Gliwice 25.05.2019r.

Laboratorium z przedmiotu
Programowanie Komputerów 4

# Sprawozdanie z Projektu - 2d adventurer game

AEI Informatyka gr.2

Michał Nalepa

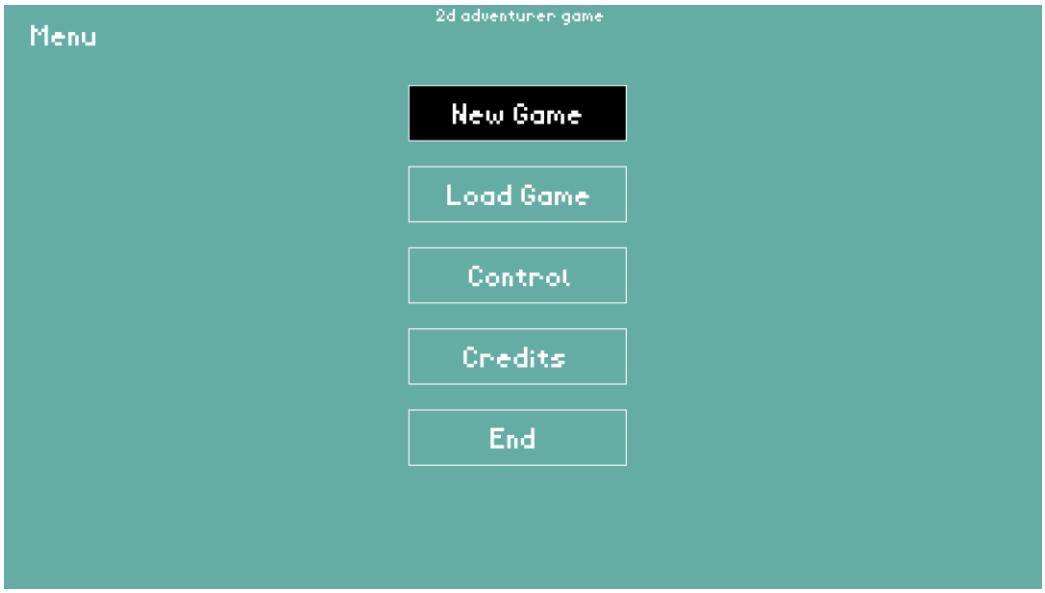
## 1. Ogólny opis projektu

Założeniem mojego projektu było stworzenie gry „2d adventurer game” w języku C++ z wykorzystaniem zaawansowanych technik języka oraz bibilioteki graficznej SFML. Zdecydowałem się na nią ponieważ jest to biblioteka typowo obiektowa, posiada przystępnie napisaną dokumentacje, a także dlatego że używałem jej już wcześniej. Gra jest platformówką 2d z systemem walki i elementami RPG takimi jak statystki postaci, ich ulepszanie, ekwipunek itp. Utrzymana w całości w stylu graficznym pixel-art. Grafika oparta na darmowych i legalnych assetach znalezionych w internecie. Projekt ten polegał bardziej na stworzeniu fizyki platformówki 2d (silnika) niż na budowie dużej ilości poziomów. W tej chwili dostępny jest jeden poziom aczkolwiek jego rozbudowa jak i budowa kolejnych jest bardzo prosta z wykorzystanie narzędzi które utworzyłem. Gra zawiera takie elementy jak wybór poziomu trudności, zapis oraz odczyt gry, mechanizm wyjątków w przypadku, gdy pliki gry są wybrakowane itp. Zaawansowane elementy C++ które wykorzystałem to kontenery stl, algorytmy i iteratory stl, unikalne wskaźniki oraz wyjątki. Więcej w punkcie 3. Aplikację można uruchomić na każdym komputerze z pliku .exe bez żadnych dodatkowych czynności. Jedynym wymogiem do poprawnego działania gry jest rozdzielczość full-hd (1920 × 1080).

## 2. Instrukcja obsługi (struktura zewnętrzna)

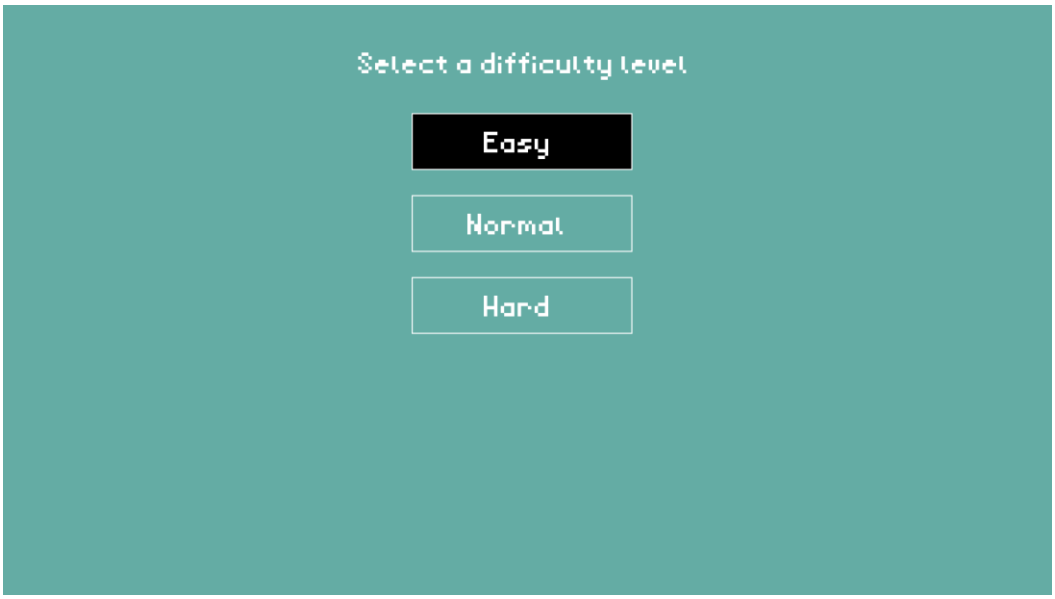
### - Menu

Po uruchomieniu gry pokazuje się okno menu. Składa się ono z 5 przycisków o nazwie opisującej czynność którą mają wykonać po wybraniu. Na czarno zaznaczony jest aktualnie wybrany prostokąt. Do poruszania się w górę i w dół wykorzystujemy przyciski W i S lub ↑ i ↓ na klawiaturze. Aby wybrać zaznaczoną opcje naciśnij Enter.



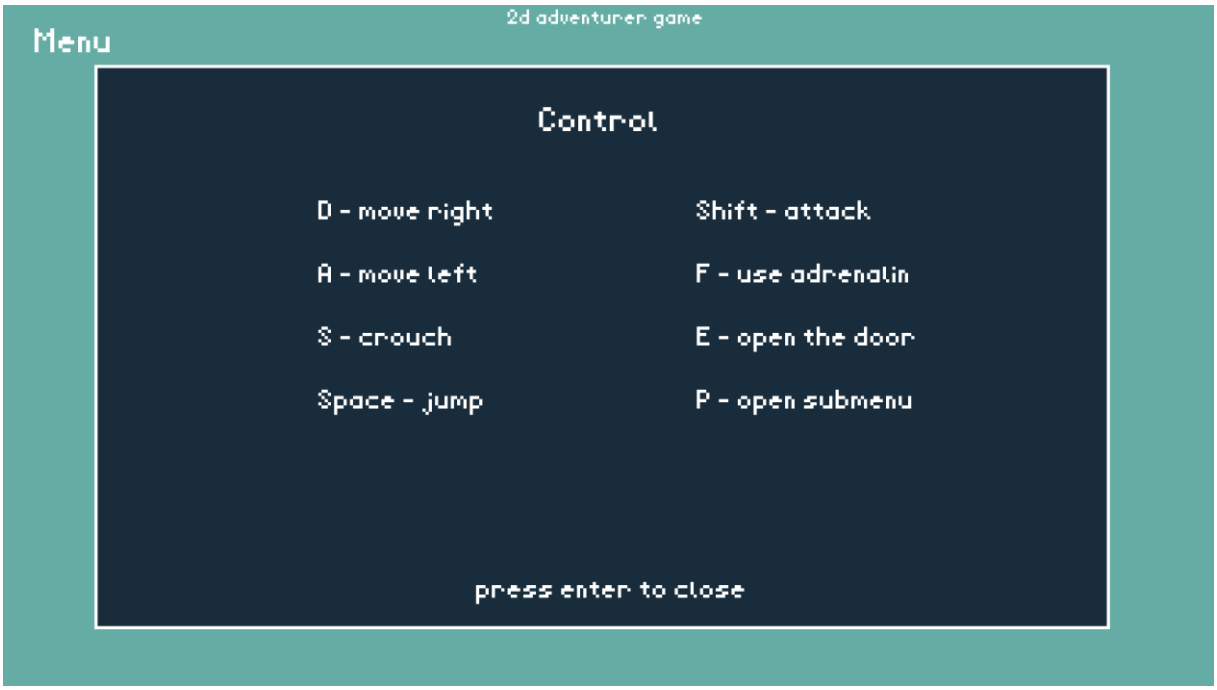

Opis poszczególnych opcji:

1. New Game – uruchamia nową grę. Kolejnym krokiem jest wybór poziomu trudności z dostępnych: łatwego, średniego lub trudnego.



2. Load Game – wczytuje zapisaną grę z pliku .txt. Kolejnym krokiem jest wybór czy gra ma być wczytana z manualnego zapisu, czy też automatycznego.


3. Control – pokazuje nam okno z opisanym sterowaniem. Warto się z nim zapoznać w celu bezproblemowej gry. Okno zamyka się naciśnięciem przycisku Enter.

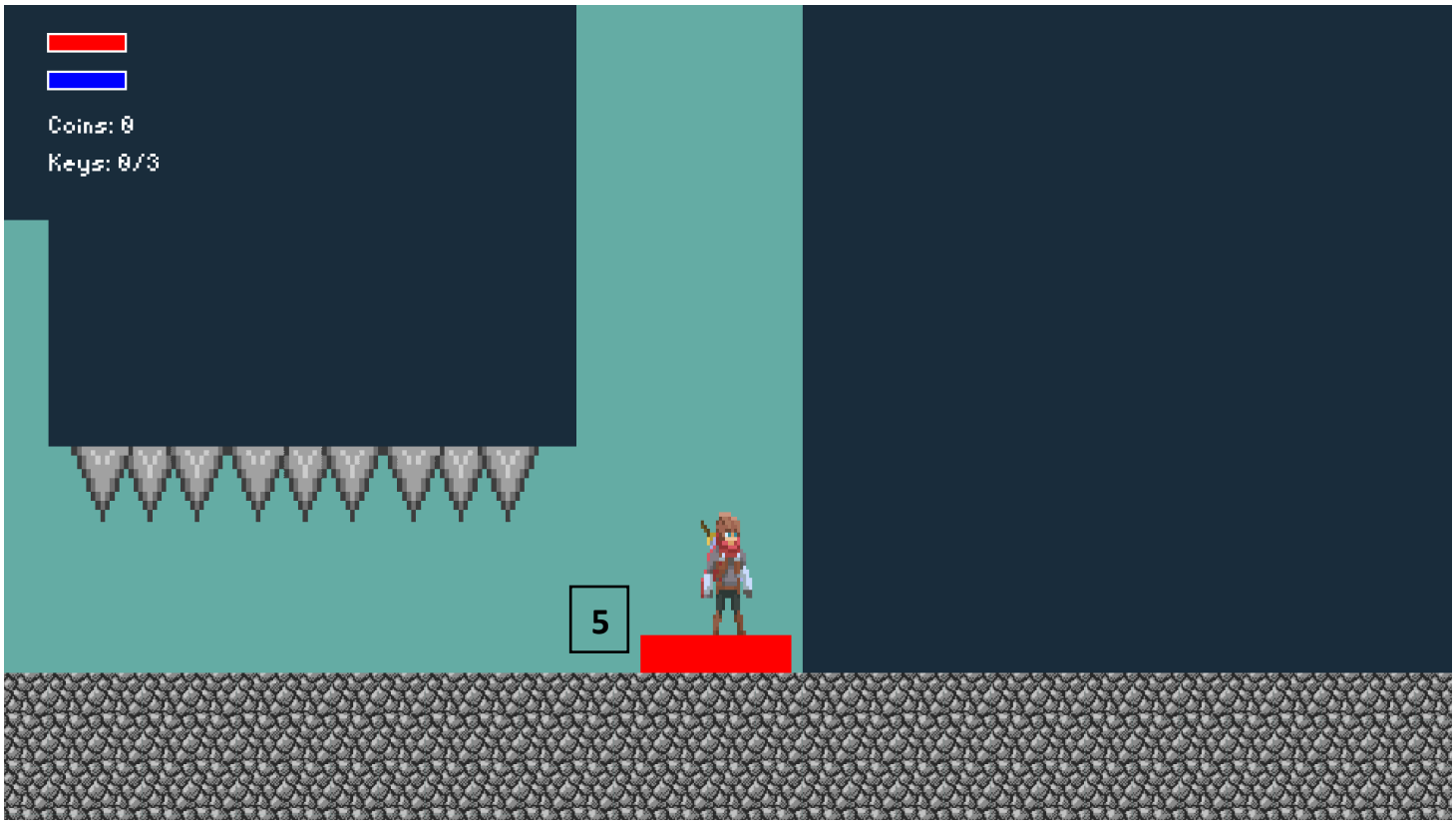


4. Credits – pokazuje nam okno z danymi autora jak i linkami do użytych grafik gry. Okno zamyka się naciśnięciem przycisku Enter.


5. End – zamyka grę

## - Gra

Po uruchomieniu gry pokazuje się nam jej okno





Gdzie:

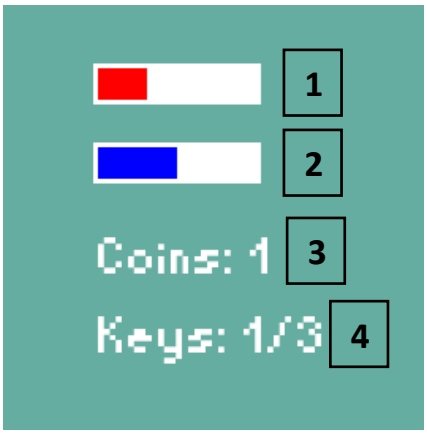
## 1. Bohater gry

Główny bohater gry którym sterujemy.

| |
|---|
| Sterowanie:<br>D - ruch w prawo<br>A - ruch w lewo<br>W, Space - skok<br>S - kucanie<br>Shift - atak<br>F - użycie adrenaliny<br>E - otwarcie drzwi<br>P, ESC - otwarcie podmenu |

Aby postać poruszała się podczas skoku należy trzymać naciśniety przycisk poruszania. Atak może być wykonywany tylko wtedy kiedy postać się nie porusza. Adrenalina powoduje że postać zadaje dwa razy więcej obrażeń.

## 2. Poglądowe statystyki




1. Na czerwono aktualny poziom zdrowia, na biało maksymalny. Maksymalny poziom zdrowia może być ulepszany w podmenu,

2. Na niebiesko aktualny poziom adrenaliny, na biało maksymalny. Maksymalny poziom adrenaliny może być ulepszany w podmenu,

3. Liczba posiadanych monet,

4. Liczba posiadanych kluczy na liczbę kluczy wymaganych do otwarcia drzwi.

## 3. Przedmioty

W grze są 3 rodzaje przedmiotów do zebrania:

- monety, za ich pomocą możemy wykupywać ulepszenia,

- serca, podnoszą nam aktualny poziom zdrowia,

- klucze, po znalezieniu ich wymaganej ilości możemy otworzyć drzwi, wypadają też z przeciwników,

Aby je zebrać wystarczy przez nie przejść.

## 4. Kolce

Jeżeli postać dotknie kolców z dowolnej strony - otrzymuje obrażenia i jest od nich odpychana.

## 5. Trampolina

Jeżeli będąc na niej naciśniemy przycisk skoku to uniesiemy się bardzo wysoko.

## 6. Przeciwnik

Nad przeciwnikiem znajduje się pasek z jego obecnym poziomem życia. Za pokonanego przeciwnika dostajemy klucz który jest niezbędny do otwarcia drzwi. Logiki działania przeciwnika nie opisuje, gdyż każdy gracz powinien odkryć ją sam 😉

(opisana w algorytmach)

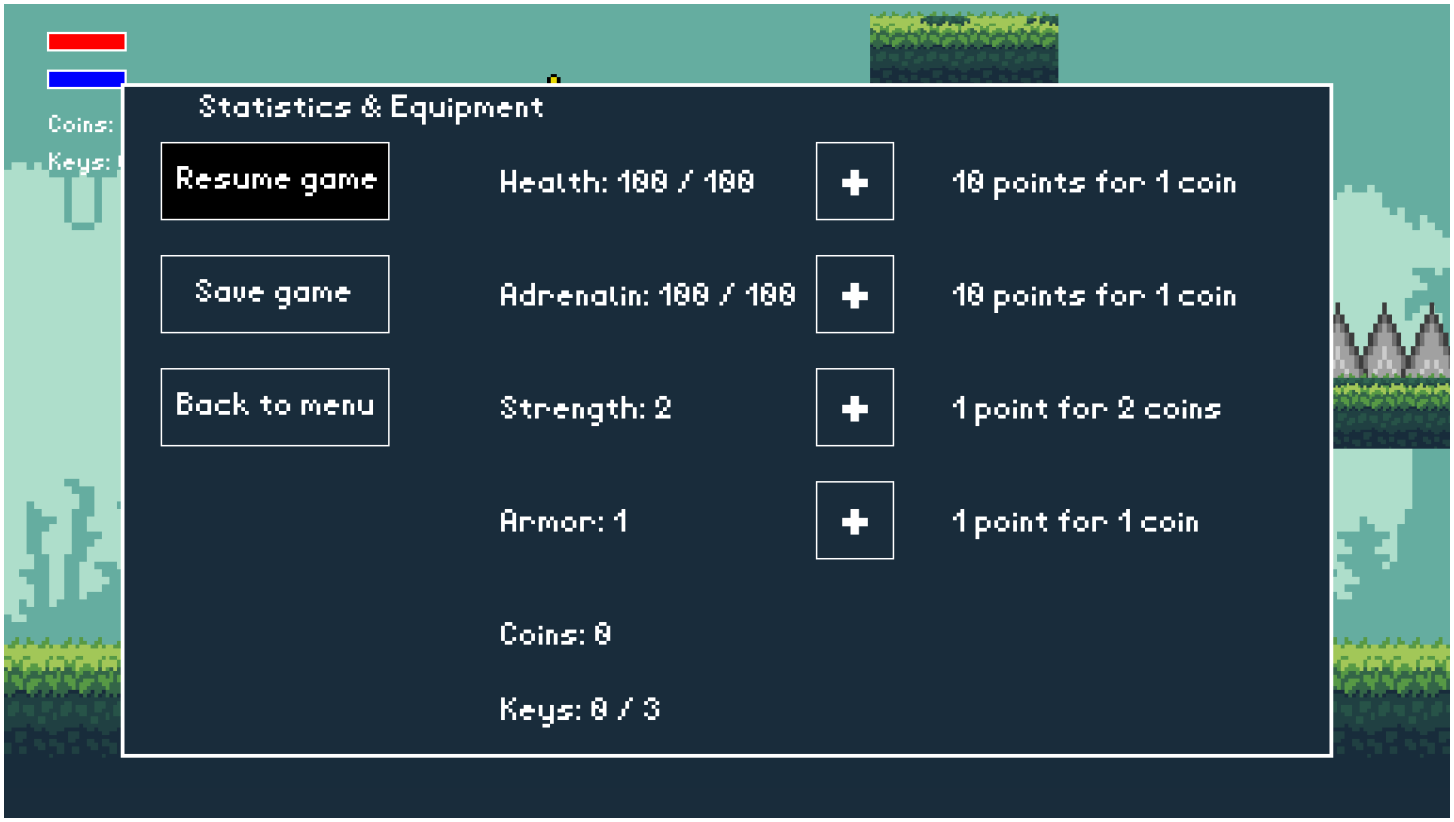
## 7. Drzwi

Drzwi którymi możemy dostać się do następnego poziomu. Aby je otworzyć musimy mieć wymaganą liczbę kluczy. Drzwi otwieramy klawiszem E. Po otwarciu przechodzimy przez nie także naciskając E.

## - Submenu

Submenu otwieramy za pomocą Esc lub P i w ten sam sposób je zamykamy.

Służy one do przeglądania szczegółowych statystyk w postaci liczbowej, kupna ulepszeń i sprawdzania ich cen, a także obsługi takich funkcjonalności jak zapis gry czy wyjście do menu.




Poruszanie się po submenu jak i wybór odpowiednich opcji jest identyczny jak w menu, z tym że można się także poruszać na prawo i lewo za pomocą A i D lub ← i →. Po zakupieniu pewnego ulepszenia pojawi się stosowny komunikat, jeżeli nie uda nam się go zakupić, gdyż mamy za mało monet gra będzie o tym informować.

### - Save i AutoSave

Grę możemy zapisać ręcznie z poziomu submenu, a oprócz tego gra zapisuje się automatycznie co 30s. Po zapisie gry pokazuje się odpowiedni komunikat przez parę sekund.

## 3. Struktura wewnętrzna

Projekt składa się z 23 klas. Ich funkcje i działanie są krótko opisane w poniższej tabeli:

| Klasa | Funkcja |
|---|---|
| Object | Jest podstawą do budowy wszelkiego rodzaju obiektów które znajdują się w grze. |
| Moving_Object: public Object | Dodaje do klasy Object parametry jak i metody służące do poruszania się obiektów. |
| Gate: public Object | Reprezentuje drzwi, ich animacje, logikę działania, kolizje itp. |
| Item: public Object | Reprezentuje przedmioty takie jak monety, serca i klucze do zebrania. |
| Animation | Uniwersalna klasa do animacji wykorzystywana przez główną postać, przeciwnika i poruszające się monety. |
| Background | Reprezentuje tło i zawiera metody do jego rysowania. |
| Game | Klasa zawiera główną pętle gry i w niej uruchamia metody innych klas. Obsługuje m.in. zapis gry czy śmierć postaci. Liczy także czas który jest przekazywany do innych metod. |
| Main_Character | Reprezentuje postać głównego bohatera, przechowuje atrybuty opisujące aktualne położenie i stan postaci. Obsługuje poruszanie się postaci. |
| Character_Stats | Przechowuje statystyki postaci takie jak stan zdrowia czy liczba posiadanych monet i zawiera metody do ich obsługi. |

| | |
|---|---|
| Graphical_Character_Stats | Stanowi graficzną interpretacje klasy Character_Stats. Są to poglądowe statystyki wyświetlane w lewym górnym rogu ekranu. |
| Environment | Klasa zawierająca wszystkie elementy otoczenia takie jak platformy, ziemia, przedmioty do zebrania, przeciwnicy, kolce itp. Ze względów optymalizacyjnych umieszczone są w niej także metody do kolizji i grawitacji, czyli duża część logiki gry. |
| Enemy | Klasa abstrakcyjna reprezentująca przeciwnika oraz jego logikę, kolizje itp. |
| Enemy_Stats | Klasa zawierająca statystki przeciwnika takie jak stan zdrowia czy siła i metody do ich obsługi. |
| Graphical_Enemy_Stats | Graficzna interpretacja klasy Enemy_Stats. Jest paskiem zdrowia wyświetlanym nad przeciwnikiem. |
| Skeleton : public Enemy | Reprezentuje przeciwnika w postaci szkieletu. Implementuje wszystkie odziedziczone metody czysto wirtualne klasy Enemy. |
| Game_Saving | Pozwala dokonać zapisu gry. Zapisuje m.in. poziom gry, położenie postaci, jej stan, wszystkie statystki, zebrane przedmioty, pokonanych przeciwników i otwarte drzwi. |
| Automatic_Game_Saving: public Game_Saving | Do klasy Game_Saving dodaje obsługę czasu konieczną do automatycznego zapisu. |
| Game_Loading | Pozwala na wczytanie stanu gry. |
| Menu | Klasa reprezentująca menu, jego logikę i uruchamiająca grę w trybie wybranym przez użytkownika. |
| Credits_Control | Klasa wyświetlająca sterowanie i linki do grafik. |
| Submenu | Jest to submenu uruchamiane podczas działania gry. Służy one do zarządzania statystykami, a także obsługi takich funkcjonalności jak zapis, czy wyjście do menu. |
| Improvment_Prices | Klasa zawierająca ceny ulepszeń. |

| | |
|---|---|
| Exceptions | Klasa obsługująca wyjątki, pokazująca odpowiedni komunikat na ekran i zamykająca grę. |

Wykorzystane zaawansowane elementy języka C++:

**1. Kontenery stl:**

W projekcie używam kontenerów: vector, list oraz forward_list.

vector – używam w przypadku gdy potrzebuje mieć szybki dostęp do każdego z elementów. Głównie przechowuje w nim teksty oraz prostokąty w menu i submenu.

forward list – używam do przechowywania statycznych elementów otoczenia które nie są nigdzie przekazywane, a ich logika polega na sprawdzaniu kolizji z bohaterem 60 razy na sekundę element po elemencie.

list – jak wyżej z tą różnicą że przechowuje w nim obiekty które są usuwane np. obiekty klasy Item. Usuwanie elementów z listy dwukierunkowej jest prostsze i szybsze niż z listy jednokierunkowej, gdyż nie musimy pamiętać poprzedniego elementu.

Przykładowe użycie kontenerów w klasie Environment:

Environment.h

```
class Environment
{
	std::forward_list<Object>collison_objects;
	std::forward_list<Moving_Object>moving_platforms;
	std::forward_list<Object>spikes;
	std::forward_list<Object>decorative_objects;
	std::list<Item>items;
	std::vector<sf::Texture> textures;
```

Environment.cpp

```
void Environment::create_level_1()
{
	delete_enviroment();
	level_number = 1;
	required_number_of_keys = 3;

	// size, position, rect, texture
	collison_objects.emplace_front(Object{ sf::Vector2f{ 50, 1000 }, sf::Vector2f{ -50, 0 },
 sf::IntRect{ 0, 0, 200, 40 }, &textures[1] }); // granica1
	float size_x = 1000, size_y;
	collison_objects.emplace_front(Object{ sf::Vector2f{ size_x, 200 }, sf::Vector2f{ 0, 900 },
sf::IntRect{ 0, 0, rect_ground(size_x), 40 }, &textures[1] }); // ziemia1
```

```
	collison_objects.emplace_front(Object{ sf::Vector2f{ 400, 400 }, sf::Vector2f{ 1000, 720 },
sf::IntRect{ 77, 13, 80, 88 }, &textures[2] }); // kwadrat1
	size_x = 1500;
	collison_objects.emplace_front(Object{ sf::Vector2f{ size_x, 200 }, sf::Vector2f{ 1400, 900 },
sf::IntRect{ 0, 0, rect_ground(size_x), 40 }, &textures[1] }); // ziemia2
	collison_objects.emplace_front(Object{ sf::Vector2f{ 600, 100 }, sf::Vector2f{ 1550, 550 },
sf::IntRect{ 0, 192, 160, 32 }, &textures[0] }); // platforma1
```

## 2. Algorytmy i iteratory stl

Elementy te wykorzystywane w moim projekcie są bardzo często, jakiekolwiek operacje na kontenerach wykonuje za pomocą iteratorów. Ciekawym wykorzystaniem iteratora jest wybrany element w menu który jest kolorowany na czarno i pozwala wybierać opcje. Wielokrotnie wykorzystuje także pętle zakresowe. Z algorytmów można wymienić metodę erase wykorzystywaną w menu przy przekształcaniu menu z 5 prostokątami wyboru do 2 prostokątów.

Environment.cpp

```
void Environment::draw(sf::RenderWindow &window, float &delta_time)
{
	for (auto &i : spikes)
		window.draw(i);

	if (gate)
	{
		gate->draw(window);
		if (gate->get_during_the_opening() == true)
		{
			if (gate->update_single_animation(delta_time) == false)
			{
				gate->set_during_the_opening(false);
				gate->set_is_open(true);
			}
		}
	}
	if(enemy) enemy->draw(window);

	for (auto &i : collison_objects)
		window.draw(i);

	for (auto &i : moving_platforms)
		window.draw(i);

	for (auto &i : decorative_objects)
		window.draw(i);

	for (auto &i : items)
	{
		if (i.get_type() == coin) i.Update(delta_time);
		window.draw(i);
	}
}
```

Menu.cpp

```
void Menu::transform_to_loading_menu()
{
	texts.erase(texts.end() - 4, texts.end());
	buttons.erase(buttons.end() - 3, buttons.end());

	buttons[0].setPosition(750, 300);
	buttons[1].setPosition(750, 450);

	texts[0].setString("Select the loading source");
	texts[0].setPosition (650, 75);
	texts[0].setCharacterSize(50);

	texts[1].setString("Manual Save");
	texts[1].setPosition(buttons[0].getPosition().x + 50,
buttons[0].getPosition().y + 20);

	texts[2].setString("Auto Save");
	texts[2].setPosition(buttons[1].getPosition().x + 80,
buttons[1].getPosition().y + 20);

	iterator->setFillColor(sf::Color::Transparent);
	--iterator;
	iterator->setFillColor(sf::Color::Black);
	iterator_clock.restart();
}
```

### 3. Inteligentne wskaźniki

W projekcie użyłem dwa inteligentne wskaźniki. Pierwszy wskazuje na obiekt klasy Gate, a drugi wykorzystywany przy polimorfizmie przeciwnika.

Environment.h

```
class Environment
{
	std::forward_list<Object>collison_objects;
	std::forward_list<Moving_Object>moving_platforms;
	std::forward_list<Object>spikes;
	std::forward_list<Object>decorative_objects;
	std::list<Item>items;
	std::unique_ptr<Gate>gate;
	std::vector<sf::Texture> textures;
	unsigned int required_number_of_keys;
	std::unique_ptr<Enemy>enemy;
```

Environment.cpp

```
	gate = std::make_unique<Gate>(required_number_of_keys, sf::Vector2f{ 7350, 575 }, &textures[8]);
	enemy = std::make_unique<Skeleton>(6500, 7900, sf::Vector2f{ 7150, 710 }, &textures[9]);
```

```
void Environment::enemy_support(Main_Character* character, float& delta_time)
{
	if (enemy)
	{
		enemy->collision(character);
		enemy->being_attacked(character);
		enemy->death_animation(delta_time);
		enemy->attacking(character, delta_time);
		enemy->attack_animation(delta_time);
		enemy->attacking(character, delta_time);
		enemy->movement(delta_time);
	}
}
```

Jak pokazuje powyższy przykład nie ma żadnych przeszkód aby stosować inteligentne wskaźniki do polimorfizmu.

### 4. Wyjątki

Wyjątki wykorzystuje do sprawdzenia czy wszystkie tekstury .png, pliki .txt i czcionka .ttf zostały wczytane poprawnie. Do ich obsługi mam utworzoną specjalną klasę Exceptions która pokazuje odpowiedni komunikat i zamyka program.

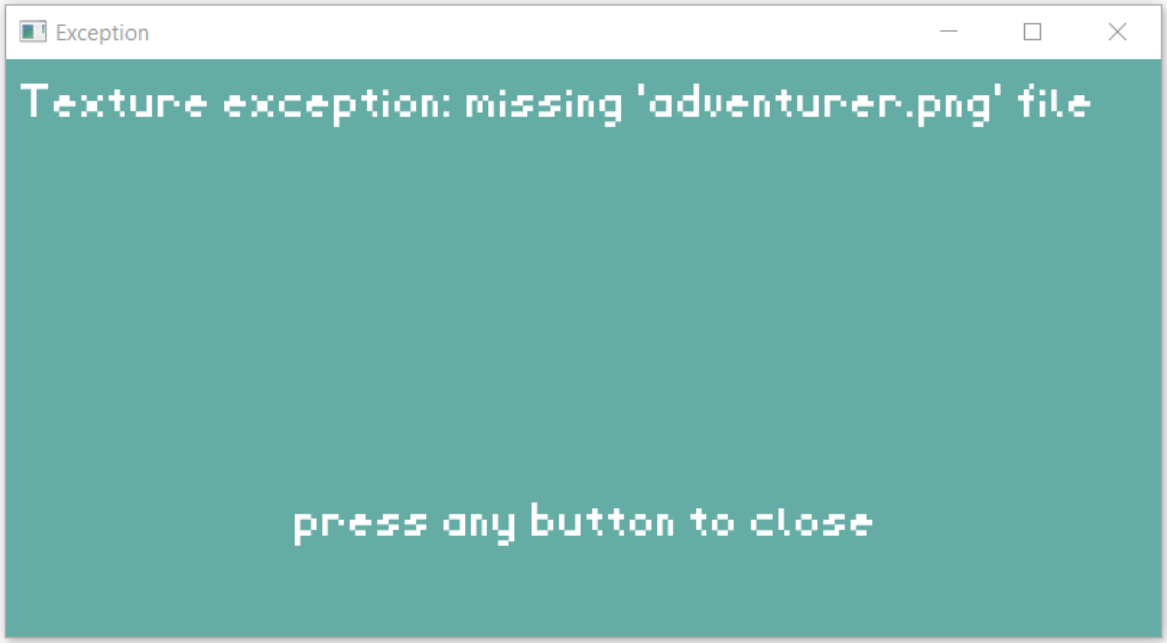
Environment.cpp

```
	//texture1 jungle
	textures.emplace_back(sf::Texture{});
	if (!textures[0].loadFromFile("jungle2.png"))
	{
		std::string exception = "jungle2.png";
		throw exception;
	}
	//texture 2 ziemia
	textures.emplace_back(sf::Texture{});
	if (!textures[1].loadFromFile("single.png"))
	{
		std::string exception = "single.png";
		throw exception;
	}
	textures[1].setRepeated(true);
	//texture 3 kwadrat
	textures.emplace_back(sf::Texture{});
	if (!textures[2].loadFromFile("square.png"))
	{
		std::string exception = "square.png";
		throw exception;
	}
	//coins
	textures.emplace_back(sf::Texture{});
	if (!textures[3].loadFromFile("coins.png"))
	{
		std::string exception = "coins.png";
		throw exception;
	}
	//heart
	textures.emplace_back(sf::Texture{});
```

```
	if (!textures[4].loadFromFile("heart.png"))
	{
		std::string exception = "heart.png";
		throw exception;
	}
	//spikes
	textures.emplace_back(sf::Texture{});
	if (!textures[5].loadFromFile("spikes.png"))
	{
		std::string exception = "spikes.png";
		throw exception;
	}
	textures[5].setRepeated(true);
	//stones
	textures.emplace_back(sf::Texture{});
	if (!textures[6].loadFromFile("stones.png"))
	{
		std::string exception = "stones.png";
		throw exception;
	}
	textures[6].setRepeated(true);
	//key
	textures.emplace_back(sf::Texture{});
	if (!textures[7].loadFromFile("key.png"))
	{
		std::string exception = "key.png";
		throw exception;
	}
	//gate
	textures.emplace_back(sf::Texture{});
	if (!textures[8].loadFromFile("gate.png"))
	{
		std::string exception = "gate.png";
		throw exception;
	}
	//skeleton
	textures.emplace_back(sf::Texture{});
	if (!textures[9].loadFromFile("skeleton.png"))
	{
		std::string exception = "skeleton.png";
		throw exception;
	}
}
```

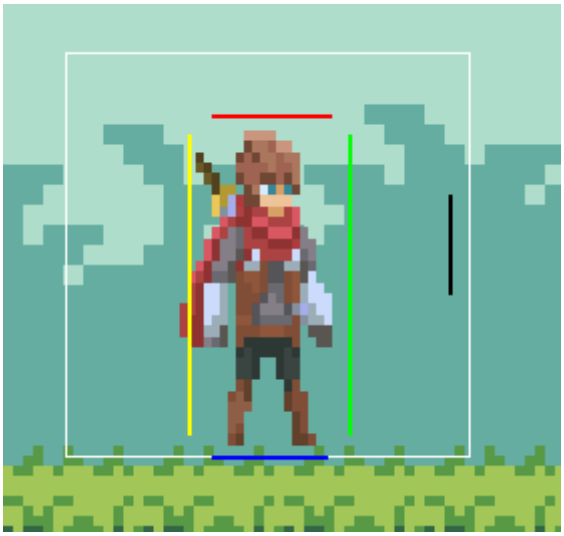
```
	try
	{
		if (!font.loadFromFile("RevMiniPixel.ttf"))
		{
			std::string exception = "RevMiniPixel.ttf";
			throw exception;
		}
	}
	catch (std::string file_name)
	{
		Exceptions exception;
		exception.font_file_exception(file_name);
	}
```

Omawiany wcześniej komunikat po usunięciu pliku adventurer.png z plików gry:




## Interesujące algorytmy czy rozwiązania warte omówienia

### 1. System kolizji



Wokół postaci są umieszczone 4 prostokąty na podstawie których za pomocą metody Intersects klasy sf::RectangleShape sprawdzane są kolizje. Dzięki temu jestem w stanie określić z której strony postać wchodzi w kolizje z obiektami i odpowiednio je obsłużyć. Aby ułatwić sobie programowanie „przeciążyłem" metodę move() oraz set_position() dla postaci w ten sposób że przemieszczając postać przemieszczamy automatycznie wszystkie 6 prostokątów. Czarny prostokąt reprezentuje kolizje miecza.

Długo zastanawiałem się czy lepiej przekazywać kontenery z obiektami do bohatera i aby to klasa bohatera sprawdzała kolizje, czy może przekazywać wskaźnik do bohatera do klasy Environement, gdzie znajdują się kontenery z obiektami i to tam sprawdzać kolizję. Ostatecznie zdecydowałem się na to

drugie rozwiązanie gdyż wydaje się znacznie bardziej optymalne. Przykładowa metoda sprawdzania kolizji i ich obsługi dla zwykłych nieruchomych obiektów kolizyjnych takich jak platformy czy ziemia:

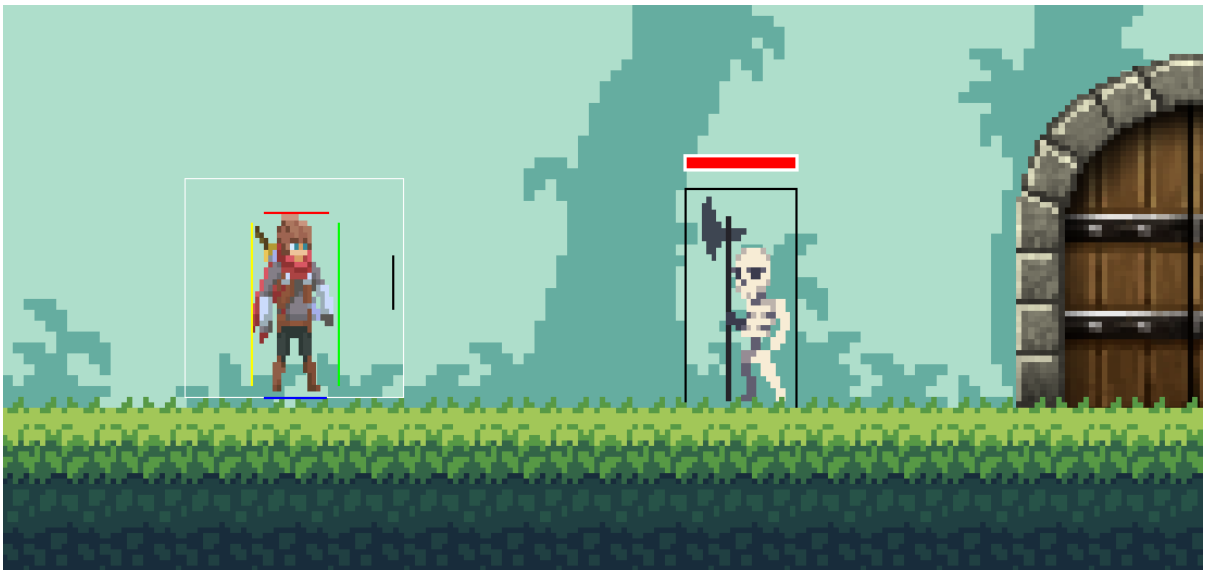
Environment.cpp

```
void Environment::collision(Main_Character *character)
{
	float movement_speed = character->get_movement_speed();
	for (auto &i: collison_objects)
	{
		if (character->get_right().getGlobalBounds().intersects(i.getGlobalBounds()) == true &&
character->get_left().getGlobalBounds().intersects(i.getGlobalBounds()) == true &&
character->get_down().getGlobalBounds().intersects(i.getGlobalBounds()) == true)
			character->move(0, -1 * movement_speed);
		if (character->get_top().getGlobalBounds().intersects(i.getGlobalBounds()) == true)
		{
			character->set_is_jumping(false);
			character->set_is_gravity(true);
			character->set_initial_jump_speed();
		}
		if (character->get_right().getGlobalBounds().intersects(i.getGlobalBounds()) == true)
		{
			if (character->get_is_jumping() == true) character->move(-2.2 * movement_speed, 0);
			else character->move(-1 * movement_speed, 0);

		}
		if (character->get_left().getGlobalBounds().intersects(i.getGlobalBounds()) == true)
		{
			if (character->get_is_jumping() == true) character->move(2.2 * movement_speed, 0);
			else character->move(movement_speed, 0);
		}
	}
}
```

**2. Logika przeciwnika**

Jeżeli nie znajdujemy się w pobliżu przeciwnika chodzi on spokojne od prawej do lewej strony w ustalonych granicach



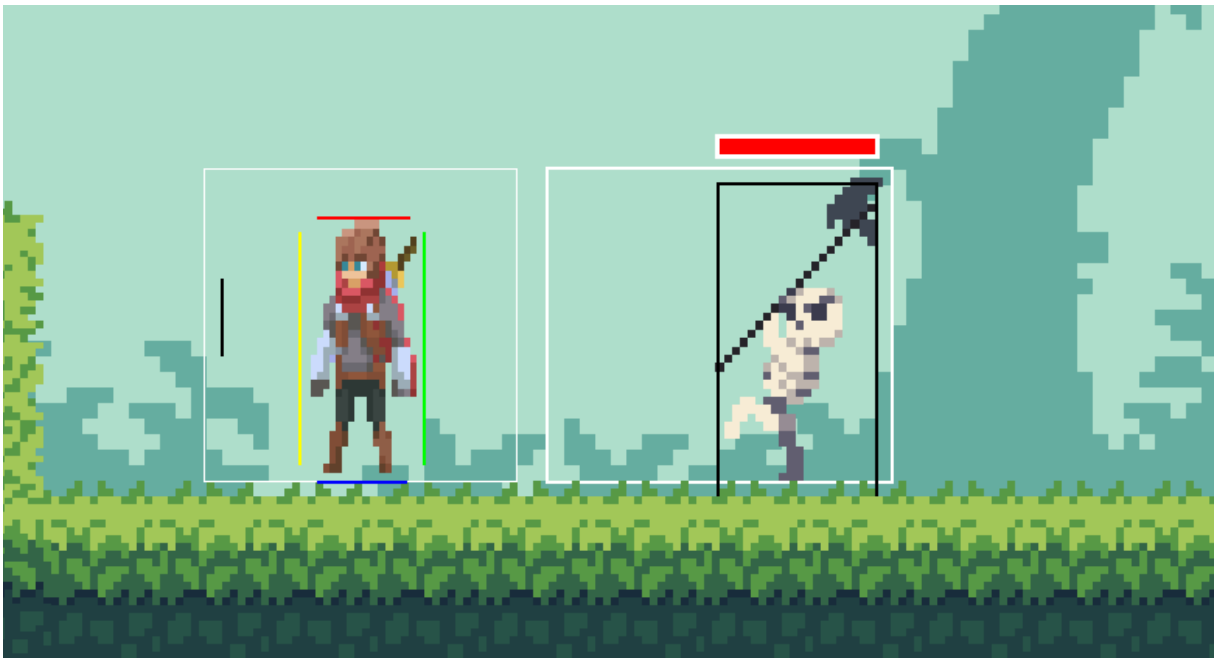
Metoda odpowiedzialna za poruszanie się szkieletu

Skeleton.cpp

```
void Skeleton::movement(float& delta_time)
{
	if (enemy_stats.get_alive() == true && enemy_stats.get_attack_animation() == false)
	{
		if (shape.getPosition().x <= x1)
		{
			movement_direction = true;
			move(movement_speed, 0);
		}
		else if (shape.getPosition().x + shape.getSize().y >= x2)
		{
			movement_direction = false;
			move(-1 * movement_speed, 0);
		}
		else
		{
			if (movement_direction == true)
			{
				move(movement_speed, 0);
				run.Upload(delta_time, shape, true);
			}

			else
			{
				move(-1 * movement_speed, 0);
				run.Upload(delta_time, shape, false);
			}
		}
	}
}
```

Szkielet tak naprawdę składa się z dwóch prostokątów shape(biały) i body(czarny). Przy jego swobodnym ruchu oba prostokąty się pokrywają. W przypadku gdy nasza postać zbliży się do przeciwnika prostokąt shape zwiększy swój rozmiar, aby móc wykonać animacje ataku. Prostokąt body pozostanie na swoim miejscu, ponieważ odpowiada on za kolizje bohatera z przeciwnikiem.



Metoda odpowiedzialna za rozpoczęcie ataku:

Skeleton.cpp

```
void Skeleton::attacking(Main_Character* character, float& delta_time)
{
	if (character->get_shape().getGlobalBounds().intersects(shape.getGlobalBounds()) == true &&
enemy_stats.get_alive() == true)
	{
		//rozpoczecie nowej animacji ataku
		if (enemy_stats.get_attack_animation() == false)
		{
			if (movement_direction == true)
			{
				if (shape.getPosition().x >= character->get_shape().getPosition().x)
movement_direction = false;
			}
			else
			{
				if (shape.getPosition().x <= character->get_shape().getPosition().x)
movement_direction = true;
			}
			enemy_stats.set_attack_animation(true);
			shape.setSize(sf::Vector2f(shape_size.x * 2.2, shape_size.y));
			if (movement_direction == true) shape.move(-10, -10);
			else shape.move(-110, -10);
			delta_time = 0.15f;
		}
		//sprawdzenie czy cios zadal obrazenia
		else if (character->get_left().getGlobalBounds().intersects(shape.getGlobalBounds()) == true
&& attack.get_animation_counter() == 8
			|| character->get_right().getGlobalBounds().intersects(shape.getGlobalBounds()) ==
true && attack.get_animation_counter() == 8)
		{
			character->subtract_health(enemy_stats.get_strength());
			character->set_hurt_color();
		}
	}
}
```

Z powyższego listingu można także wywnioskować w jaki sposób działa zadawanie obrażeń. Wykorzystujemy do tego metodę get_animation_counter() klasy animation która zwraca nam numer aktualnie wyświetlanej klatki animacji. W ten sposób przeciwnik zadaje nam obrażenia w czasie gdy animacja ataku jest pomiędzy 8, a 9 klatką animacji. Odwrotna sytuacja w której to my zadajemy obrażenia przeciwnikowi została napisana w analogiczny sposób.

### 3. Działanie iteratora w Submenu

Poniższa metoda przedstawia działanie zmiany prostokątu na który wskazuje iterator w klasie Submenu. Jak widać zmiany mogą być przeprowadzane tylko co iterator_period_time(0.15s) dzięki zastosowaniu clocka. Bez tego pojedyncze naciśnięcie przycisku S spowodowało by „zjechanie” iteratora na sam dół. Pierwszy if obsługuje ruch w górę, else if ruch w dół, a dwa kolejne odpowiednio w prawo i lewo.

Submenu.cpp

```
void Submenu::iterator_support()
{
	if (sf::Keyboard::isKeyPressed(sf::Keyboard::W) || sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
	{
		if (iterator_clock.getElapsedTime().asSeconds() > 0.15f)
		{
			if (iterator != buttons.begin())
			{
				iterator->setFillColor(sf::Color::Transparent);
				--iterator;
				iterator->setFillColor(sf::Color::Black);
				iterator_clock.restart();
			}
		}
	}
	else if (sf::Keyboard::isKeyPressed(sf::Keyboard::S) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
	{
		if (iterator_clock.getElapsedTime().asSeconds() > iterator_period_time)
		{
			if (iterator != --buttons.end())
			{
				iterator->setFillColor(sf::Color::Transparent);
				++iterator;
				iterator->setFillColor(sf::Color::Black);
				iterator_clock.restart();
			}
		}
	}
	else if (sf::Keyboard::isKeyPressed(sf::Keyboard::D) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
	{
		if (iterator_clock.getElapsedTime().asSeconds() > iterator_period_time)
		{
			if (iterator  == buttons.begin() || iterator == buttons.begin() + 1 || iterator ==
buttons.begin() + 2)
			{
				iterator->setFillColor(sf::Color::Transparent);
				iterator += 3;
				iterator->setFillColor(sf::Color::Black);
				iterator_clock.restart();
			}
		}
	}
	else if (sf::Keyboard::isKeyPressed(sf::Keyboard::A) ||
sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
	{
		if (iterator_clock.getElapsedTime().asSeconds() > iterator_period_time)
		{
			if (iterator == buttons.begin() + 3 || iterator == buttons.begin() + 4 || iterator ==
buttons.begin() + 5)
			{
				iterator->setFillColor(sf::Color::Transparent);
				iterator -= 3;
				iterator->setFillColor(sf::Color::Black);
				iterator_clock.restart();
			}
		}
	}
}
```

## 4. Diagram Klas

Diagram klas znajduje się pod linkiem:

https://go.gliffy.com/go/share/sp0hdrgbgbsr6ec66qpl

## 5. Źródła do grafik

https://rvros.itch.io/animated-pixel-hero - main character

https://jesse-m.itch.io/jungle-pack - jungle assets

http://pixelartmaker.com/art/e00f94d43b5c1ba - key

https://pezcame.com/ZG9vciBzcHJpdGU/ - door

https://jesse-m.itch.io/skeleton-pack - skeleton

http://pixelartmaker.com/art/dca2574a41f6294 - spikes

https://pixelsapphire.itch.io/pixel-16x16-heart - heart

## 6. Wnioski

Jest to zdecydowanie najbardziej złożony i skomplikowany program jaki napisałem jak i pierwsza prawdziwa gra. Pracowałem nad nią regularnie przez ponad 3 miesiące, co potwierdza historia commitów w serwisie github. Poświęciłem jej bardzo dużo czasu, ale nie żałuje gdyż wiele mnie nauczyła. Podczas pisania gry trzeba myśleć o innych rzeczach niż przy pisaniu innych aplikacji. Przede wszystkim spora część kodu wykonuje się 60 razy na sekundę i dlatego przy pisaniu gry tak ważna jest jej optymalizacja. Starałem się także oddzielić logikę od grafiki, na ile było to możliwe, np. poprzez podział na klasy Character_Stats oraz Graphical_Character_Stats. Pisanie gry było dla mnie znacznie ciekawsze od poprzednich projektów, gdyż efekty jak i postęp w mojej pracy były bardziej namacalne.