Labirynt 3D

Radosław Leluk
Konrad Malski
Damian Płóciennik

11 czerwca 2019

1 Tytuł projektu i autorzy projektu

Tytuł projektu to **Labirynt 3D**, jednak sam program nosi nadaną przez autorów anglojęzyczną nazwę **Maze 3D**.

Program został zrealizowany przez trzyosobowy zespół w składzie:

- Radosław Leluk
- Konrad Malski
- Damian Płóciennik

2 Opis projektu

Projekt zakłada stworzenie programu pozwalającego użytkownikowi na przejście labiryntu, który można stworzyć w edytorze plansz wbudowanym w program lub przygotować w dowolnym edytorze tekstowym. Program mierzy czas potrzebny na przejście labiryntu od punktu startowego do punktu końcowego. Dla uproszczenia przyjęto, że w labiryncie wszystkie ściany są do siebie prostopadłe oraz pojedyncze segmenty ścian mają tę samą długość. Ściany nie są teksturowane, lecz mogą różnić się od siebie kolorem. Jako motyw gry wykorzystano postać Jokera z serii komiksów oraz filmów Batman.

3 Założenia wstępne przyjęte w realizacji projektu

- Program wyświetla ściany labiryntu z perspektywy oczu użytkownika, który znajduje się w labiryncie. Ściany labiryntu są sześcianami i są do siebie prostopadłe.
- Program pozwala na wczytanie mapy z pliku tekstowego, w którym kolejne linie przedstawiają układ ścian.
- Program pozwala na wybranie ścian różnego koloru.
- Labirynt nie musi być w kształcie prostokąta - ważne aby tworzył wielokąt zamknięty oraz na zewnątrz jego ścian krańcowych nie znajdowały się żadne inne ściany.
- Program posiada własny kreator map pozwalający na ich łatwe i intuicyjne tworzenie. Sam kreator pozwala również na wczytanie gotowej mapy, jej modyfikacje i zapisanie do pliku.
- Jeżeli gracz podejdzie do punktu końcowego gra się kończy, a sam gracz dostaje informacje o czasie potrzebnym na pokonanie labiryntu.
- Sterowanie odbywa się za pomocą strzałek lub klawiszy "WSAD".
- Przytrzymanie klawiszu "Shift" zwiększa prędkość poruszania dwukrotnie.

- Kliknięcie klawisza "M" powoduje włączenie minimapy, która pokazuje labirynt oraz gracza jako rzut z góry.
- W menu gry gracz ma możliwość wyboru FOV (ang. Field of View - pole widzenia) w zakresie od 45 do 120 stopni.
- Grze towarzyszy muzyka, którą można włączyć lub wyłączyć w menu gry.
- Intuicyjność oraz prostota w obsłudze programu.

4 Analiza projektu

4.1 Specyfikacja danych wejściowych

Program pozwala na wczytanie pliku tekstowego, w którym zapisana jest mapa. Użytkownik nie określa w żaden sposób jak duża jest mapa - program sam odczytuje jej rozmiar. Kolejne linie pliku przedstawiają układ ścian, np.:

```
XXXXXEX
XSXX  X
X     X
XX    X
XXXXXXX
```

Litera "S" oznacza miejsce, w którym gracz startuje. Litera "E" oznacza wyjście z labiryntu, które reprezentuje ściana koloru żółtego. Pozostałe znaki oznaczają kolory różnych ścian, do wyboru są ściany koloru:

- "X" - czerwonego,
- "Y" - zielonego,
- "Z" - niebieskiego.

Wymagane jest:

- aby labirynt tworzył wielokąt zamknięty,
- aby na zewnątrz jego ścian krańcowych nie znajdowały się żadne inne ściany,
- aby znajdowały się w nim punkt startowy oraz końcowy.

Prócz znaków "X", "Y", "Z", "S", "E" oraz spacji w pliku nie mogą znajdować się żadne inne znaki. Puste miejsca powinny zostać uzupełnione spacjami. Spełnienie tych wymagań pozwala na pomyślne wczytanie mapy.

4.2 Opis oczekiwanych danych wyjściowych

Głównymi danymi wyjściowymi jest labirynt wyświetlany na ekranie monitora, który początkowo został wczytany z pliku lub stworzony w edytorze. Program pozwala również na pokazanie minimapy podczas rozgrywki, dzięki której gracz wie, co znajduje się w jego otoczeniu. Dodatkowo jako dane wyjściowe można uznać mapę labiryntu zapisywaną do pliku podczas korzystania z kreatora.

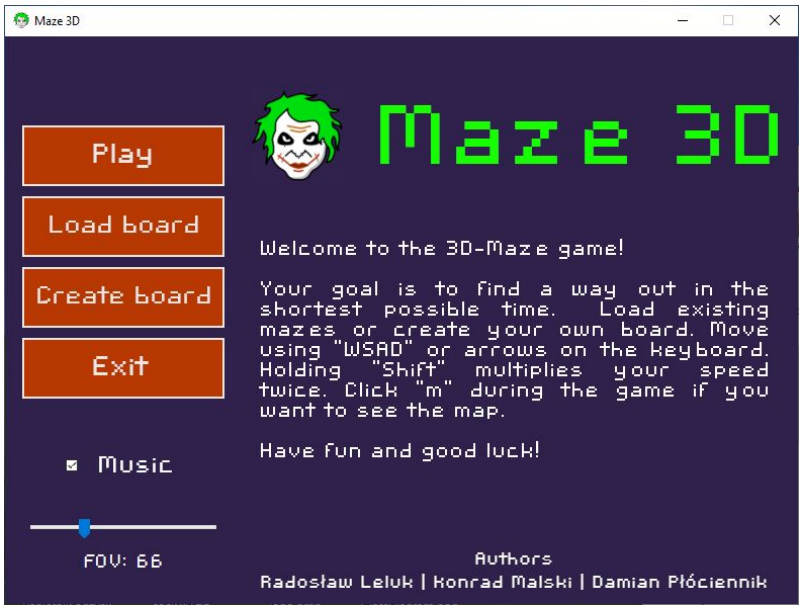
4.3 Zdefiniowanie struktur danych

Podczas działania programu mapa przechowywania jest w `std::vector`, co nie wymusza zapisywania jej rozmiaru w pliku. Grafiki użyte w programie przetrzymywane są jako `wxStaticBitmap`. Dla uproszczenia i przejrzystości kodu współrzędne przechowywane są jako `sf::Vector2<double>`.

4.4 Specyfikacja interfejsu użytkownika

Interfejs użytkownika różni się w zależności od okna, w jakim aktualnie użytkownik się znajduje. W menu:

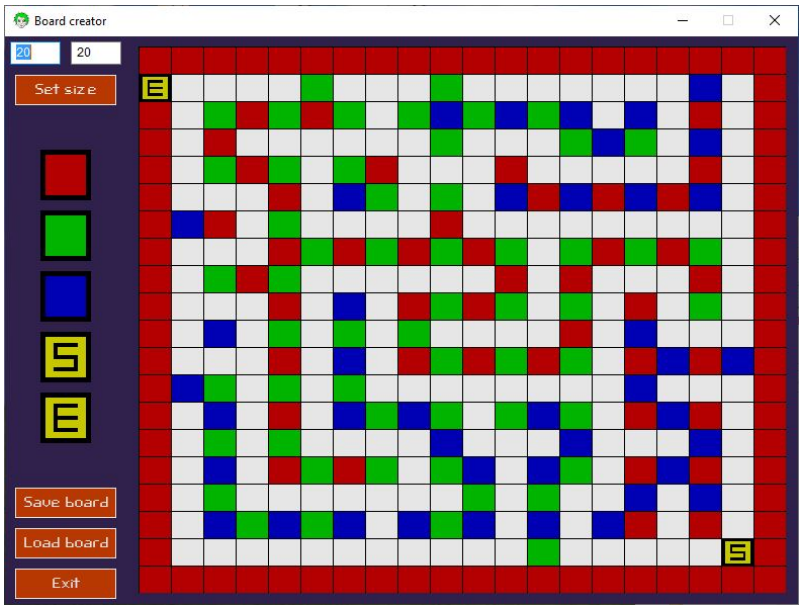
- przycisk **Play** pozwala na uruchomienie gry, jeśli wcześniej została wczytana poprawna mapa,
- przycisk **Load board** pozwala na wczytanie pliku z mapą,
- przycisk **Create board** pozwala na uruchomienie kreatora mapy,
- przycisk **Exit** pozwala na wyłączenie gry,
- checkbox **Music** pozwala na włączenie/wyłączenie muzyki odtwarzanej w czasie gry,
- slider **FOV** pozwala na ustawienie oczekiwanego pola widzenia z zakresu 45-120 stopni.



Rysunek 1: Menu gry

W kreatorze mapy:

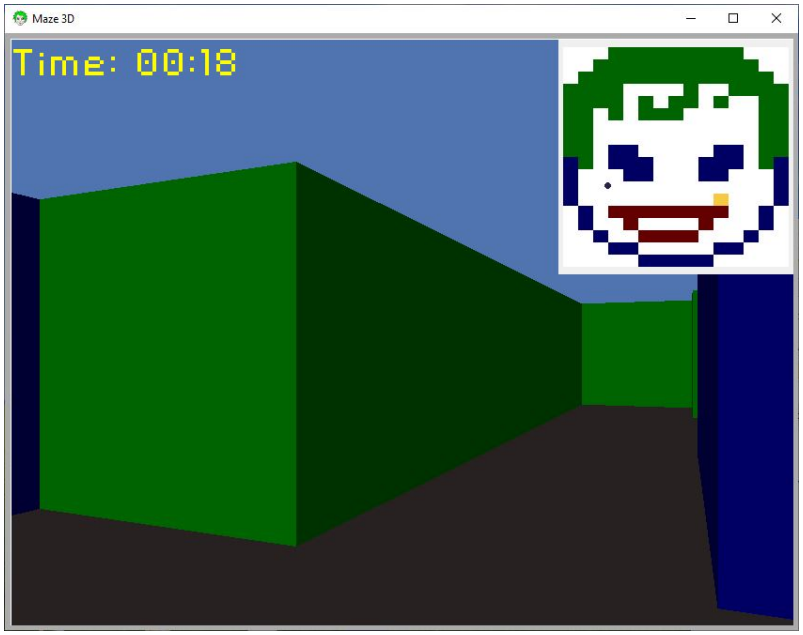
- przycisk **Set size** pozwala na ustawienie rozmiaru planszy wpisanego w pola nad przyciskiem,
- przyciski z kolorami odpowiednich ścian pozwalają na wybranie aktualnie używanej struktury,
- użytkownik może zabarwić dane pole kolorem wcześniej wybranym, klikając na nie lewym przyciskiem myszy lub przeciągając mysz z wciśniętym lewym klawiszem,
- użytkownik może zabarwić dane pole kolorem białym, reprezentującym brak przeszkód, klikając na planszę prawym przyciskiem myszy lub przeciągając mysz z wciśniętym prawym klawiszem,
- przyciski **Save board** oraz **Load board** pozwalają odpowiednio na zapisanie oraz wczytanie mapy do kreatora,
- przycisk **Exit** pozwala na wyjście z kreatora.



Rysunek 2: Kreator map z wczytaną przykładową mapą

Podczas trwania gry:

- przyciski **"WASD"** oraz klawisze **strzałek** umożliwiają poruszanie się,
- przytrzymanie klawisza **"Shift"** zwiększa dwukrotnie szybkość poruszania się,
- wciśnięcie klawisza **"M"** włącza/wyłącza minimapę.



Rysunek 3: Okno gry z wyświetloną minimapą. Wczytana została mapa w kształcie twarzy Jokera - ikony gry.

4.5 Wyodrębnienie i zdefiniowanie zadań

Głównym zadaniem było spełnienie wszystkich założeń. Wyodrębniono m.in. następujące zadania:

- stworzenie prostego i intuicyjnego GUI, aby użytkownik mógł z łatwością korzystać z programu,
- wymyślenie oraz implementacja wszystkich algorytmów potrzebnych do poprawnego działania programu, między innymi: algorytm wizualizacji labiryntu, algorytm odczytywania mapy z pliku, algorytm zapisywania do pliku, algorytm walidacji mapy gry, algorytm rysowania minimapy,
- stworzenie odpowiednich grafik i dobór kolorów, aby aplikacja była estetyczna i czytelna.

4.6 Decyzja o wyborze narzędzi programistycznych

Projekt został wykonany z wykorzystaniem języka C++ oraz dwóch bibliotek:

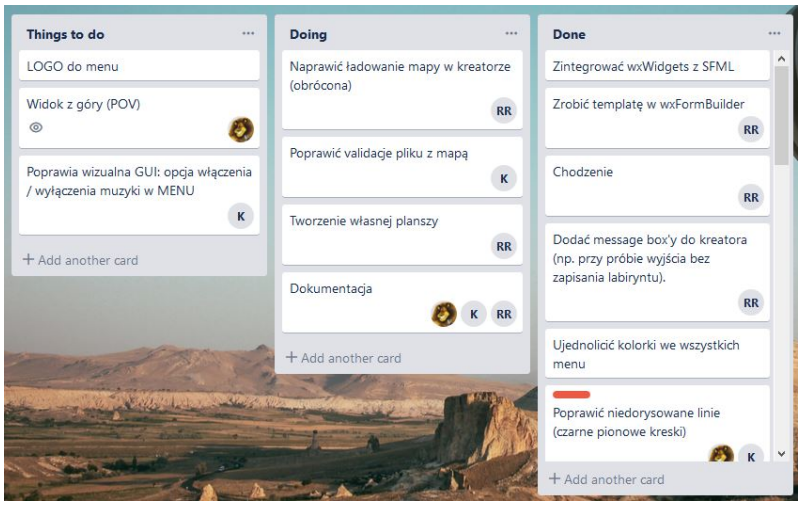
- wxWidgets (wersja 3.0.4),
- SFML (wersja 2.5.1).

SFML wymagał kompilacji statycznej z flagą dynamiczną ustawioną w Visual Studio, co wymusiło kompilację dynamiczną wxWidgets w celu poprawnego działania obu bibliotek jednocześnie.

Biblioteka SFML pozwoliła na bardzo wydajne rysowanie i implementację samego algorytmu rysowania, z kolei biblioteka wxWidgets dała możliwość prostego oraz szybkiego stworzenia GUI. Oba środowiska zostały wcześniej dobrze poznane podczas zajęć laboratoryjnych, a możliwość wykorzystania obu w jednym projekcie zdecydowała o ich wyborze.

Każdy z członków zespołu korzystał ze środowiska Visual Studio 2017.

W celu dobrej organizacji pracy zastosowano tablicę Kanban, którą stworzono na portalu Trello.com.



Rysunek 4: Tablica Kanban zrealizowana na portalu Trello

5 Podział pracy i analiza czasowa

Na początku dodano wszystkie konieczne biblioteki oraz przygotowano środowiska pracy. Następnie wyodrębnione zostały poszczególne zadania i zajęto się ich realizacją. Dzięki wykorzystaniu systemu kontroli wersji oraz tablicy Kanban etap programowania był bardzo efektywny, a poszczególne elementy programu były bez trudności łączone w całość.

Cele zrealizowane przez poszczególne osoby przedstawiały się następująco:

- Radosław Leluk - rysowanie labiryntu, podstawowe założenia gry, kreator mapy, zapisywanie mapy do pliku, GUI, dokumentacja,
- Konrad Malski - rysowanie labiryntu, podstawowe założenia gry, wczytywanie mapy oraz walidacja jej poprawności, GUI, dokumentacja
- Damian Płóciennik - rysowanie labiryntu, podstawowe założenia gry, minimapa, GUI, dokumentacja.

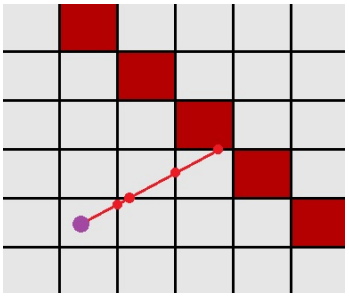
6 Opracowanie i opis niezbędnych algorytmów

6.1 Algorytm rysowania labiryntu

W celu zrealizowania głównego założenia, jakim było rysowanie labiryntu 3D z wykorzystaniem wyłącznie bibliotek 2D, zdecydowano się na wykorzystanie metody rzucania promieni, czyli tzw. ray castingu. Technika ta została wykorzystana m.in. w protoplaście strzelanek 3D, czyli grze Wolfenstein 3D.

Podstawowe założenie tego algorytmu polega na podzieleniu mapy na siatkę 2D, w której każde z pól oznaczone jest jako przeszkoda (ściana) lub jej brak. Praca algorytmu rozpoczyna się od przechodzenia po wszystkich pikselach wszerz oraz od wysyłania promienia zaczynającego się w miejscu, gdzie aktualnie znajduje się gracz. Kierunek "przemieszczania się" promienia zależny jest od kierunku, w który aktualnie skierowany jest gracz. Promień "przemieszcza się" do momentu, aż trafi na ścianę (występuje pętla sprawdzająca, czy promień znajduje się w ścianie). Kolejno obliczana jest odległość postaci od ściany, żeby potem móc narysować pionową linię, której wysokość zależna jest właśnie od odległości. Im ściana znajduje się dalej, tym krótsza linia zostanie narysowana.

Problemem okazuje się sprawdzanie kolizji ze ścianą. Człowiek jest w stanie od razu zobaczyć, gdzie promień napotyka ścianę. Komputer musi sprawdzać kolizję co pewien krok. Ustawienie zbyt dużego kroku może spowodować pominięcie ściany, z kolei ustawienie zbyt małego powoduje bardzo dużą ilość obliczeń, co przekłada się na wolne działanie programu. Rozwiązaniem tego problemu okazało się być sprawdzanie każdego boku ściany, z którą spotkać ma się promień. Każde pole ma szerokość równą 1, dzięki czemu wszystkie boki ściany są liczbami całkowitymi, a każde miejsce pomiędzy nimi jest liczbą zmiennoprzecinkową. Rozmiar kroku nie jest zatem stały, a zmienia się wraz z odległością do następnego boku.



Rysunek 5: Wizualizacja algorytmu sprawdzania kolizji

Dla lepszej estetyki oraz wyglądu programu, sprawdzane jest też z której strony znajduje się ściana (północ-południe czy wschód-zachód). Ściany z jednej ze stron mają przyciemniony kolor. Uzyskano w ten sposób znacznie bardziej uwydatnione krawędzie ścian, co pozwala na lepszą orientację w labiryncie.

6.2 Algorytm walidacji mapy

Na początku poszukiwany jest jakikolwiek segment ściany zewnętrznej labiryntu. Gdy zostanie on znaleziony ustawiana jest zmienna, która określa kierunek przemierzania mapy. Szukany jest kolejny segment ściany taki, który nie znajduje się za kierunkiem, który został obrany, np. poruszając się w lewo następny segment na jaki można przeskoczyć to: bezpośrednio nad poprzednim, u góry na skos, kolejny po lewej, na skos do dołu oraz bezpośrednio pod poprzednim:

```
XX
XO     <-   kierunek przemierzania w lewo
XX

X - możliwe wybory, O - poprzednia pozycja
```

Wybierany jest segment pierwszy od prawej strony kierunku przemierzania, czyli w przedstawionym przypadku bezpośrednio nad poprzednim. Na podstawie tego, w która stronę wykonano ruch, ustawiana jest zmienna kierunku przemierzania (w tym przypadku zostanie ustawiona na "do góry"). Pozwala to na przemierzenie całego labiryntu "mając" po lewej środek labiryntu. Ta część zakończy się, gdy algorytm ponownie dotrze do miejsca, w którym rozpoczął przemierzanie labiryntu. Końcowo sprawdzane jest, czy labirynt zawiera początek i koniec oraz czy w pliku nie występują żadne niedopuszczalne znaki.

Algorytm nie jest doskonały, ponieważ trzeba przyjąć założenie, że nie ma żadnych segmentów ścian poza zewnętrznymi krawędziami labiryntu. Jeśli algorytm natrafiłby na takie przed znalezieniem początku właściwego labiryntu, uznałby mapę za niepoprawną. Z kolei jeśli takie "wolne" segmenty nie przeszkodziłyby w znalezieniu początku właściwego labiryntu, walidacja mapy powiodłaby się.

7 Kodowanie

Klasy występujące w programie oraz ich metody i atrybuty to:

- **BaseBoardFrame**, **BaseMenuFrame**, **BaseMinimapPanel**, **BasePanelFrame** - klasy wygenerowane przez program wxFormBuilder, w których znajdują się wszystkie elementy interfejsu (kolejno - kreatora map, menu głównego, minimapy, panelu gry),
- **BoardFrame** - klasa kreatora, implementująca wszystkie funkcje potrzebne do jego obsługi. Dziedziczy interfejs po klasie bazowej BaseBoardFrame,
  - `BoardFrame(wxWindow* parent)` - konstruktor,
  - `void setSizeButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca rozmiar planszy w kreatorze po wciśnięciu przycisku "Set size",
  - `void startButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca element aktualnie rysowany na pole "Start" po wciśnięciu odpowiedniego przycisku,
  - `void endButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca element aktualnie rysowany na pole "Koniec" po wciśnięciu odpowiedniego przycisku,
  - `void redButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca element aktualnie rysowany na czerwoną ścianę po wciśnięciu odpowiedniego przycisku,
  - `void greenButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca element aktualnie rysowany na zieloną ścianę po wciśnięciu odpowiedniego przycisku,
  - `void blueButtonOnButtonClick(wxCommandEvent& event)` - metoda ustawiająca element aktualnie rysowany na niebieską ścianę po wciśnięciu odpowiedniego przycisku,

- `void loadButtonOnButtonClick(wxCommandEvent& event)` - metoda pozwalająca na wczytanie do kreatora mapy z pliku po wciśnięciu przycisku "Load board",
- `void saveButtonOnButtonClick(wxCommandEvent& event)` - metoda pozwalająca na zapisanie do pliku mapy z kreatora po wciśnięciu przycisku "Save board",
- `void exitButtonOnButtonClick(wxCommandEvent& event);` - metoda wyłączająca kreator mapy po wciśnięciu przycisku "Exit",
- `void frameOnClose(wxCloseEvent& event)` - metoda ostrzegająca użytkownika w chwili zamykania kreatora o niezapisanych zmianach, jeśli takie występują,
- `void onResize(wxSizeEvent& event)` - metoda skalująca w poprawny sposób kreator w przypadku zmiany rozmiaru,
- `void onLeftDown(wxMouseEvent& event)` - metoda dodająca aktualnie wybrany element do planszy po kliknięciu lewym przyciskiem myszy,
- `void onRightDown(wxMouseEvent& event)` - metoda usuwająca wybrany element z planszy po kliknięciu prawym przyciskiem myszy,
- `void onMotion(wxMouseEvent& event)` - metoda pozwalająca na dodawanie lub usuwanie elementów z planszy poprzez przeciągnięcie myszy z wciśniętym odpowiednim klawiszem,
- `bool computeIndex(int &i, int &j, const wxPoint &position)` - metoda sprawdzająca, czy aktualna pozycja myszy znajduje się w planszy,
- `void updateBox(int i, int j, const wxImage &img, char sign)` - metoda ustawiająca pole planszy na odpowiednie,
- `void prepareBoard()` - metoda przygotowująca i tworząca planszę,
- `void updatePosition()` - metoda aktualizująca pozycję segmentu,
- `void updateVariables()` - metoda aktualizująca wartości zmiennych, tj. rozmiar planszy, rozmiar pola oraz translację,
- `void draw()` - metoda rysująca planszę na ekran,
- `void failedLoadingScheme(std::string)` - metoda wyświetlająca informację o wczytaniu nieprawidłowej mapy do kreatora,
- `wxWindow *_parent` - rodzic obiektu BoardFrame,
- `wxImage _redImg` - obrazek przedstawiający czerwone pole,
- `wxImage _greenImg` - obrazek przedstawiający zielone pole,
- `wxImage _blueImg` - obrazek przedstawiający niebieskie pole,
- `wxImage _startImg` - obrazek przedstawiający pole startu,
- `wxImage _endImg;` - obrazek przedstawiający pole końca,
- `wxImage _floorImg;` - obrazek przedstawiający pole podłogi,
- `wxImage _currentImg` - obrazek przedstawiający aktualnie wybrane pole,
- `wxSize _panelSize` - rozmiar planszy,
- `wxSize _boxSize` - rozmiar pola,
- `wxPoint _translation` - translacja, o którą przesunięta jest plansza, aby była wyświetlona w odpowiednim miejscu okna,
- `char _currentSign` - aktualnie wybrany element,
- `bool _isStart` - flaga określająca, czy na planszy został umieszczony start,
- `bool _isEnd` - flaga określająca, czy na planszy został umieszczony koniec,
- `bool _boardChanged` - flaga określająca, czy na planszy zostały dokonane jakieś zmiany,
- `int _rows` - liczba wierszy,
- `int _columns` - liczba kolumn,
- `struct BoardBox` - struktura reprezentująca jedno pole planszy,

  - `std::vector<std::vector<BoardBox>> _board;` - wektor wektorów przechowujący wszystkie pola planszy.

- **MenuFrame** - klasa reprezentująca menu główne, wraz z funkcjami realizującymi przyciski. Odpowiednie funkcje mają za zadanie otwieranie okien kreatora czy głównego panelu gry. Dziedziczy interfejs po klasie BaseMenuFrame.
  - `void _playButtonOnButtonClick(wxCommandEvent& event)` - metoda rozpoczynająca rozgrywkę po wciśnięciu przycisku "Play", jeśli została wczytana poprawna mapa, w przeciwnym wypadku informująca o błędzie,
  - `void _loadBoardButtonOnButtonClick(wxCommandEvent& event)` - metoda pozwalająca wczytać wybraną z pliku mapę po wciśnięciu przycisku "Load Board",
  - `void _createBoardButtonOnButtonClick(wxCommandEvent& event)` - metoda uruchamiająca kreator map po wciśnięciu przycisku "Create Board",
  - `void _exitButtonOnButtonClick(wxCommandEvent& event)` - metoda wyłączająca program po wciśnięciu przycisku "Exit",
  - `void _FOVSliderOnScroll(wxScrollEvent& event)` - metoda zmieniająca FOV gracza w zależności od wartości na sliderze,
  - `void _musicCheckBoxOnCheckBox(wxCommandEvent& event)` - metoda włączająca/wyłączająca muzykę w zależności od tego, czy checkbox jest zaznaczony czy nie.

- **MinimapPanel** - klasa reprezentująca minimapę. Dziedziczy interfejs po klasie BaseMinimapPanel.
  - `MinimapPanel(wxWindow* parent, const sf::Vector2<double>& playerPosition)` - konstruktor,
  - `void _minimapPanelOnEraseBackground(wxEraseEvent& event)` - jawnie nadpisana metoda zapobiega rysowaniu przez wxWidgets, co mogłoby powodować migotanie,
  - `void draw()` - metoda rysująca minimapę na ekran,
  - `wxSize _boxSize` - rozmiar pojedynczego segmentu na minimapie,
  - `wxPoint _translation` - translacja, o którą przesunięta jest minimapa, aby była wyświetlona na środku panelu,
  - `const sf::Vector2<double>& _playerPosition` - stała referencja do aktualnej pozycji gracza,
  - `struct FieldBox` - struktura reprezentująca pojedynczy segment minimapy,
  - `std::vector<std::vector<FieldBox>> _fields;` - wektor wektorów przechowujący wszystkie pola minimapy.

- **PanelFrame** - klasa reprezentująca główny panel gry, na którym rysowany jest labirynt. Dziedziczy interfejs po klasie BasePanelFrame,
  - `PanelFrame(wxWindow* parent)` - konstruktor,
  - `void onResize(wxSizeEvent& event)` - metoda wywołująca się przy zmienie rozmiaru okna PanelFrame,
  - `void frameOnClose(wxCloseEvent& event)` - metoda wywołująca się przy zamykaniu okna PanelFrame,
  - `std::unique_ptr<Canvas> _canvas` - unikatowy wskaźnik na obiekt Canvas, w którym rysowany jest cały labirynt,
  - `wxWindow *_parent` - rodzic obiektu PanelFrame.

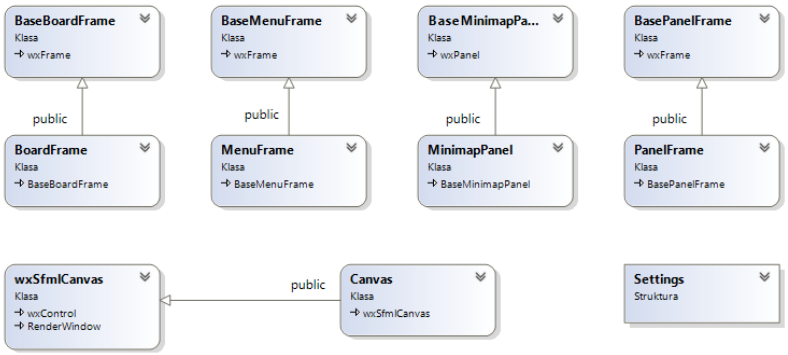
- **wxSfmlCanvas** - klasa umożliwiająca połączenie bibliotek wxWidgets oraz SFML,
  - `wxSfmlCanvas(wxWindow *parent = nullptr, wxWindowID windowId = -1, const wxPoint &positio` - konstruktor,

- `void onIdle(wxIdleEvent& event)` - metoda wykonująca się, kiedy okno pozstaje bezczynne,
- `void onPaint(wxPaintEvent& event)` - metoda wywołująca się, kiedy okno musi zostać przerysowane,
- `void onEraseBackground(wxEraseEvent& event)` - jawnie nadpisana metoda zapobiega rysowaniu przez wxWidgets, co mogłoby powodować migotanie.
- `void createRenderWindow()` - metoda tworząca obszar rysowania,
- `virtual void onUpdate()` - wirutalna metoda wywoływana podczas zmiany stanu okna,
- `virtual ~wxSfmlCanvas()` - destruktor.

- **Canvas** - klasa, w której zaimplementowany został algorytm rysowania labiryntu. Dziedziczy po klasie bazowej wxSfmlCanvas.
  - `Canvas(wxWindow* parent, wxWindowID id, wxPoint position, wxSize size, long style = 0)` - konstruktor,
  - `~Canvas()` - destruktor,
  - `void onResize(wxSizeEvent& event)` - metoda wywoływana podczas zmiany rozmiaru okna, pozwalająca na poprawne skalowanie,
  - `virtual void onUpdate()` - wirutalna metoda wywoływana podczas zmiany stanu okna,
  - `void drawBackground()` - metoda rysujaca tło,
  - `void drawMaze()` - metoda rysująca aktualnie widoczną część labiryntu,
  - `void move(double moveSpeed, int multiplier)` - metoda przemieszczająca gracza z podaną predkościa w przód lub tył (multiplier = 1 w przód, multiplier = -1 w tył),
  - `void rotate(double rotSpeed, int multiplier)` - metoda obracająca gracza w odpowiednią stronę (multiplier = 1 w lewo, multiplier = -1 w prawo),
  - `void calculateStepAndSideDist(sf::Vector2<double> &sideDistance, sf::Vector2<int> &step,` - metoda obliczająca długość promienia od aktualnej pozycji do następnej strony x lub y ściany oraz kierunek kroku (1 lub -1),
  - `void findCollision(sf::Vector2<double> &sideDistance, const sf::Vector2<int> &step, const` - metoda znajdująca ścianę, z która koliduje promień,
  - `sf::Color pickColor(const sf::Vector2<int> &mapBox, int side)` - metoda wybierająca odpowiedni kolor do pomalowania ściany,
  - `std::string prepareTimeString(double time)` - metoda przygotowująca łańcuch znaków reprezentujący aktualny czas spędzony w labiryncie,
  - `sf::Clock _clock` - zegar pomagający w odmierzaniu czasu spędzonego w labiryncie oraz utrzymaniu stałej ilości klatek na sekundę (FPS),
  - `sf::Vector2<double> _direction` - aktualny kierunek gracza,
  - `sf::Vector2<double> _playerPosition` - aktualna pozycja gracza,
  - `sf::Vector2<double> _end` - pozycja wyjścia z labiryntu,
  - `sf::Vector2<double> _cameraPlane` - zmienna określająca płaszczyznę kamery,
  - `sf::Font _font` - używana w grze czcionka,
  - `sf::Text* _timeText` - tekst reprezentujący czas spędzony w labiryncie,
  - `double _time = 0.` - czas rozpoczęcia rysowania klatki,
  - `double _oldTime = 0.` - czas rozpoczęcia rysowania poprzedniej klatki,
  - `MinimapPanel*  _minimap` - minimapa,
  - `bool _isActiveMinimap = false` - zmienna mówiąca czy minimapa ma być aktualnie wyświetlana,

- `sf::Music _music` - muzyka odtwarzana w grze.

- **Settings** - struktura, w której zgromadzone zostały ustawienia aplikacji, tj. wykorzystywane kolory, FOV, mapa, położenia startu i końca.
  - `static void getStartEnd(sf::Vector2<double> &start, sf::Vector2<double> &end)` - metoda ustawiająca zmienne początku oraz końca labiryntu,
  - `static bool checkSigns()` - metoda sprawdzająca, czy w mapie nie ma niedozwolonych znaków oraz czy znajduje się w niej koniec i początek,
  - `static bool validateMaze()` - metoda wykonujące walidację mapy,
  - `static void neighborDown(int &x, int &y, direction &dir)` - metoda szukająca następnego segmentu przy poruszaniu się w dół,
  - `static void neighborUp(int &x, int &y, direction &dir)` - metoda szukająca następnego segmentu przy poruszaniu się w górę,
  - `static void neighborLeft(int &x, int &y, direction &dir)` - metoda szukająca następnego segmentu przy poruszaniu się w lewo,
  - `static void neighborRight(int &x, int &y, direction &dir)` - metoda szukająca następnego segmentu przy poruszaniu się w prawo,
  - `static bool mapCreated` - zmienna mówiąca, czy mapa została stworzona,
  - `static bool music` - zmienna mówiąca, czy muzyka ma być odtwarzana,
  - `static sf::Vector2<double> start` - położenie punktu startowego,
  - `static sf::Vector2<double> end` - położnie punktu końcowego,
  - `static double FOV` - pole widzenia,
  - `static std::vector<std::vector<char>> worldMap` - wektor wektorów przechowujący mapę,
  - `static sf::Color wallX[2]` - kolor ściany oznaczonej "X",
  - `static sf::Color wallY[2]` - kolor ściany oznaczonej "Y",
  - `static sf::Color wallZ[2]` - kolor ściany oznaczonej "Z",
  - `static sf::Color ceil` - kolor sufitu,
  - `static sf::Color floor` - kolor podłogi,
  - `static sf::Color exit` - kolor wyjścia,
  - `enum direction { Up, Down, Left, Right }` - typ wyliczeniowy określający kierunek sprawdzania labiryntu.



Rysunek 6: Schemat UML klas występujących w programie

Starano się podejść do kodowania obiektowo, stworzono również wiele funkcji, żeby ograniczyć jak najbardziej obszerność kodu. Metody oraz zmienne nazywane były w sposób przedstawiający ich zadania.

Plansza przechowywana jest w wektorze `std::vector<std::vector<BoardBox>> _board`, co umożliwia wygodne poruszanie się po niej. Rysowanie siatki w kreatorze zostało zaimplementowane w intuicyjny sposób. Każde pole na planszy reprezentowane jest przez strukturę `BoardBox`, w której umieszczana jest tekstura (`wxImage`) oraz pozycja danego prostokąta, co ułatwia jego rysowanie. Rysowanie odbywa się przy każdorazowej zmianie rozmiaru okna oraz przy interakcji z użytkownikiem. Wygląd pola (tekstura) zmieniany jest po kliknięciu przez użytkownika na obszar należący do danego prostokąta. Wykorzystanie `wxImage` do przechowywania obrazów sprawia, że w bardzo łatwy sposób można dopasować wygląd planszy do własnych upodobań (wystarczy podmienić odpowiednie tekstury).

8 Testowanie

Etap testowania przebiegał równolegle z etapami implementacji poszczególnych modułów. Pozwoliło to na bieżące wykrywanie i niwelowanie błędów. Na początku przetestowane zostało rysowanie i poruszanie się po labiryncie, sprawdzając czy uzyskano zamierzone efekty. Następnie, głównie manualnie poprzez wczytywanie różnych wariancji, testowano walidację poprawności wczytywanych map. Testując program brano pod uwagę jak najbardziej skrajne przypadki. Przy projektowaniu wzięliśmy pod uwagę dodanie "message boxów", które pozwoliły na podpowiadanie użytkownikowi, co należy zrobić w przypadku błędu. Ciągłe testowanie pozwoliło między innymi na zwiększenie prędkości działania programu, podjęcie się zaimplementowania walidacji mapy, aby gracz sam sobie nie wyrządził krzywdy oraz wykrycia wielu niedoróbek, które były na bieżaco poprawiane.

9 Wdrożenie, raport, wnioski

Program spełnia wszystkie początkowe założenia. Spełniono zatem wszystkie wymagania podstawowe - użytkownik może wczytać własną, wcześniej przygotowaną mapę. Jeżeli gracz podejdzie do ściany oznaczonej jako wyjście, kończy grę i wyświetlony zostaje czas, w którym udało się mu przejść labirynt. Wymagania rozszerzone również zostały spełnione - ściany mogą być w różnych kolorach oraz gracz ma do wyboru zakres pola widzenia. Widok rzutu z góry zamieniliśmy na minimapę, dzięki której gracz może odnaleźć się w labiryncie (co może być bardzo trudne bez takiego ułatwienia). Jest ona jednak opcjonalna - jeżeli gracz nie chce z niej korzystać, to nie musi tego robić. Dodatkowo, dla wygody użytkownika, stworzony został kreator umożliwiający efektywne tworzenie oraz edycję map labiryntu, a autorski algorytm walidacji zabezpiecza program przez wczytaniem nieprawidłowych danych wejściowych. W programie wykorzystano twarz Jokera ze strony FreeVector.com oraz muzykę ze strony Bensound.com na licencjach pozwalających na ich użycie.