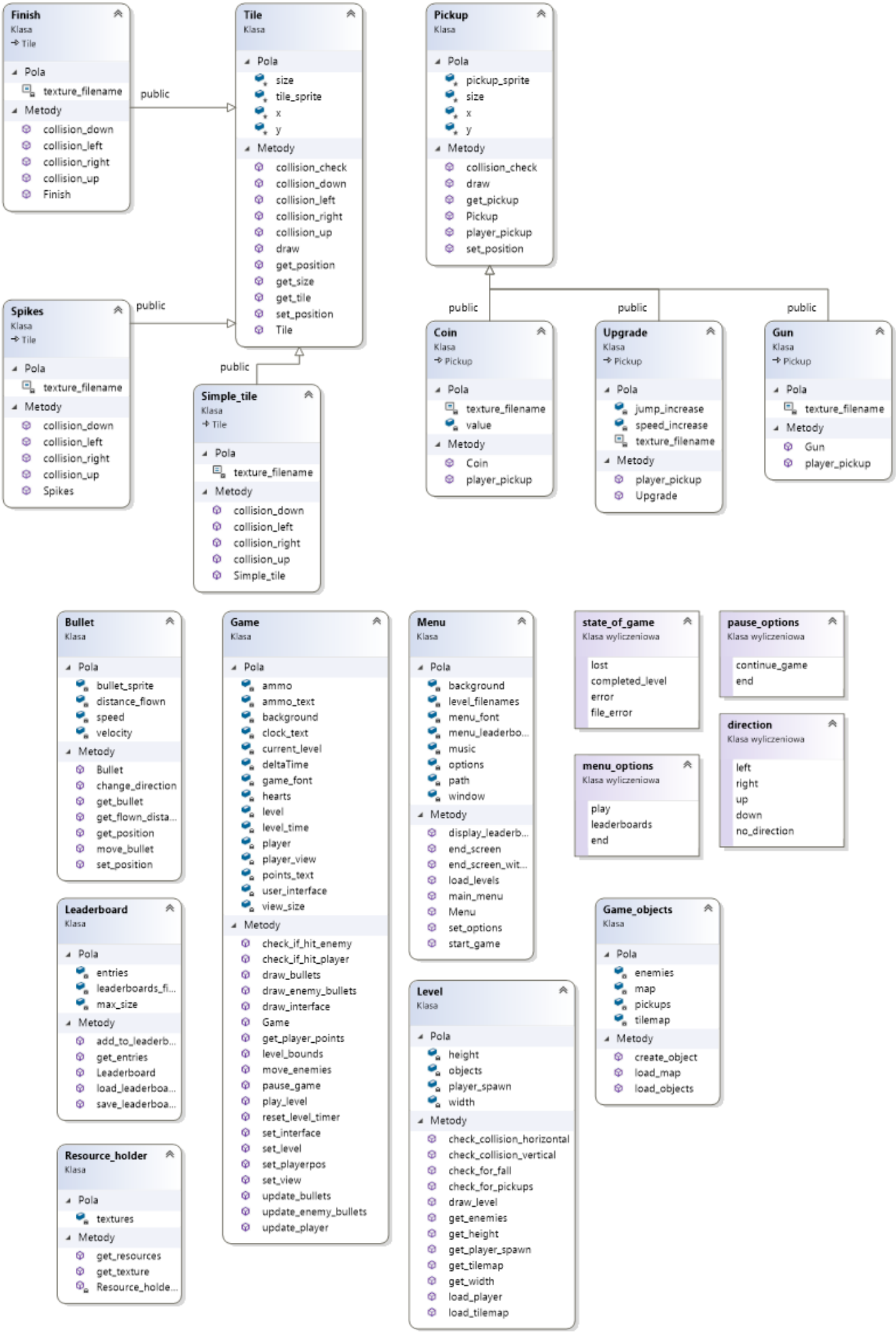
Wybranie odpowiedniej opcji pozwala na kontynuowanie gry, lub powrót do menu głównego.

Zakończenie gry powoduje wyświetlenie następującego ekranu, w którym trzeba wpisać pseudonim, który przy odpowiednio dużej ilości punktów powoduje wpis do tablicy najlepszych wyników.



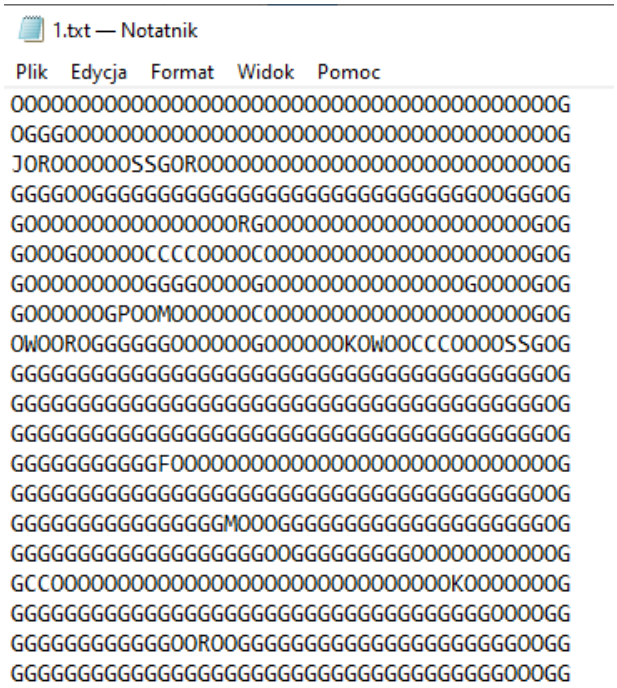
Specyfikacja wewnętrzna




Finish
Klasa
Tile
Pola
texture_filename
Metody
collision_down
collision_left
collision_right
collision_up
Finish
public
Tile
Klasa
Pola
size
tile_sprite
x
y
Metody
collision_check
collision_down
collision_left
collision_right
collision_up
draw
get_position
get_size
get_tile
set_position
Tile
Pickup
Klasa
Pola
pickup_sprite
size
x
y
Metody
collision_check
draw
get_pickup
Pickup
player_pickup
set_position
Spikes
Klasa
Tile
Pola
texture_filename
Metody
collision_down
collision_left
collision_right
collision_up
Spikes
public
public
Simple_tile
Klasa
Tile
Pola
texture_filename
Metody
collision_down
collision_left
collision_right
collision_up
Simple_tile
public
Coin
Klasa
Pickup
Pola
texture_filename
value
Metody
Coin
player_pickup
public
Upgrade
Klasa
Pickup
Pola
jump_increase
speed_increase
texture_filename
Metody
player_pickup
Upgrade
public
Gun
Klasa
Pickup
Pola
texture_filename
Metody
Gun
player_pickup
Bullet
Klasa
Pola
bullet_sprite
distance_flown
speed
velocity
Metody
Bullet
change_direction
get_bullet
get_flown_dista...
get_position
move_bullet
set_position
Game
Klasa
Pola
ammo
ammo_text
background
clock_text
current_level
deltaTime
game_font
hearts
level
level_time
player
player_view
points_text
user_interface
view_size
Metody
check_if_hit_enemy
check_if_hit_player
draw_bullets
draw_enemy_bullets
draw_interface
Game
get_player_points
level_bounds
move_enemies
pause_game
play_level
reset_level_timer
set_interface
set_level
set_playerpos
set_view
update_bullets
update_enemy_bullets
update_player
Menu
Klasa
Pola
background
level_filenames
menu_font
menu_leaderbo...
music
options
path
window
Metody
display_leaderb...
end_screen
end_screen_wit...
load_levels
main_menu
Menu
set_options
start_game
state_of_game
Klasa wyliczeniowa
lost
completed_level
error
file_error
pause_options
Klasa wyliczeniowa
continue_game
end
direction
Klasa wyliczeniowa
left
right
up
down
no_direction
menu_options
Klasa wyliczeniowa
play
leaderboards
end
Leaderboard
Klasa
Pola
entries
leaderboards_fi...
max_size
Metody
add_to_leaderb...
get_entries
Leaderboard
load_leaderboa...
save_leaderboa...
Level
Klasa
Pola
height
objects
player_spawn
width
Metody
check_collision_horizontal
check_collision_vertical
check_for_fall
check_for_pickups
draw_level
get_enemies
get_height
get_player_spawn
get_tilemap
get_width
load_player
load_tilemap
Game_objects
Klasa
Pola
enemies
map
pickups
tilemap
Metody
create_object
load_map
load_objects
Resource_holder
Klasa
Pola
textures
Metody
get_resources
get_texture
Resource_holde...

Klasa Menu jest główna klasą gry. Odpowiada ona za inicjalizowanie rozgrywki oraz obsługę zdarzeń w grze. Jej metody (end_screen, start_game, czy main_menu) odpowiadają za wyświetlanie poszczególnych stadiów rozgrywki. Przechowuje ona również obiekt klasy Leaderboard – odpowiedzialnej za obsługę tabeli najlepszych wyników, z możliwością zapisu i odczytu z pliku, a także dodania nowego wyniku. Tabela przechowywana jest w wektorze łańcuchów znakowych.

Metoda start_game tworzy obiekt klasy Game i sprawdza obecny stan gry, przechodząc jednocześnie przez wszystkie poziomy gry. Plansze tworzone są w plikach tekstowych w folderze levels/. Poniżej przykładowy poziom:



Każda litera odpowiada tutaj obiektowi w grze (przy czym O oznacza brak obiektu), plansza podzielona jest wiec na bloki 32x32 piksele. Każdy obiekt ma przypisany do siebie znak, po którym jest rozpoznawany przy odczytywaniu planszy z pliku. Program pozwala na tworzenie nowych poziomów poprzez dodawanie plików tekstowych z poziomami do folderu levels.

Klasy Tile, Pickup oraz Unit są klasami abstrakcyjnymi. Każda z nich posiada metody wirtualne odpowiedzialne za obsługę kolizji z danym obiektem, pozwalając na różne skutki kolizji.

Klasy pochodne dla klasy Tile to Simple_tile (zwykły blok, z którego zbudowane są poziomy), Spikes (kolce, powodują utratę życia gracza po wskoczeniu na nie) oraz Finish (meta, pozwalająca przejść do następnego poziomu).

Dla klasy Pickup są to Coin (po jego zebraniu, gracz otrzymuje ustaloną liczbę punktów), Upgrade (powoduje zwiększenie szybkości i wysokości skoków gracza) oraz Gun (daje graczowi losową liczbę amunicji). Klasa Pickup posiada uproszczoną metodę check_collision, sprawdzającą jedynie czy kolizja nastąpiła, gdyż nie jest w niej potrzebne sprawdzanie kierunku kolizji.

Klasa Unit posiada cztery klasy pochodne – Rat (prosty przeciwnik, poruszający się w poziomie i zawracający przy napotkaniu przeszkody), Skeleton (potrafi przeskakiwać przeszkody i może zeskakiwać na platformy poniżej), Cowboy (przeciwnik skaczący i potrafiący strzelać w gracza), a także Player (gracz, kontrolowany przez użytkownika, może skakać i strzelać). Metoda update_position pozwala na poruszanie się tych obiektów w czasie.

Obiekty tych klas przechowywane są w klasie Game_objects. Bloki i przedmioty do zebrania przechowywane są w mapie, której kluczem jest ich pozycja na planszy, natomiast przeciwnicy przechowywani są w wektorze, ze względu na to, że stale zmieniają pozycję.

```
std::map<std::pair<int, int>, std::unique_ptr<Tile>> tilemap;
std::map<std::pair<int, int>, std::unique_ptr<Pickup>> pickups;
std::vector<std::unique_ptr<Unit>> enemies;
```

Dzięki zastosowaniu klas abstrakcyjnych, obiekty klas potomnych mogą być przechowywane w jednym kontenerze. Używane są również tutaj inteligentne wskaźniki, aby ułatwić kontrolę pamięci. Za pomocą metod tej klasy możliwe jest wczytywanie poziomów do powyższych kontenerów.

Klasa Resource_holder zaimplementowana została jako singleton. Przechowuje ona wszystkie tekstury wykorzystywane w grze i pozwala na wczytywanie ich z pliku i przekazywanie ich do obiektów przy ich tworzeniu. W przypadku gdy nie uda się wczytać danej tekstury, wyrzucany jest wyjątek.

Klasa Level posiada obiekt klasy Game_objects. Jest to klasa odpowiedzialna za wczytywanie poziomu z pliku, wyświetlanie go na ekranie, a także sprawdzanie kolizji między obiektami.

Za całą logikę gry odpowiedzialna jest klasa Game. Posiada ona obiekt klasy Level, który jest obecnie rozgrywanym poziomem, obiekt Player, a także wszelkie elementy interfejsu. Jej metody pozwalają na poruszanie się obiektów w grze, rozgrywanie poziomów, wyświetlanie okna gry oraz interfejsu, a także wyświetlanie pocisków i sprawdzanie ich kolizji.

Obiekty klasy Bullet przechowywane są w wektorach w klasach Player oraz Cowboy. Gdy obiekty tych klas strzelają, tworzony jest nowy pocisk i dodawany do wektora. Po kolizji pocisku, lub przeleceniu przez pewną odległość, pocisk jest usuwany z pamięci i wektora.

W projekcie zastosowanych zostało też kilka klas wyliczeniowych, pomagających w czytelności kodu, z których najważniejsza jest klasa direction, pozwalająca na określenie kierunku kolizji, a także kierunku w którym porusza się dany obiekt (pochodne klasy Unit oraz Bullet).

```
std::vector<std::unique_ptr<Bullet>> player_bullets;
```

W projekcie zostały wykorzystane następujące zagadnienia z laboratorium:

- RTTI (używany static_cast i dynamic_cast)

- Mechanizm wyjątków (gdy program wykryje brak plików tekstur, program się kończy i wyrzuca wyjątek, tak samo w przypadku błędów przy wczytywaniu poziomów)

- Inteligentne wskaźniki (przy tworzeniu wszystkich dynamicznie alokowanych obiektów używany jest unique_ptr)

- Algorytmy i iteratory (Używane są iteratory do przechodzenia przez wektory, używany algorytm sort, funkcja lambda i inne)

- Kontenery STL (mapy, wektory)

Testowanie i uruchamianie

Program został wielokrotnie przetestowany i nie posiada on większych błędów. Występują pewne niedociągnięcia, jak na przykład fakt, że przeciwnicy posiadają wektor wystrzelonych przez nich pocisków, przez co fakt pokonania ich i usunięcia z gry powoduje usunięcie również lecących jeszcze pocisków. Nie uważam tego jednak za błąd, a pewną specyfikę działania.

Problemem jest również złożoność operacji wyświetlania wszystkich obiektów w każdej klatce. W obecnym momencie program wyświetla wszystkie obiekty, co przy dużej ich ilości może powodować problemy z wydajnością. Zauważalna jest jednak różnica między trybami kompilacji – w trybie debug przy bardzo dużej ilości obiektów gra wyraźnie spowalnia, a w trybie release działa dalej płynnie.

Wnioski

Stworzenie gry, w której akcje wykonują się z każdą klatką okazało się zadaniem nietrywialnym, z powodu często dużej nieprzewidywalności wykonywanych operacji. Klatki animacji nie są wykonywane w stałym czasie, w związku z tym w pewnych sytuacjach trudno było odnaleźć błędy. Uważam jednak, że w projekcie udało mi się spełnić mój cel, jakim było stworzenie gry, w której można dość łatwo dodawać nowe elementy oraz tworzyć nowe poziomy.