Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki

Programowania Komputerów 4

Space Invaders by JR

autor	Jeremiasz Radłowski
Prowadzący	dr.inż. Anna Gorawska
Rok akademicki	2020/2021
Kierunek	Informatyka
Rodzaj studiów	SSI
Semestr	4
Termin laboratoriów	-----
Sekcja	11
Termin oddania sprawozdania	2021-05-24

Spis treści :

1 Treść zadania

Napisać za pomocą języka C++ oraz biblioteki graficznej program wykorzystujący tematy opracowane na zajęciach.

Mój wybór padł na prostą grę wzorującą się na popularnych Space Invader`ach. Gra polega na przemieszczaniu się postacią od lewej do prawej strony i strzelanie do przeciwników zapierających z górnych części widocznego ekranu.

2 Analiza zadania

Zadanie polega na poprawnym zdefiniowaniu kla oraz użyciu odpowiedniej biblioteki graficznej, pozwalającej na zaimplementowanie potrzebnych animacji. Program od samego początku miał być gotową grą pozwalającą jedynie na małe zmiany, przeciwników czy poziomu trudności.

2.1 Wybór biblioteki

Biblioteką graficzną użytą w projekcie jest biblioteka SFML. Wybór ten padł po konsultacjach z innymi studentami oraz po przeczytaniu artykułów czy tutoriali w pierwszym pisaniu gier.

2.2 Klasy oraz zastosowane technologie

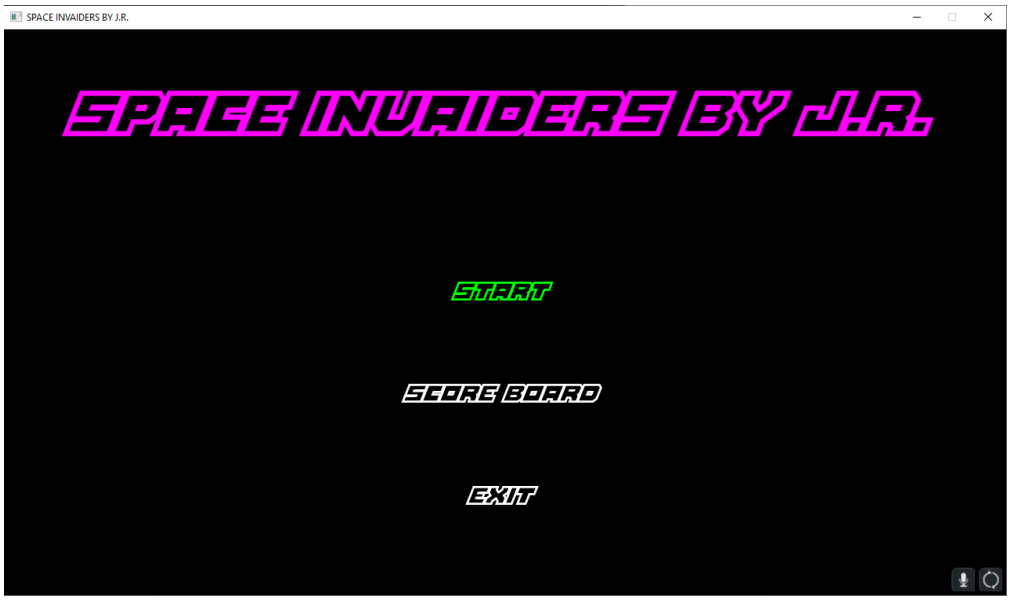
Klasy stworzone i użyte w projekcie:

- Entity - wirtualna klasa opisująca postacie
- Bullet - dziedzicząca z Entity, opisująca pociski wystrzeliwane przez wrogów czy gracza
- Enemy - dziedzicząca z Entity, opisująca przeciwników
  - Enemy 1 |
  - Enemy 2 | - Klasy dziedziczące po Enemy, przyszłościowo
  - Enemy 3 | mają pomagać w rozwijaniu gry.
- Controller - klasa posiadająca kontrolę nad plikami konfiguracyjnymi, jednocześnie będąc odpowiedzialną za ich poprawne ładowanie
- Game - klasa będąca szkieletem całej gry
- Destruction - klasa opisująca zniszczenia
- Menu - klasa odpowiadająca za menu gry
- Player - dziedzicząca z Entity, opisująca gracza
- Shield - dziedzicząca z Entity, opisująca tarcze

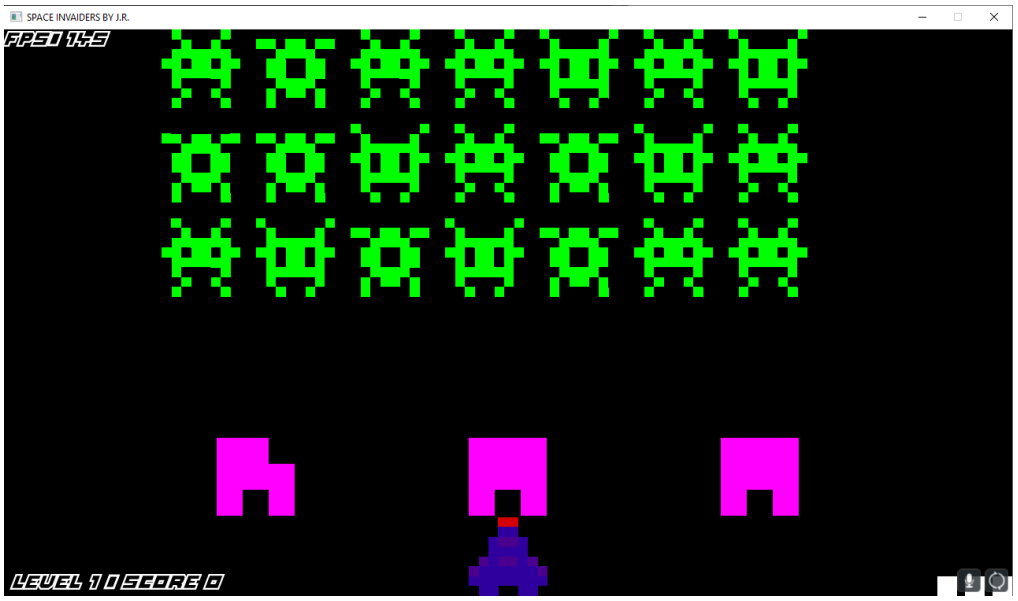
3 Specyfikacja zewnętrzna

Program wykonywany jest w nieskończonej pętli. Pętla ta kończy się gdy użytkownik zdecyduje się wyłączyć grę poprzez wybranie opcji “exit”, bądź naciśnięcie “x” na oknie.

Po uruchomieniu gry, użytkownikowi pokaże się menu główne:



Gdy użytkownik wybierze opcję pierwszą uruchomi grę:



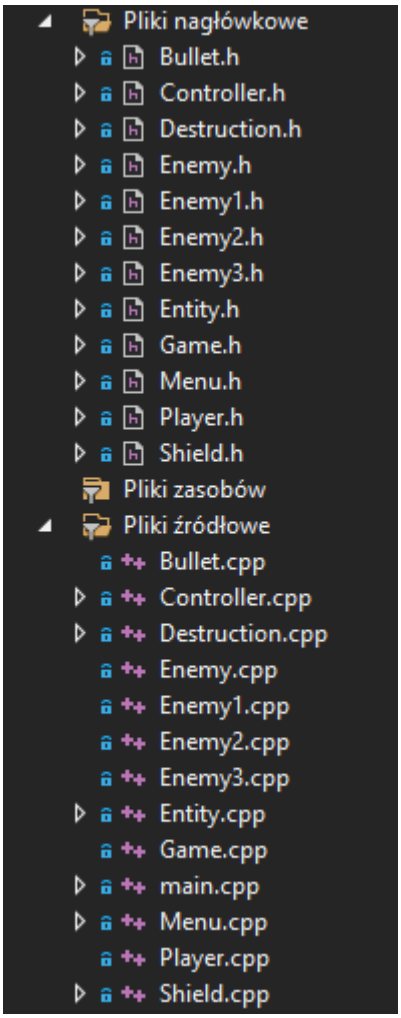
w trakcie gry gracz może zapauzować grę klikając przycisk “esc”, wtedy pokaże się to menu:

W menu głównym po wybraniu opcji “score board”, gracz będzie mógł zobaczyć najlepsze wyniki uzyskane podczas gier:

Gracz w podczas gry może się poruszać w lewo (przycisk “A”), w prawo (przycisk “D”) oraz strzelać do przeciwników (SPACJA). Celem gracza jest uzyskanie jak największej ilości punktów podczas jednej gry. W trakcie gry użytkownik nie może zapisać postępów by później do nich wrócić. Wyniki są automatycznie zapisywane podczas gdy gracz wyjdzie z gry, bądź straci wszystkie życia, które odnawiają się po każdym ukończonym poziomie. Zabezpieczeniem gracza przed atakami przeciwników są tarcze, znajdują się takie trzy na mapie i po każdym poziomie się odnawiają. Osłony są stworzone z 8 mniejszych, więc każda tarcza potrafi zatrzymać do 8 pocisków. Gracz również może owe tarcze niszczyć.

4 Specyfikacja wewnętrzna

Gra wymaga posiadania odpowiednich plików .ddl z biblioteki SFML.



Projekt został napisany w paradygmacie strukturalnym z zachowaniem odpowiednich podziałów plików, co widać na zrzucie powyżej.

4.1 Klasy, dziedziczenia, polimorfia

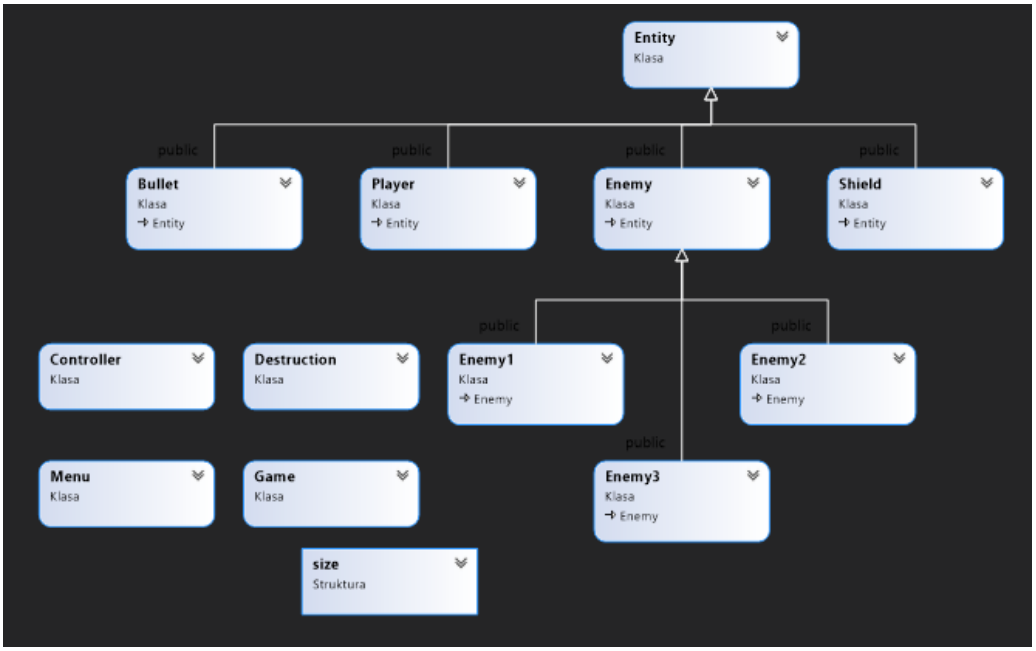


Diagram został wygenerowany za pomocą odpowiedniego narzędzia VS.

Jak można zauważyć na powyższym schemacie zostało użyte dziedziczenie jak również polimorfizm, co było najważniejszymi założeniami przy pisaniu projektu.

4.2 Ogólny schemat działania

Po uruchomieniu gry, program rysuje na ekranie odpowiednie obiekty takie jak postać gracza, tarczy czy wrogów:

```
void open_window() {
    while (window.isOpen())
    {
        //frame_delay
        Elapsed_frame_time = frame_clock.restart();
        //events
        user_events();
        anty_suprise_atack();
        event_game_over();
        new_level();
        //controls
        player_controll();
        bullet_move_controll();
        enemy_move_controll();
        enemy_shot_controll();
        colision_controll();
        //drawing
        window.setActive();
        window.clear();
        draw_objects_controll();
        draw_destructions_controll();
        draw_interface_controll();
        draw_menu_controll();
        window.display();
    }
}
```

W pętli widocznej na załączonym zrzucie widać wywoływanie odpowiednich metod odpowiedzialnych za prawidłowe działanie gry.

4.3 Najważniejsze funkcje i metody

Metoda “user_events()”, odpowiedzialna za sprawdzanie jakich przycisków aktualnie używa użytkownik oraz odpowiednie działanie po naciśnięciu czy zwolnieniu przycisku.

```
void user_events() {
    sf::Event event;
    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed) { ... }
        if (event.type == sf::Event::KeyPressed) {
            if (menu[1]) {
                if (event.key.code == sf::Keyboard::W) { ... }
                else
                    if (event.key.code == sf::Keyboard::S) { ... }
                    else
                        if (event.key.code == sf::Keyboard::Enter) { ... }
            }
            else
                if (menu[2]) {
                    if (event.key.code == sf::Keyboard::W) { ... }
                    else
                        if (event.key.code == sf::Keyboard::S) { ... }
                        else
                            if (event.key.code == sf::Keyboard::Escape) { ... }
                            else
                                if (event.key.code == sf::Keyboard::Enter) { ... }
                }
                else
                    if (menu[3]) { ... }
                    else
                        if (menu[4]) { ... }
                        else
                            if (menu[5]) { ... }
            }
            if (event.type == sf::Event::KeyReleased) {
                if (menu[3]) {
                    if (event.key.code == sf::Keyboard::A) { ... }
                    else
                        if (event.key.code == sf::Keyboard::D) { ... }
                        else
                            if (event.key.code == sf::Keyboard::Space) { ... }
                }
            }
        }
    };
```

Metoda “player_controll()” sprawdza jaki aktualnie przycisk jest naciskany przez gracza, a następnie porusza postacią na ekranie.

```
void player_controll() {
    if (!mapping["pause"]) {
        if (mapping["A"] and (player.get()->x - 1 * Elapsed_frame_time.asMilliseconds()) >= 0) { ... }
        if (mapping["D"] and (player.get()->x + 1 * Elapsed_frame_time.asMilliseconds()) <= static_cast<float>(window.getSize().x - player.get()->size.x) ) { ... }
        if (mapping["space"]) { ... }
    }
};
```

enemy_move_controll() odpowiada za prawidłowe

poruszanie się przeciwników. Gdy wrogowie poruszają się w prawo program sprawdza czy jakiś z nich nie doszedł do granicy okna, jeżeli tak to zmienia ich kierunek ruchu (enemy->direction *= -1;).

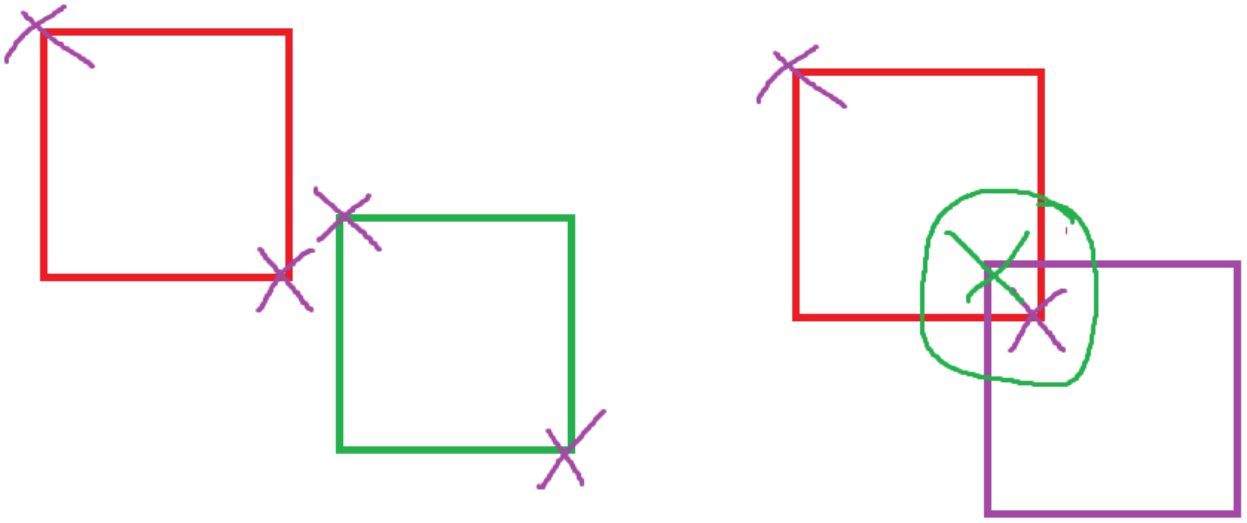
```
void enemy_move_controll() {
    if (!mapping["pause"]) {
        sf::Time enemy_move_time = enemy_move_clock.getElapsedTime();
        if (enemy_move_time.asSeconds() >= ((enemy_move)*sqrt(count)) / (sqrt(level))) {
            for (auto& obj : objects) {
                if (obj) {
                    auto enemy = dynamic_cast<Enemy*>(obj.get());
                    if (enemy) {
                        if (enemy->x + static_cast<float>(enemy->size.x / 2) * (enemy->direction) >= window.getSize().x - enemy->size.x ||
                            enemy->x + static_cast<float>(enemy->size.x / 2) * (enemy->direction) <= 0 ) {
                            for (auto& obj : objects) {
                                auto enemy = dynamic_cast<Enemy*>(obj.get());
                                if (enemy) {
                                    enemy->direction *= -1;
                                    enemy->y += static_cast<float>(enemy->size.y / 2);
                                    if (enemy->y >= player->y) {
                                        mapping["pause"] = true;
                                        health = 0;
                                    }
                                }
                            }
                        }
                    }
                }
            }
            for (auto& obj : objects) {
                if (obj) {
                    auto enemy = dynamic_cast<Enemy*>(obj.get());
                    if (enemy) {
                        enemy->x += static_cast<float>(enemy->size.x / 2) * (enemy->direction);
                        enemy->next_animation_frame();
                    }
                }
            }
```

Metoda widoczna poniżej wywołuje wewnętrzną metodę klasy “Entity” do sprawdzania kolizji dla każdego z obiektów istniejących na ekranie.

```
void colision_controll() {
    for (auto& obj1 : objects) {
        for (auto& obj2 : objects) {
            if (obj1.get() and obj2.get() and obj1 != obj2) {
                if (obj1->collision(obj2.get())) {
                    auto bullet = dynamic_cast<Bullet*>(obj1.get());
                    auto player = dynamic_cast<Player*>(obj2.get());
                    auto enemy = dynamic_cast<Enemy*>(obj2.get());
                    auto enemy1 = dynamic_cast<Enemy*>(obj1.get());
                    auto shield = dynamic_cast<Shield*>(obj2.get());
                    if (bullet and player) {
                        if (health) { ... }
                        else { ... }
                    }
                    else
                        if (enemy1 and player) { ... }
                        else
                            if (bullet and enemy) { ... }
                            else
                                if (bullet and shield) {
                                    if (shield->inside_collision(bullet)) { ... }
                                    else {
                                        bool destroy = true;
                                        for (auto box : shield->inside_boxes) {
                                            if (box) { ... }
                                        }
                                        if (destroy) { ... }
                                    }
                                }
                                else
                                    if (enemy1 and shield) { ... }
                }
            }
        }
    }
};
```

Metoda “collision()” sprawdza czy jeden z obiektów nie koliduje z drugim, poprzez porównanie rogów obu obiektów. Dzięki temu, że obiekty są w kształcie kwadratów możemy łatwo sprawdzać czy róg obiektu pierwszego nie jest pomiędzy rogami drugiego, jeżeli

jak jest zwraca wartość “true”:



```cpp
bool Entity::collision(Entity* colider)
{
    double _1lx = this->x;
    double _1ly = this->y;
    double _1rx = this->x + this->size.x;
    double _1ry = this->y + this->size.y;
    double _2lx = colider->x;
    double _2ly = colider->y;
    double _2rx = colider->x + colider->size.x;
    double _2ry = colider->y + colider->size.y;

    if (_1lx > _2rx || _2lx > _1rx)
        return false;
    if (_1ly > _2ry || _2ly > _1ry)
        return false;
    return true;
}
```

Czcionka, wyniki oraz tekstury. To są rzeczy które widzi użytkownik. Są one zatem często ważniejsze niż większość rzeczy w programie. Za ich poprawne ładowanie odpowiada klasa “Controller”.

Metody tejże klasy:

```
//ładowanie czcionki
bool Controller::load_font() { ... }
//wyników czcionki
bool Controller::load_scores() { ... }
//sortowanie oraz zapis wyników
bool Controller::add_sort_save_scores(int _score) { ... }
//ładowanie tekstur
bool Controller::load_textures() { ... }
```

Opis metody “add_sort_save_scores()”:

```
bool Controller::add_sort_save_scores(int _score)
{
    try {
        std::ofstream scores(path_scores, std::ofstream::out | std::ofstream::trunc);
        if (scores.is_open()) {

            //time \|
            auto end = std::chrono::system_clock::now();
            std::time_t end_time = std::chrono::system_clock::to_time_t(end);
            char buf[26];
            ctime_s(buf, sizeof(buf), &end_time);
            //time /|

            //add score \|
            std::string score;
            score += std::to_string(_score);
            score += " ";
            score += buf;
            score.pop_back();
            list_scores.push_back(score);
            //add score /|

            //sort by score \|
            std::map<int, std::string> to_sort;
            for (std::string e : list_scores) {
                std::stringstream line(e);
                int temp1;
                std::string temp2;
                std::string temp3;
                if (line >> temp1) {
                    while(line >> temp3)temp2 += " " + temp3;
                    to_sort.emplace(temp1, temp2);
                }
            }
            list_scores.clear();
            for (std::map<int, std::string>::iterator i = to_sort.begin(); i != to_sort.end(); i++)
            {
                std::string temp(std::to_string(i->first)+" "+i->second);
                list_scores.push_back(temp);
            }
            std::reverse(list_scores.begin(), list_scores.end());
            //sort by score /|

            for (std::string e : list_scores) {
                scores << e <<std::endl;
            }
        }
        else {
            throw std::runtime_error("No config.txt file found!\n");
        }
        scores.close();
        load_scores();
    }
    catch (std::runtime_error& e) {
        std::clog << e.what() << std::endl;
        return true;
    }
    return false;
}
```

Po odczycie z pliku linijek tekstu zapisywane są one do zmiennej typu “string”. Niestety w początkowych wersjach zapisywanie ciągów znaków do vectora nie przyniosło oczekiwanych rezultatów w sortowaniu danych. Na szczęście użycie mapy z biblioteki STL rozwiązało problem sortowania, gdyż mapa sama sortuje sobie dane po kluczu głównym, którym są wyniki gracza.

Dodatkowo kod został przygotowany pod testowanie i zmianę zmiennych na inne:

```
controller(_controller),
LOGO(controller.font, 60, 1280, 200, 0, 0),
menu_main(controller.font, 25, 1280, 520, 0, 200),
menu_pause(controller.font, 25, 1280, 720, 0, 0),
menu_game_over(controller.font, 60, 1280, 720, 0, 0),
menu_score_board(controller.font, 25, 1280 / 6, 720 / 6, 0, 0),
count(NUMBER_OF_ENEMYS),
max_count(count),
row(NUMBER_OF_ENEMYS_IN_ROW),
level(LEVEL_OF_BEGIN),
health(PLAYER_MAX_HEALTH),
score(0),
player_x(0),
player_y(0),
shield_number(NUMBER_OF_SHIELDS),
enemy_move(ENEMY_MOVMENT_DELAY),
enemy_shot(ENEMY_SHOT_DELAY),
player_shot(PLAYER_SHOT_DELAY),
bullet_speed(BULLET_SPEED)
```

4.4 Użyte tematy opracowane podczas laboratoriów.

1. Biblioteki STL. Oczywistym było wstawianie obiektów w vector. Mniej jednak użycie mapy w celu sprawdzania ruchu gracza czy odczytu jakie menu ma pokazywać program. Wybór ten jednak okazał się niemalże idealny.

```
std::vector<std::shared_ptr<Entity>> objects;
std::vector<Destruction> destructions;
```

```
std::map<std::string, bool> mapping = {
    {"A",false},
    {"D",false},
    {"space",false},
    {"pause",true}
};

/*
    1-main menu
    2-pause
    3-game
    4-game over
    5-score board
*/
std::map<int, bool> menu = {
    {1,true},
    {2,false},
    {3,false},
    {4,false},
    {5,false}
};
```

2. Inteligentne wskaźniki. Użycie ich pomogło w uniknięciu wycieków pamięci.

```
            objects.push_back(std::shared_ptr<Entity>(temp));
        }
        else
            if (random == 1) {
                auto temp = new Enemy2;
                temp->size.x = SIZE_X;
                temp->size.y = SIZE_Y;
                objects.push_back(std::shared_ptr<Entity>(temp
            }
            else
                if (random == 2) {
                    auto temp = new Enemy3;
                    temp->size.x = SIZE_X;
                    temp->size.y = SIZE_Y;
                    objects.push_back(std::shared_ptr<Entity>(
                }
        if (static_cast<int>(objects[i].get()->size.x * i )% r
        objects[i].get()->x += objects[i].get()->size.x * (i %
        objects[i].get()->y += objects[i].get()->size.y * y +
    }
    auto temp = new Player;
    temp->size.x = SIZE_X;
    temp->size.y = SIZE_Y;
    temp->x = player_x;
    temp->y = player_y;
    objects.push_back(std::shared_ptr<Entity>(temp));
    player = objects[objects.size() - 1];
    for (int i = 1; i <= shield_number; i++) {
        auto temp = new Shield;
        temp->size.x = SIZE_X;
        temp->size.y = SIZE_Y;
        objects.push_back(std::shared_ptr<Entity>(temp));
        objects[objects.size() - 1].get()->x = static_cast<flo
        objects[objects.size() - 1].get()->y = window.getSize(
    }
}
```

3. RTTI. Szybkie i bezpieczne wyszukiwanie odpowiednich obiektów.

```
objects[i].get()->x += objects[i].get()->size.x * (i % row) + static_cast<float>(objects[i].get()->size.x / 5) * (i % row);
objects[i].get()->y += objects[i].get()->size.y * y + static_cast<float>(objects[i].get()->size.y / 5) * y;
```

```
auto shield = dynamic_cast<Shield*>(obj.get());
auto p = dynamic_cast<Player*>(obj.get());
```

4. Wyjątki. Używane podczas sprawdzania plików konfiguracyjnych.

```
//ładowanie czcionki
bool Controller::load_font()
{
    try { ... }
    catch (std::runtime_error& e) {
        std::clog << e.what() << std::endl;
        return true;
    }
}
//ładowaanie wyników
bool Controller::load_scores()
{
    try { ... }
    catch (std::runtime_error& e) {
        std::clog << e.what() << std::endl;
        return true;
    }
    return false;
}
```

```
//sortowanie oraz zapis wyników
bool Controller::add_sort_save_scores(int _score)
{
    try { ... }
    catch (std::runtime_error& e) {
        std::clog << e.what() << std::endl;
        return true;
    }
    return false;
}
//ładowanie tekstur
bool Controller::load_textures()
{
    try { ... }
    catch (std::runtime_error& e) {
        std::clog << e.what() << std::endl;
        return true;
    }
    return false;
}
```

5. Wątki. Użyte do animacji zniszczeń.

```
std::thread(&Destruction::animation, &destructions[destructions.size() - 1]).detach();//creatin destruction animation
```

5 Testowanie

Podczas opracowywanie oraz pisania programu od początku był on przygotowywany by było jak najmniej błędów czy crashy. Jednak nie obyło się i tak bez jakiś wpadek. Najlepszym przykładem jest powstała w trakcie testowań metoda "anty_suprise_atack()". Chroni ona przed natychmiastowym strzałem przeciwnika.

```
void anty_suprise_atack() {
    if (!menu[3]) {
        bullet_shot_clock.restart();
        enemy_shot_clock.restart();
    }
};
```

Inne testy pokazywały, że gracz może ugrzęznąć w "ścianie". Powodem okazało się złe klatkowanie programu, gdzie przy zmniejszonych liczbach klatek gracz mógł połowicznie wejść w ścianę, a program nie pozwalał mu z niej wyjść. Zostało to także naprawione.

Wszystkie inne ewentualne błędy także zostały sprawdzone takie jak:

- Różne prędkości gracza/przeciwników
- Zniszczenia tarcz przez przeciwników
- Dotarcie przeciwników do dolnej granicy okna

Błędy, jeśli się pojawiały, zostawały od razu naprawiane oraz usuwane.

Program został sprawdzony, także pod kątem braku plików czy ich złym nazewnictwem. Gdy się tak dzieje program od razu powiadomi użytkownika o brakach w plikach.

Wycieki pamięci zostały sprawdzone - brak.

6 Wnioski

Projekt ten był dotychczas moim największym. Nigdy nie byłem z czegoś tak zadowolony jak po ukończeniu pierwszej, mam nadzieje że nie ostatniej, mojej gry. Nauczyłem się wielu przydatnych rzeczy oraz funkcji. Szczerze mówiąc dopiero przy tym projekcie zobaczyłem jak duży potencjał mają inteligentne wskaźniki oraz RTTI.

Nie tylko sam kod jest moim tworem, stworzyłem także tekstury do gry co było ciekawym doświadczeniem i świetną zabawą. Oczywiście nie obyło się bez pomocy internetu czy kolegów z roku lecz myślę, że i takie doświadczenie jest równie ważne, gdyż uczy jakich rzeczy szukać oraz o co pytać.

Pomimo wszystko wiem, że determinacja oraz myśl o zbliżającym się deadlinie pomogły mi zabrać się do projektu i nauczeniu się wszystkiego co zostało użyte przeze mnie w projekcie.

Literatura

1. „Język c++ - szkoła programowania” Stephan Prata
2. https://4programmers.net/
3. http://cpp0x.pl/
4. https://pasja-informatyki.pl/
5. https://stackoverflow.com/
6. tutoriale na youtube.com
7. https://en.cppreference.com/w/cpp/language
8. https://www.cplusplus.com