



POLITECHNIKA WARSZAWSKA
WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH



PRACA DYPLOMOWA INŻYNIERSKA
INFORMATYKA

Symulacja ruchu robota w labiryncie

Autorzy:

Wojciech Kowalik
Konrad Miśkiewicz
Mateusz Pielat

Promotor: dr Paweł Rzążewski

Warszawa, 15.01.2016

.....
podpis promotora

.....
podpis autora

Spis treści

Streszczenie	1
1 Analiza wymagań	2
1.1 Słownik pojęć	2
1.2 Wymagania funkcjonalne edytora map i pojazdów	3
1.3 Wymagania funkcjonalne trasyera plików graficznych	4
1.4 Wymagania funkcjonalne modułu symulacji	4
1.5 Wymagania niefunkcjonalne	5
2 Technologie	7
2.1 XAML	8
2.2 MVVM Light	9
2.3 MonoGame	9
2.4 SVG	9
2.5 D3DPotrace	10
3 Dokumentacja techniczna	11
3.1 Architektura systemu	11
3.1.1 MVVM	11
3.1.2 Service Locator	11
3.1.3 Mediator	12
3.2 Połączenie MonoGame i WPF	12
3.3 Modele	15
3.4 Reprezentacja mapy i pojazdu	17
3.5 Trasowanie plików graficznych	18
4 Algorytmy	19
4.1 Algorytmy graficzne	19
4.1.1 Rysowanie linii bez antyaliasingu - algorytm Bresenhama	19
4.1.2 Rysowanie linii z antyaliasingiem - algorytm Xiaolin Wu	19
4.1.3 Algorytm mieszania kolorów	19
4.2 Algorytmy geometryczne	19

4.2.1	Suma Minkowskiego	19
4.3	Algorytm wyznaczania ścieżki	26
4.3.1	Opis algorytmu	26
4.3.2	Pseudokod	40
4.3.3	Złożoność algorytmu	42
5	Analiza porównawcza dla algorytmu wyznaczania ścieżki	44
6	Instrukcja użytkownika	45
6.1	Edytor map i pojazdów	46
6.1.1	Edytor map	46
6.1.2	Edytor pojazdów	47
6.2	Traser plików graficznych	49
6.3	Moduł symulacji	51
6.4	Ustawienia aplikacji	54
7	Przebieg pracy	55
7.1	Model wytwórczy	55
7.2	Harmonogram i podział prac	55
7.2.1	Etap 1	55
7.2.2	Etap 2	56
7.2.3	Etap 3	56
7.2.4	Etap 4	57
	Bibliografia	58

Streszczenie

Niniejsza praca dyplomowa opisuje proces tworzenia aplikacji umożliwiającej przeprowadzenie symulacji ruchu robota w labiryncie. Aplikacja została wyposażona w trzy podstawowe moduły: ręczny edytor map i pojazdów, traser plików graficznych oraz moduł symulacji. Program działa poprawnie pod kontrolą systemów operacyjnych z rodziny Windows z zainstalowaną platformą .NET Framework 4.6+.

W pierwszym rozdziale szczegółowo opisano wszystkie wymagania funkcjonalne dla każdego z modułów. Najważniejszą funkcjonalnością dla ręcznych edytorów jest możliwość stworzenia pojazdu i mapy oraz zapisania ich do pliku SVG. Traser plików graficznych umożliwia wczytanie pliku graficznego w jednym z dozwolonych formatów, a następnie stworzenie z niego pojazdu lub mapy. Moduł symulacji pozwala na wczytanie mapy oraz pojazdu, wyznaczenie punktu startowego i końcowego oraz obliczenie ścieżki pomiędzy tymi punktami i wizualizację ruchu pojazdu (jeśli ścieżka istnieje). Dodatkowo możliwa jest zmiana rozmiaru pojazdu oraz jego początkowego obrotu.

Kolejne trzy rozdziały zostały poświęcone użytym technologiom, architekturze systemu oraz zastosowanym w aplikacji algorytmom. Opisano między innymi w jaki sposób stworzyć i osadzić kontrolkę opartą na silniku graficznym MonoGame w aplikacji zbudowanej w technologii WPF (Windows Presentation Foundation), a także jakie ograniczenia wynikają z takiego połączenia. Szczegółowo opisano również sposób w jaki obliczana jest ścieżka od punktu startowego do punktu końcowego, oraz jakie parametry wpływają na wynik obliczeń.

W rozdziale piątym dokonana została analiza porównawcza obejmująca zarówno wydajność, jak i jakość tworzonej ścieżki ze względu na dobór parametrów przez użytkownika oraz złożoność mapy testowej i pojazdu.

Rozdział szósty zawiera szczegółową instrukcję użytkownika. Krok po kroku opisane zostały czynności prowadzące do przygotowania symulacji.

Podsumowaniem pracy jest krótki raport z przeprowadzonych testów oraz opis modelu wytwórczego i przebiegu pisania pracy.

Rozdział 1

Analiza wymagań

Celem niniejszej pracy dyplomowej było zbudowanie aplikacji umożliwiającej przygotowanie i przeprowadzenie symulacji ruchu pojazdu w labiryncie. Aplikacja została podzielona na trzy podstawowe moduły. Poniżej szczegółowo opisane zostały wymagania funkcjonalne dla każdego z modułów oraz wymagania niefunkcjonalne dotyczące całej aplikacji.

1.1 Słownik pojęć

Algorytm - algorytm wyszukiwania ścieżki prowadzącej od *punktu startowego* do *punktu końcowego* dla danej *mapy* i *pojazdu*.

Mapa - zbiór wielokątów reprezentujących *przeszkody*, których powinien unikać *pojazd*.

Oś obrotu pojazdu - punkt znajdujący się wewnątrz wielokąta reprezentującego *pojazd*, do którego sprowadzany będzie *pojazd* podczas działania *algorytmu*.

Platforma .NET Framework - platforma programistyczna opracowana przez Microsoft, obejmująca środowisko uruchomieniowe oraz biblioteki klas dostarczające standardowej funkcjonalności dla aplikacji.

Plik graficzny - zdjęcie/bitmapa w jednym z dozwolonych formatów.

Pojazd - dowolny wielokąt z wyznaczoną *osią obrotu* oraz wskazanym kierunkiem ruchu.

Przeszkoda - dowolny wielokąt wchodzący w skład wielokątów tworzących *mapę*.

Punkt końcowy - wyznaczany przez *użytkownika* punkt na *mapie*, do którego będzie wyznaczana ścieżka, którą powinien przebyć *pojazd*.

Punkt startowy - wyznaczany przez *użytkownika* punkt na *mapie*, z którego wyruszy *pojazd*.

System - wszystkie moduły aplikacji połączone w całość.

Trasa - wyznaczona przez *algorytm* ścieżka pomiędzy *punktem początkowym* i *punktem końcowym*, którą powinien przebyć *pojazd*.

Traser plików graficznych - jeden z modułów systemu umożliwiający *trasowanie plików graficznych*, a następnie przekształcenie ich w *pojazd* lub *mapę*.

Trasowanie - proces polegający na przekształceniu grafiki w postaci rastrowej do postaci wektorowej.

Użytkownik - osoba korzystająca z aplikacji.

Wizualizacja - animacja poruszającego się *pojazdu* po *trasie* wyznaczonej przez *algorytm*.

Wzorzec MVVM - wzorzec architektoniczny wykorzystywany do rozdzielania warstwy prezentacji, logiki oraz warstwy danych

XML - uniwersalny język znaczników przeznaczony do reprezentowania różnych danych w strukturalizowany sposób.

1.2 Wymagania funkcjonalne edytora map i pojazdów

Głównym zadaniem edytora map i pojazdów jest umożliwienie użytkownikowi stworzenia i zapisania do pliku odpowiednio mapy lub pojazdu. Mapa może składać się z dowolnej liczby wielokątów, natomiast pojazd jest reprezentowany jako jeden wielokąt z wyznaczoną osią obrotu i kierunkiem jazdy. W celu stworzenia pojazdu/mapy moduł umożliwia:

- rysowanie wielokątów reprezentujących pojazdy lub przeszkody omijane przez pojazd.
- zapis narysowanych pojazdów /map do plików SVG
- wczytywanie uprzednio zapisanych pojazdów /map
- wyznaczenie osi obrotu oraz kierunku jazdy pojazdu
- cofnięcie ostatnio wykonanej akcji (np. usunięcie ostatniej dodanej krawędzi)
- wyczyszczenia obszaru edytora i rozpoczęcia tworzenia pojazdu/mapy od początku

Dodatkowo moduł został wyposażony w mechanizm zapobiegający przecinaniu się krawędzi tego samego wielokąta. Jeśli użytkownik chciałby narysować krawędź przecinającą inną krawędź obecnie rysowanego wielokąta, nie będzie to możliwe. Dopuszczone są jedynie sytuacje, w których przecinają się krawędzie różnych wielokątów.

1.3 Wymagania funkcjonalne trasera plików graficznych

Głównym zadaniem trasera plików graficznych jest przekształcenie pliku graficznego w postaci rastrowej do postaci wektorowej, a następnie umożliwienie stworzenia pojazdu lub mapy z wielokątów powstałych w wyniku trasowania. W celu realizacji tego zadania moduł umożliwia:

- wczytanie pliku graficznego w jednym z dozwolonych formatów
- trasowanie wczytanego pliku graficznego
- wybór jednego wielokąta w celu utworzenia pojazdu
- wybór wielu wielokątów w celu utworzenia mapy
- dobór parametrów trasowania [patrz 6.2]

Moduł umożliwia stworzenie pojazdu w przypadku, gdy wybrany został tylko jeden wielokąt. W przeciwnym wypadku możliwe jest utworzenie mapy. Dopuszczona jest sytuacja, w której nie został wybrany żaden wielokąt (mapa może być pusta). Dodatkowo po zatwierdzeniu wielokąta jako pojazdu konieczne jest wyznaczenie jego osi obrotu i kierunku ruchu. Po wybraniu kilku wielokątów i utworzeniu z nich mapy, użytkownik ma możliwość ręcznego dodania nowych wielokątów.

Dozwolone formaty dla plików graficznych:

- PNG
- BMP
- JPG, JPEG
- GIF - w przypadku tego formatu poprawne działanie nie jest gwarantowane dla plików będących animacją

1.4 Wymagania funkcjonalne modułu symulacji

Głównym zadaniem modułu symulacji jest obliczenie ścieżki oraz wizualizacja ruchu pojazdu od punktu początkowego do punktu końcowego. W celu realizacji tego zadania moduł umożliwia:

- wczytanie uprzednio stworzonej i zapisanej mapy
- wczytanie uprzednio stworzonego i zapisanego pojazdu

- wybór punktu początkowego
- dopasowanie rozmiaru oraz obrotu początkowego pojazdu
- wybór punktu końcowego
- dobór parametrów dla algorytmu [patrz 4.3]
- obliczenie ścieżki za pomocą algorytmu [patrz 4.3]
- wczytanie uprzednio zapisanej symulacji
- zapisanie gotowej symulacji
- rozpoczęcie tworzenia symulacji od początku

Jako wynik obliczeń algorytmu, moduł zwraca listę rozkazów [patrz x.x] na podstawie której tworzona jest animacja ruchu pojazdu po mapie. Pojedynczy rozkaz zawiera następujące informacje:

- punkt docelowy, do którego powinien przemieścić się pojazd,
- bezwzględną wartość kąta o jaki powinien być obrócony pojazd, przed rozpoczęciem ruchu w kierunku punktu docelowego.

Ponadto wartość kąta może być ujemna lub dodatnia, co umożliwi wykonywanie obrotów zgodnie z ruchem wskazówek zegara lub przeciwnie do ruchu wskazówek zegara. Zostało to dokładniej opisane w dalszej części dokumentu. [patrz x.x]

Po przygotowaniu symulacji i obliczeniu ścieżki, użytkownik ma możliwość rozpoczęcia odtwarzania, wstrzymywania lub całkowitego zatrzymania animacji. Dodatkowo ma on do dyspozycji suwak, dzięki któremu może ręcznie przeglądać animację lub rozpocząć jej odtwarzanie od wybranego miejsca.

1.5 Wymagania niefunkcjonalne

Poza wymienionymi powyżej wymaganiami funkcjonalnymi dla poszczególnych modułów, cała aplikacja spełnia wymienione poniżej wymagania niefunkcjonalne.

- Aplikacja poprawnie działa na systemach operacyjnych z rodziny Windows z zainstalowaną platformą .NET Framework 4.6+.
- System informuje użytkownika, gdy wyznaczenie ścieżki od punktu startowego do punktu końcowego nie jest możliwe.
- Rysowanie wielokątów odbywa się poprzez dodawanie kolejnych punktów w obszarze edytora.

- Podczas rysowania wielokątów ich krawędzie nie mogą się przecinać.
- System umożliwia łatwe przetłumaczenie interfejsu na różne języki - posiada wbudowany język polski i angielski

Rozdział 2

Technologie

W tym rozdziale przedstawione zostały technologie, które wybrano w celu realizacji wymagań opisanych w rozdziale 1.

Główną technologią wykorzystaną podczas pisania pracy dyplomowej jest technologia Windows Presentation Foundation (WPF) wchodząca w skład platformy .NET Framework. API w WPF opiera się na języku XML, dokładniej na jego implementacji o nazwie XAML [patrz 2.1]. O wyborze tej technologii zdecydowała prostota tworzenia interfejsu użytkownika oraz fakt, iż WPF świetnie sprawdza się podczas implementacji wzorca MVVM [patrz 3.x]. Aby interfejs aplikacji był jak najbardziej przyjazny dla użytkownika zastosowano bibliotekę ModernUI, która odświeża wygląd podstawowych kontrolki oraz sprawia, iż aplikacja swoim wyglądem przypomina aplikacje przeznaczone dla systemu Windows 8 lub Windows 10. Ponadto przy implementacji wzorca MVVM wykorzystano bibliotekę MVVM Light [patrz 2.2].

Podczas implementacji modułu ręcznych edytorów map i pojazdów oraz modułu symulacji zastosowano silnik graficzny MonoGame [patrz 2.3]. Przy implementacji modułu trasowania plików graficznych wykorzystano zaś bibliotekę D3DPotrace [patrz 2.5]. Dodatkowo stworzone przez użytkownika pojazdy, mapy oraz symulacje mogą być zapisywane do plików SVG [patrz 2.4], które następnie można podejrzeć bezpośrednio za pomocą przeglądarki internetowej, wspierającej format SVG.

Ze względu na to, iż algorytm wyszukiwania ścieżki [patrz 4.3] na mapie opiera się na tworzeniu i przeszukiwaniu grafów, konieczne było skorzystanie z biblioteki, która udostępnia wymienione operacje. Wybrana została biblioteka Graph autorstwa dr. Jana Bródki.

2.1 XAML

XAML jest językiem używanym do opisu interfejsu użytkownika opartym na języku XML. Jest on elementem platformy .NET Framework począwszy od wersji 3.0. Język XAML umożliwia rozdzielenie definicji interfejsu użytkownika od kodu logiki, który często znajduje się w osobnych plikach. Takie podejście nazywamy podejściem code-behind. Poza rozdzieleniem interfejsu użytkownika od kodu logiki język XAML umożliwia także prostą implementację wzorca MVVM [patrz 3.x].

Przykład 2.1.1. Przypisanie metody do zdarzenia kliknięcia przycisku w podejściu code-behind

```
<Button Background="Blue" Content="Hello World" Click="Button_Click"/>
```

W przypadku pracy z MVVM zawartość kontrolki czy wywoływane zdarzenia możemy wiązać z właściwościami lub metodami zaimplementowanymi w odrębnej klasie, nazywanej klasą view-modelu. Klasa ta zawiera komendy, do których wiązane są akcje z interfejsu użytkownika oraz publiczne właściwości do których możemy podpiąć różne elementy interfejsu [patrz 3.x]. Aby było to możliwe, jako źródło danych w pliku XAML należy podać odpowiednią klasę view-modelu.

Przykład 2.1.2. Zmiana źródła danych

```
<DataContext="{Binding User, Source={StaticResource Locator}}">
```

Dodanie powyższej linii oznacza, że od tej pory wszystkie dane pochodzić będą z view-modelu o nazwie *User*. Nazwa ta została zdefiniowana w klasie *Locator*, która przechowuje nazwy wszystkich view-modeli.

Przykład 2.1.3. Definicja nazwy view-modelu w klasie *Locator*

```
public UserViewModel User
{
    get
    {
        return ServiceLocator.Current.GetInstance<UserViewModel>();
    }
}
```

Dzięki zmianie źródła danych, odwoływanie się do właściwości czy metod z klasy *UserViewModel* staje się bardzo proste i może ono wyglądać jak na poniższym przykładzie.

Przykład 2.1.4. Odwoływanie się do właściwości i metod z view-modelu

```
<TextBox Text="{Binding Name}"/>
<Button Command="{Binding EditProfileCommand}"/>
```

2.2 MVVM Light

MVVM Light to biblioteka, której głównym zadaniem jest zapewnienie wsparcia dla wzorca MVVM. Dzięki zastosowaniu tej biblioteki uzyskujemy dostęp do kilku istotnych z punktu widzenia naszej pracy elementów, takich jak:

- *RelayCommand* - lokalna implementacja interfejsu *ICommand*. Możemy w ten sposób tworzyć różne akcje, które później zostaną podpięte do widoku [patrz 3.x]
- *ViewModelBase* - bazowa klasa dla wszystkich view-modeli [patrz 3.x]

Dostęp do właściwości i metod z view-modeli można uzyskać w identyczny sposób jak w przykładzie 2.1.4.

2.3 MonoGame

MonoGame jest silnikiem graficznym będącym kontynuacją projektu XNA stworzonego przez firmę Microsoft. Obecnie kod źródłowy MonoGame jest otwarty, dzięki czemu technologia ta wciąż zyskuje nowych zwolenników. MonoGame pozwala na tworzenie projektów działających na większości popularnych systemów operacyjnych, jak na przykład Windows, Android czy iOS.

Podczas pisania niniejszej pracy dyplomowej, silnik MonoGame został wykorzystany przy tworzeniu modułu ręcznych edytorów map i pojazdów oraz modułu symulacji. W tym celu została przygotowana specjalna kontrolka hostująca [patrz x.x], pozwalająca na osadzenie MonoGame wewnątrz WPF. Okazało się jednak, że poprzez takie działanie, część z dostępnych w MonoGame funkcjonalności nie mogła zostać wykorzystana w projekcie. Głównym problemem natury estetycznej, okazał się brak możliwości włączenia antyaliasingu dla rysowanych obiektów. Z tego powodu konieczna była ręczna implementacja odpowiedniego algorytmu odpowiedzialnego za rysowanie linii z antyaliasingiem [patrz 5.1].

2.4 SVG

SVG jest uniwersalnym formatem dwuwymiarowej grafiki wektorowej. Pozwala na tworzenie grafiki statycznej lub animowanej. Format SVG powstał z myślą o zastosowaniu głównie na stronach WWW, jednak obecnie jego zastosowanie jest znacznie szersze. Do popularyzacji formatu SVG przyczyniła się jego niezależność od platformy systemowej oraz prostota.

SVG należy do rodziny języka XML. Poniżej zaprezentowano przykładowy kod pliku SVG oraz efekt jego działania.

Przykład 2.4.1. Przykładowy plik SVG

```
<svg height="210" width="500">  
  <polygon points="200,10 250,190 160,210"  
    style="fill:lime; stroke:purple; stroke-width:1" />  
</svg>
```



Obrazek 2.4.1. Przykładowy plik SVG

W opisywanej pracy format SVG został wykorzystany do zapisywania stworzonych map, pojazdów i symulacji [patrz x.x]. Dzięki takiemu rozwiązaniu, użytkownik ma możliwość ich podglądu bez konieczności uruchamiania aplikacji, bezpośrednio w przeglądarce internetowej. Mapy i pojazdy zapisywane są jako statyczne pliki SVG, natomiast symulacje są zapisywane jako prosta animacja.

2.5 D3DPotrace

D3DPotrace to biblioteka będąca portem biblioteki Potrace przeznaczonym dla platformy .NET Framework. Jest to jedna z dwóch (obok AutoTrace) popularnych bibliotek do trasowania obrazków rastrowych o otwartym kodzie.

Biblioteka D3DPotrace została wykorzystana przy tworzeniu modułu trasowania plików graficznych [patrz 3.x]. Jest ona opakowana specjalnie do tego przygotowaną klasą *Bitmap-Tracer* [patrz x.x], która upublicznia metodę służącą do trasowania pliku graficznego. Metoda ta przyjmuje kilka parametrów, pozwalających na dostosowanie trasowania do konkretnego pliku graficznego [patrz x.x].

Rozdział 3

Dokumentacja techniczna

W tym rozdziale opisano architekturę systemu oraz modele wykorzystane podczas jego implementacji. Przedstawiono również sposób na osadzenie silnika MonoGame wewnątrz interfejsu WPF.

3.1 Architektura systemu

3.1.1 MVVM

Architektura systemu została oparta o wzorzec *Model-View-ViewModel* (MVVM) zaprojektowany dla platformy WPF. Polega on na rozdzieleniu warstwy prezentacji (view), logiki biznesowej (model) oraz logiki prezentacji (view-model) umożliwiającej łatwe przetwarzanie modelu przez widok. Istotnym elementem technologii pozwalającej wprowadzić ten wzorzec jest tzw. *Binder*, który opisuje jak dane zawarte w danym momencie w view-modelu będą wyświetlone w widoku, lub też jak wprowadzanie danych przez użytkownika w interfejsie graficznym wpływa na dane w view-modelu.

Kontrolki MonoGame jako mechanizm spoza technologii WPF spowodowały jednak, że konieczne były pewne kompromisy w implementacji wzorca. W przeciwieństwie do reszty logiki prezentacji napisanej w kodzie XAML, renderowanie zawartości tych kontrolek odbywa się z pominięciem Bindera w tzw. *code-behind* (czyli w języku nie-znacznikowym jakim jest C#). Mimo wszystko zachowane zostały najważniejsze postulaty wzorca i logika poszczególnych warstw pozostała maksymalnie odseparowana, a warstwa biznesowa nigdy nie odwołuje się do żadnych mechanizmów związanych z prezentacją danych.

3.1.2 Service Locator

W aplikacji wykorzystano również wzorzec Service Locator będący szczególną wersją paradygmatu *Odwrócenie Sterowania* (Inversion of Control, IoC). Polega on na stworzeniu obiektu

zarządzającego i udostępniającego zarejestrowane wcześniej usługi, co pozwala na rozdzielenie klas od ich zależności.

Zastosowanie wzorca Service Locator (zamiast np. klasycznego *Dependency Injection*) podyktowane było charakterystyką aplikacji, w której w dużym stopniu mamy do czynienia z pojedynczymi obiektami klas istniejącymi przez cały cykl działania aplikacji (np. view-modele).

3.1.3 Mediator

Kolejnym użytym wzorcem podczas pisania pracy dyplomowej jest wzorzec *Mediator*. Ułatwia on komunikację poprzez wprowadzenie pośrednika dystrybuującego wiadomości wysyłane i odbierane przez obiekty różnych klas. Komunikacja jest w ten sposób prowadzona niebezpośrednio, co zwiększa poziom separacji zależności w programie i pozwala na obsługę interakcji między-obiektowej w sposób niezależny.

Mediator to popularny sposób na omijanie pewnych ograniczeń MVVM, jednak użycie tego wzorca w systemie zostało sprowadzone do minimum.

3.2 Połączenie MonoGame i WPF

Technologia WPF domyślnie nie umożliwia osadzenia zewnętrznego silnika graficznego, więc aby móc skorzystać z MonoGame w tym środowisku, konieczne było napisanie odpowiedniej kontrolki. Została ona zaimplementowana jako osobna biblioteka DLL, aby możliwe było jej ewentualne wykorzystanie w innych projektach. Schemat kontrolki przedstawiono na diagramie poniżej.

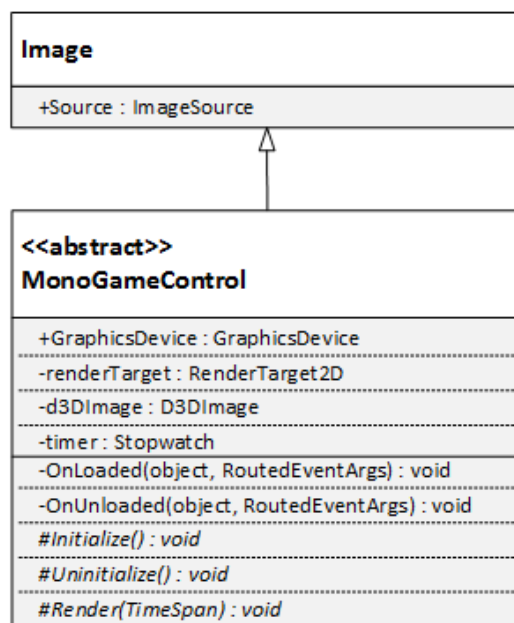


Diagram x.x. Klasa MonoGameControl

Kontrolka umożliwiająca osadzenie MonoGame w WPF została zaimplementowana jako klasa abstrakcyjna *MonoGameControl*, dziedzicząca po klasie *Image*, która wchodzi w skład technologii WPF. Aby możliwe było jej użycie, należy dodatkowo utworzyć własną klasę dziedziczącą po *MonoGameControl*, która implementuje następujące metody:

- **Initialize()** - metoda wywoływana przy tworzeniu kontrolki,
- **Uninitialize()** - metoda wywoływana przy destrukcji kontrolki,
- **Render(TimeSpan time)** - wywoływana cyklicznie metoda, odpowiadająca za renderowanie obrazu wyświetlanego w kontrolce.

Przykład 3.2.1. Implementacja własnej kontrolki MonoGame

```
public class MyOwnControl : MonoGameControl
{
    private SpriteBatch spriteBatch;

    protected override void Initialize()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void Unitialize()
    {
        spriteBatch.Dispose();
    }

    protected override void Render(TimeSpan time)
    {
        GraphicsDevice.Clear(Color.LightSkyBlue);
        GraphicsDevice.RasterizerState = RasterizerState.CullNone;

        spriteBatch.BeginDraw();

        // DRAW

        spriteBatch.End();
    }
}
```

Po zaimplementowaniu własnej klasy dziedziczącej po *MonoGameControl*, kontrolka jest gotowa do użycia. Poniżej przedstawiono fragment kodu w języku XAML, który umożliwia osadzenie kontrolki w interfejsie aplikacji.

Przykład 3.2.2. Osadzenie kontrolki w pliku XAML

```
<local:MyOwnControl SnapsToDevicePixels="True"/>
```

Zasada działania napisanej kontrolki jest dość prosta - łączy ona w sobie pola będące częścią technologii WPF (typy *D3DImage*, *Stopwatch*) z polami, które wchodzi w skład silnika *MonoGame* (typy *GraphicsDevice*, *RenderTarget2D*). W momencie jej tworzenia wywoływana jest metoda *OnLoaded*. Wewnątrz tej metody następuje wykonanie następujących operacji:

- inicjalizacja pola *GraphicsDevice*
- inicjalizacja pola *d3DImage*
- inicjalizacja pola *renderTarget* - jako jeden z parametrów przekazywane jest pole *GraphicsDevice*
- inicjalizacja pola *timer* i rozpoczęcie odliczania
- ustawienie tylnego bufora dla obiektu *d3DImage* na obiekt *renderTarget*
- wykonanie zaimplementowanej przez użytkownika metody *Initialize()*
- przypisanie metody *OnRendering()* do zdarzenia *Render* dla obiektu *CompositionTarget*. Obiekt ten jest dostępny wewnątrz klasy *MonoGameControl*, ponieważ dziedziczy ona po klasie *Image*. Metoda *OnRendering()* zawiera ustawienie pola *RenderTarget* dla obiektu *GraphicsDevice* na obiekt *renderTarget* oraz wywołanie zaimplementowanej przez użytkownika metody *Render(TimeSpan)*, do której jako *TimeSpan* przekazywana jest wartość z pola *timer.Elapsed*.
- ustawienie jako źródła (*Source*) obrazu obiektu *d3DImage*.

Po wykonaniu powyższych operacji, kontrolka rozpoczyna wyświetlanie obrazu wygenerowanego przez silnik graficzny *MonoGame*. Przy kończeniu pracy kontrolki wywoływana jest Metoda *OnUnloaded*, w której następuje usunięcie obiektów utworzonych w metodzie *OnLoaded* w kolejności odwrotnej do ich utworzenia. Wykonywana jest również metoda *Uninitialize()*.

3.3 Modele

Podczas pisania aplikacji zostały utworzone następujące modele danych i interfejsy:

- **interfejs *IAlgorithm*** - interfejs, który musi implementować klasa realizująca funkcjonalności algorytmu wyznaczania ścieżki. Interfejs ten zawiera trzy nagłówki metod:
 - *GetPath(...)* - nagłówek metody wyznaczającej ścieżkę od punktu startowego do punktu końcowego, dla danej mapy i pojazdu. Metoda zwraca listę obiektów typu *Order*.
 - *GetOptions()* - nagłówek metody zwracającej parametry działania algorytmu wyznaczania ścieżki. Metoda zwraca listę obiektów typu *Option*.
 - *SetOptions(...)* - nagłówek metody umożliwiającej ustawienie parametrów działania algorytmu wyznaczania ścieżki. Jako parametr metoda przyjmuje listę obiektów typu *Option*.
- ***SvgSerializable*** - klasa abstrakcyjna implementująca interfejs *IXmlSerializable*. Każdy obiekt, który może być zapisany do pliku SVG dziedziczy po tej klasie. Zawiera ona następujące metody:
 - *Serialize<T>(...)* - metoda uogólniona umożliwiająca serializację obiektu typu *T*.
 - *Deserialize<T>(...)* - metoda uogólniona umożliwiająca deserializację pliku na obiekt typu *T*.
 - *CanDeserialize<T>(...)* - metoda uogólniona umożliwiająca sprawdzenie poprawności pliku, podanego jako plik przeznaczony do deserializacji na obiekt typu *T*.
- ***Frame*** - klasa reprezentująca pojedynczą klatkę animacji ruchu pojazdu po wyznaczonej trasie. Klasa *Frame* zawiera następujące pola:
 - *Rotation* - pole typu *double*, przechowuje informację o bezwzględnym obrocie pojazdu.
 - *Position* - pole typu *Point*, przechowuje informacje o aktualnej pozycji pojazdu.
- ***Map*** - klasa reprezentująca mapę. Klasa *Map* dziedziczy po klasie *SvgSerializable* oraz implementuje interfejs *IEquatable*. Jedynym jej polem jest lista wielokątów wchodzących w skład mapy - *Obstacles*.
- ***Option*** - klasa reprezentująca pojedynczy parametr algorytmu. Dzięki przekazywaniu do algorytmu parametrów za pomocą listy obiektów typu *Option* możliwe jest łatwe modyfikowanie liczby przekazywanych parametrów, oraz przekazywanie tylko tych parametrów, które faktycznie mają zostać zmienione (pozostałe parametry będą mieć wartości domyślne). Klasa *Option* zawiera następujące pola:

- *Type* - pole określające typ przekazywanego parametru. Przyjmuje jedną z trzech wartości: *Integer*, *Double*, *Boolean*.
- *Value* - pole określające wartość przekazywanego parametru.
- *MinValue* - pole określające minimalną wartość przekazywanego parametru. Pole to może przyjmować wartość *NULL*.
- *MaxValue* - pole określające maksymalną wartość przekazywanego parametru. Pole to może przyjmować wartość *NULL*.
- **Order** - klasa reprezentująca pojedynczy rozkaz wygenerowanej symulacji. Klasa *Order* dziedziczy po klasie *SvgSerializable*. Zawiera ona następujące pola:
 - *Rotation* - pole typu *double*, określa do jakiego kąta powinien zostać obrócony pojazd przed przemieszczeniem do punktu *Destination*. Dodatkowo kąt ujemny oznacza, że obrót powinien zostać wykonany zgodnie z ruchem wskazówek zegara, natomiast kąt nieujemny determinuje obrót przeciwny do ruchu wskazówek zegara.
 - *Destination* - pole typu *Point*, określa do jakiego położenia powinien przemieścić się pojazd.
- **Polygon** - klasa reprezentująca wielokąt. Klasa *Polygon* dziedziczy po klasie *SvgSerializable* oraz implementuje interfejs *IEquatable*. Jedynym jej polem jest lista punktów tworzących wielokąt - *Vertices*.
- **Simulation** - klasa reprezentująca symulację. Klasa *Polygon* dziedziczy po klasie *SvgSerializable* oraz implementuje interfejs *IEquatable*. Zawiera ona następujące pola:
 - *Map* - obiekt typu *Map*, w którym przechowywana jest mapa.
 - *Vehicle* - obiekt typu *Vehicle*, w którym przechowywany jest pojazd.
 - *VehicleSize* - pole typu *double*, określa rozmiar pojazdu.
 - *InitialVehicleRotation* - pole typu *double*, określa początkowy obrót pojazdu.
 - *StartPoint* - obiekt typu *Point*, określa punkt startowy dla algorytmu wyznaczania ścieżki.
 - *EndPoint* - obiekt typu *Point*, określa punkt końcowy dla algorytmu wyznaczania ścieżki.
 - *Orders* - lista rozkazów symulacji wygenerowanych przez algorytm wyznaczania ścieżki. Zawiera obiekty typu *Order*.
- **Vehicle** - klasa reprezentująca pojazd.

Zależności pomiędzy opisanymi modelami zostały zaprezentowane na poniższym diagramie klas.

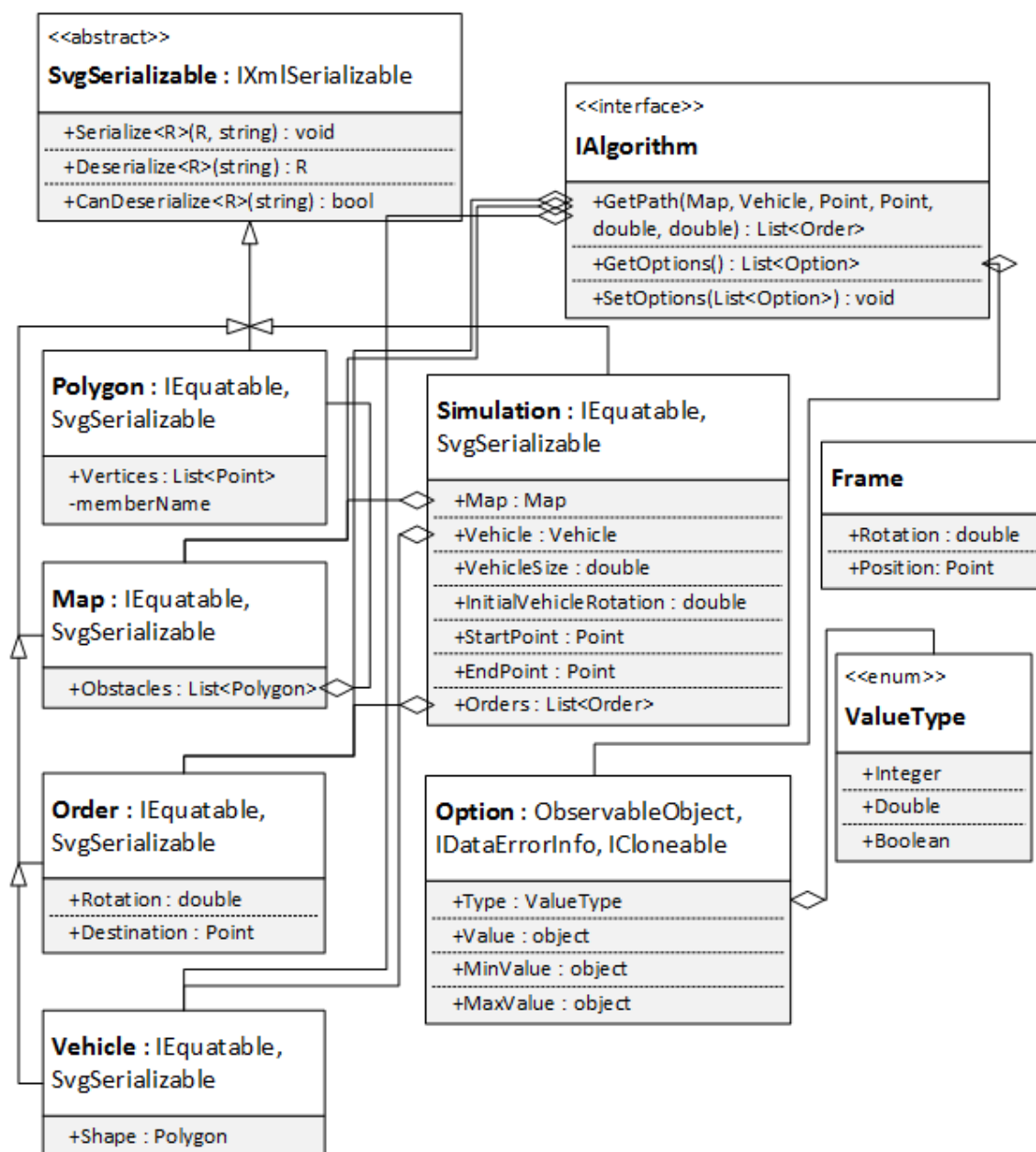


Diagram x.x. Diagram klas modeli

3.4 Reprezentacja mapy i pojazdu

TODO

3.5 Trasowanie plików graficznych

Do trasowania plików graficznych użyta została zewnętrzna biblioteka D3DPotrace [patrz 2.5]. Użycie biblioteki odbywa się za pośrednictwem zaprojektowanej w tym celu klasy *BitmapTracer*, widocznej na poniższym schemacie.

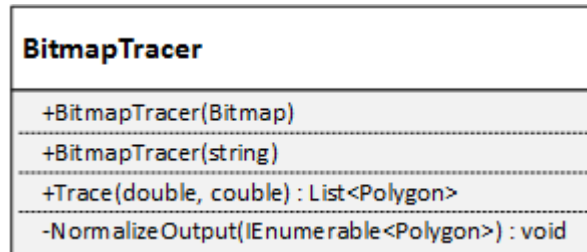


Diagram x.x. Klasa BitmapTracer

Klasa *BitmapTracer* posiada dwa konstruktory. Jeden z nich jako parametr przyjmuje bitmapę, drugi przyjmuje ścieżkę do pliku, z którego następnie tworzona jest nowa bitmapa. Najważniejszą metodą w tej klasie jest metoda *Trace* służąca do trasowania wczytanego pliku graficznego. Metoda ta przyjmuje dwa parametry typu *double* [patrz 6.2]. Wynikiem działania metody *Trace* jest lista obiektów typu *Polygon*.

Działanie samej biblioteki oparte jest na koncepcji hierarchizacji trasowanych obszarów. Przykładowo: umieszczona na białym tle czarna, wypełniona w środku plama jest obszarem traktowanym jako (zamknięty) wielokąt. Jeżeli na takiej czarnej plamie umieścimy całkowicie się w niej zawierającą plamę białą (tworząc czarny obwarzanek), to zostanie ona uznana za potomka plamy czarnej. Jeszcze mniejsza plama czarna zawarta w białej będzie potomkiem białej i pra-potomkiem pierwszej czarnej plamy itd. Ponieważ wielokąty zawarte całkowicie w innych wielokątach nie mają żadnego znaczenia (pojazd omijając przeszkodę i tak bierze pod uwagę tylko jej krawędź, a nie to co znajduje się w jej wnętrzu), dzięki takiej hierarchizacji danych możemy w łatwy sposób odrzucać wszystkie nieistotne dla obliczenia ścieżki kształty.

Rozdział 4

Algorytmy

4.1 Algorytmy graficzne

4.1.1 Rysowanie linii bez antyaliasingu - algorytm Bresenhama

4.1.2 Rysowanie linii z antyaliasingiem - algorytm Xiaolin Wu

4.1.3 Algorytm mieszania kolorów

4.2 Algorytmy geometryczne

W niniejszej pracy dyplomowej zostały zaimplementowane opisane poniżej algorytmy geometryczne.

4.2.1 Suma Minkowskiego

Definicja 4.2.1.1. Suma Minkowskiego

Sumą Minkowskiego nazywamy działanie określone na rodzinie wszystkich podzbiorów danej przestrzeni liniowej X wzorem:

$$A + B = \{a + b : a \in A, b \in B\},$$

gdzie A i B są podzbiorami przestrzeni liniowej X .

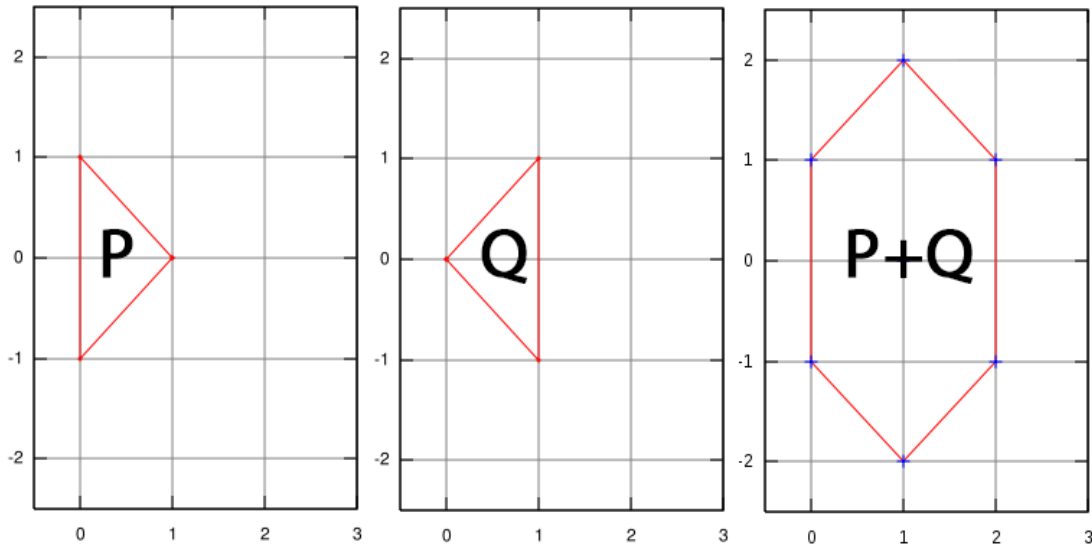
Powyższa definicja ma sens dla dowolnego zbioru X z określonym działaniem dodawania. W przypadku zbiorów dwuwymiarowych $P, Q \in \mathbb{R}^2$, będących podzbiorami płaszczyzny, sumę Minkowskiego tych zbiorów można interpretować jako sumę wektorową wektorów $p \in P$ i $q \in Q$, czyli gdy $p = (p_x, p_y)$ oraz $q = (q_x, q_y)$, to $p + q = (p_x + q_x, p_y + q_y)$.

Na poniższym obrazku zaprezentowano przykład sumy Minkowskiego dla dwóch podzbiorów płaszczyzny $P, Q \in \mathbb{R}^2$.

$$P = \{(1, 0), (0, 1), (0, -1)\}$$

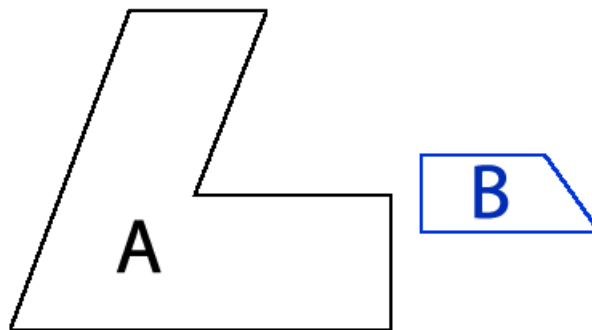
$$Q = \{(0, 0), (1, 1), (1, -1)\}$$

$$P + Q = \{(1, 0), (2, 1), (2, -1), (0, 1), (1, 2), (0, -1), (1, -2)\}.$$



Obrazek 4.3.1.1. Przykład sumy Minkowskiego dwóch wielokątów (źródło: wikipedia.org)

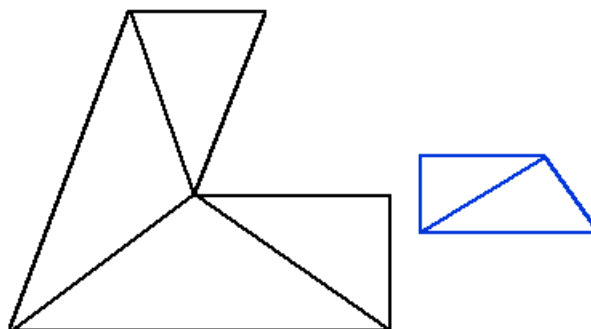
W ogólności wyznaczenie sumy Minkowskiego dwóch dowolnych wielokątów nie jest prostym zadaniem, dlatego działanie opisywanego algorytmu zostanie zaprezentowane na prostym przykładzie. Obliczana będzie suma Minkowskiego wielokątów A i B znajdujących się na poniższym obrazku.



Obrazek 4.3.1.1. Wielokąty A i B , dla których obliczana będzie suma Minkowskiego

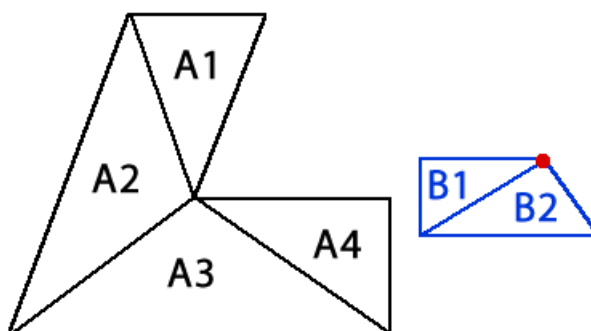
Pierwszym krokiem działania algorytmu obliczającego sumę Minkowskiego dwóch wielo-

kątown jest triangulacja obu wielokątów, czyli podział ich na trójkąty. Efekty triangulacji zostały przedstawione na poniższym obrazku.



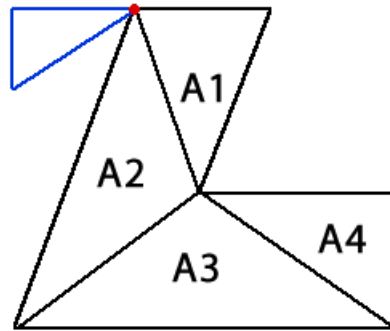
Obrazek 4.3.1.1. Wielokąty A i B po triangulacji

W kolejnym kroku należy wybrać dowolny punkt znajdujący się wewnątrz lub na brzegu jednego z wielokątów. W szczególności może być to jeden z wierzchołków wielokąta. Wybierzmy jeden z wierzchołków wielokąta B.



Obrazek 4.3.1.1. Wybór punktu z wielokąta B (czerwony kolor)

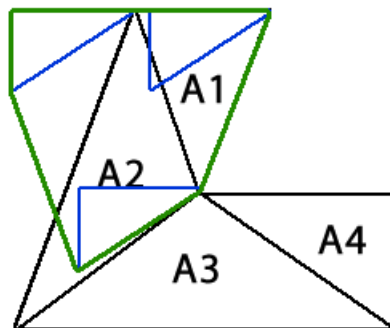
Po dokonaniu wyboru punktu rozpoczyna się główna faza algorytmu. W pierwszym kroku rozważane będą trójkąty A1 i B1. Na początek następuje translacja wybranego w poprzednim kroku punktu do jednego z wierzchołków trójkąta A1. Po wykonaniu tej operacji o taki sam wektor należy przesunąć wszystkie wierzchołki trójkąta B1. Efekt tej operacji zaprezentowano na obrazku poniżej.



Obrazek 4.3.1.1. Translacja wybranego punktu i trójkąta B1

Po wykonaniu opisanej wyżej translacji należy zapisać (np. na liście) współrzędne wierzchołków przesuniętego trójkąta B1, a następnie przeprowadzić translację wybranego wcześniej punktu do kolejnego wierzchołka trójkąta A1 oraz przesunąć o taki sam wektor wszystkie wierzchołki trójkąta B1. Ponownie należy zapisać nowe współrzędne wierzchołków trójkąta B1 oraz powtórzyć operację translacji dla ostatniego wierzchołka trójkąta A1. Na koniec, po raz kolejny należy zapisać współrzędne wierzchołków przesuniętego trójkąta B1.

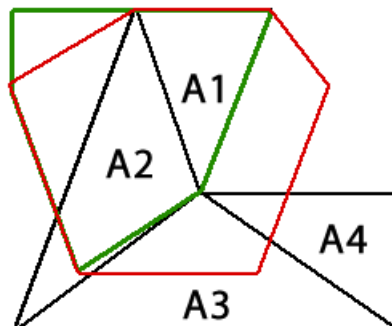
W następnym kroku należy wyznaczyć otoczkę wypukłą zbioru punktów zapisanych w poprzednim kroku. Efekt wyznaczania otoczki wypukłej zbioru tych punktów oraz odpowiednio przesunięty trójkąt B1 (do każdego z wierzchołków trójkąta A1) prezentuje poniższy obrazek.



Obrazek 4.3.1.1. Otoczkę wypukłą (kolor zielony) zapisanego zbioru punktów oraz przesunięty (do każdego z wierzchołków A1) trójkąt B1 (kolor niebieski)

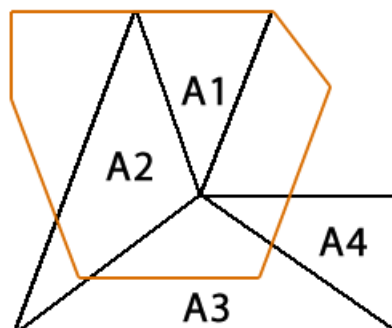
W kolejnym kroku należy powtórzyć opisane wyżej operacje dla pary trójkątów A1 i B2.

Ponownie wyznaczony zostanie zbiór punktów, dla którego należy wyznaczyć otoczkę wypukłą. Efekt zaprezentowano na obrazku poniżej.



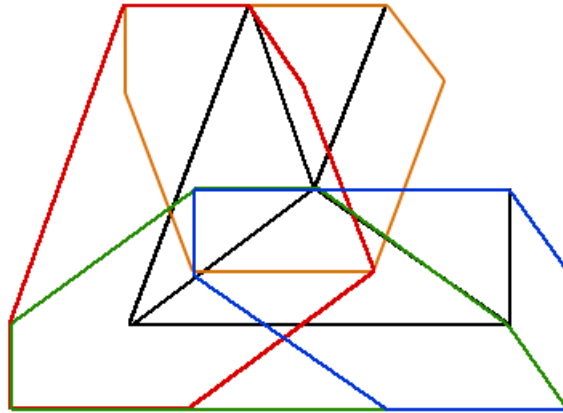
Obrazek 4.3.1.1. Otoczką wypukłą (kolor zielony) zapisanego zbioru punktów dla trójkątów $A1$ i $A2$

Wielokąt B nie zawiera już więcej nieprzetworzonych trójkątów, więc kolejnym krokiem jest złączenie powstałych w poprzednich krokach dwóch otoczek wypukłych w jeden wielokąt. Efekt złączenia wielokątów będących wyznaczonymi otoczkami wypukłymi przedstawiono na poniższym obrazku.



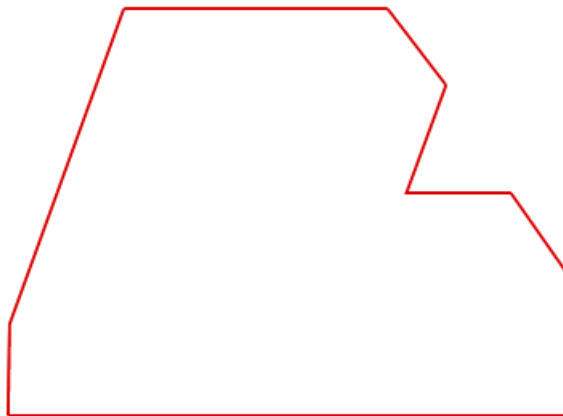
Obrazek 4.3.1.1. Efekt złączenia wyznaczonych w poprzednich krokach otoczek wypukłych

Po wykonaniu powyższych kroków powstał nowy wielokąt (kolor pomarańczowy) i przetwarzanie trójkąta $A1$ zostało zakończone. Następnie wszystkie operacje wykonane dla trójkąta $A1$ należy powtórzyć dla trójkąta $A2$. Po zakończeniu przetwarzania trójkąta $A2$ przetwarzany jest trójkąt $A3$, itd. Operacja zakończy się po przetworzeniu wszystkich trójkątów wchodzących w skład wielokąta A . Efekt przetworzenia wszystkich trójkątów zaprezentowano na poniższym obrazku.



Obrazek 4.3.1.1. Efekt przetworzenia wszystkich trójkątów wielokąta A

Ostatnim krokiem prowadzącym do uzyskania sumy Minkowskiego jest złączenie wszystkich nowo powstałych wielokątów (kolory pomarańczowy czerwony, zielony i niebieski). Efektem tej operacji jest wyznaczona suma Minkowskiego wielokątów A i B .



Obrazek 4.3.1.1. Suma Minkowskiego wielokątów A i B

Aby wyznaczyć złożoność algorytmu obliczania sumy Minkowskiego dwóch wielokątów przeanalizujemy złożoność występujących w nim operacji. Przyjmijmy następujące oznaczenia:

n - liczba wierzchołków pierwszego wielokąta,

m - liczba wierzchołków drugiego wielokąta.

Każdy wielokąt jest reprezentowany w postaci listy punktów.

W pierwszym kroku działania algorytmu znajdowana jest triangulacja każdego z wielokątów w czasie $O(p^2)$, gdzie p jest liczbą wierzchołków tworzących dany wielokąt. Następnie następuje przetwarzanie parami trójkątów z pierwszego wielokąta z trójkątami drugiego wielokąta. Dla każdej pary trójkątów trzykrotnie liczymy translację wszystkich wierzchołków jednego z trójkątów. Przyjmijmy że będzie to trójkąt należący do drugiego wielokąta. Operacja translacji wykonywana jest w czasie stałym - $O(1)$. Ponadto na koniec przetwarzania każdej z par liczona jest otoczka wypukła wyznaczonych przez algorytm punktów.

Złożoność średnia operacji wyznaczania otoczki wypukłej wynosi $O(k \log(k))$, gdzie k jest liczbą punktów, dla których ta otoczka jest wyznaczana. Jednak w naszym przypadku można uznać, że operacja ta jest wykonywana w czasie $O(1)$, gdyż niezależnie od liczby wierzchołków tworzących wielokąty dla których liczona jest suma Minkowskiego, otoczka będzie zawsze liczona dla $k = 9$.

Po zakończeniu przetwarzania jednego trójkąta należącego do pierwszego wielokąta z wszystkimi trójkątami drugiego wielokąta wykonywana jest operacja złączania powstałych w tym procesie otoczek wypukłych. Operacja ta wykonywana jest w czasie $O(\ell^2)$, gdzie ℓ jest łączną liczbą wierzchołków wchodzących w skład wyznaczonych otoczek wypukłych. Operacja złączania wielokątów jest wykonywana także na końcu działania algorytmu, po przetworzeniu wszystkich par.

Złożoność całego algorytmu można oszacować poprzez oszacowanie złożoności jego operacji dominującej. W tym wypadku operacją tą jest operacja, w której parami przetwarzane są wszystkie trójkąty pierwszego wielokąta z trójkątami drugiego wielokąta. Jej złożoność można oszacować w następujący sposób:

$$(n - 2) \cdot (m - 2) + (n - 2) \cdot (9 \cdot (m - 2))^2.$$

Liczby $n - 2$ oraz $m - 2$ oznaczają liczbę trójkątów na jakie został podzielony odpowiednio pierwszy i drugi wielokąt. Pierwszy składnik powyższej sumy opisuje więc złożoność przetworzenia parami wszystkich trójkątów i wyznaczenia kolejnych otoczek wypukłych. Drugi składnik opisuje złożoność operacji złączania otoczek wypukłych, która jest wykonywana dla każdego trójkąta pierwszego wielokąta. Składnik ten ma taką postać, gdyż $n - 2$ to oczywiście liczba trójkątów na jakie można podzielić pierwszy wielokąt, a $9 \cdot (m - 2)$ to z góry oszacowana liczba wierzchołków wchodzących w skład każdej otoczki wypukłej - jedna otoczka mogłaby mieć maksymalnie 9 wierzchołków, a dla jednego trójkąta pierwszego wielokąta powstanie $m - 2$ takich otoczek, gdyż na tyle trójkątów można podzielić drugi wielokąt. Ogólną złożoność algorytmu obliczającego sumę Minkowskiego można więc oszacować jako $O(n \cdot m^2)$.

4.3 Algorytm wyznaczania ścieżki

Algorytm wyznaczania ścieżki ma za zadanie wyznaczenie ścieżki od punktu początkowego do punktu końcowego z uwzględnieniem aktualnej mapy oraz pojazdu. Jego działanie można podzielić na trzy podstawowe etapy:

- powiększenie przeszkód i sprowadzenie pojazdu do punktu za pomocą sumy Minkowskiego,
- stworzenie odpowiedniego grafu,
- wyznaczenie ścieżki w utworzonym grafie, od punktu startowego do punktu końcowego przy pomocy algorytmu A^* ,
- wygenerowanie listy rozkazów dla znalezionej ścieżki.

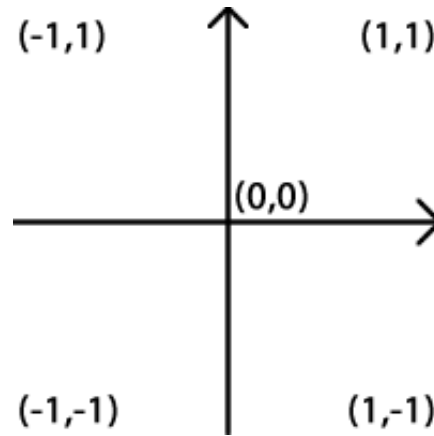
4.3.1 Opis algorytmu

Jako parametry wejściowe algorytm przyjmuje następujące dane:

- mapę,
- pojazd,
- początkowy obrót pojazdu,
- punkt startowy,
- punkt końcowy.

Oprócz danych wejściowych, użytkownik może przekazać do algorytmu także inne parametry, które mają wpływ między innymi na długość obliczeń oraz jakość znalezionej ścieżki. Parametry te zostały opisane poniżej.

- **Gęstość podziału kąta pełnego** - określa na ile jednostek zostanie podzielony kąt pełny. Im większa jest gęstość podziału, tym dokładniejsze obroty będzie mógł wykonywać pojazd, ale zwiększy się czas obliczeń. Zbyt mała wartość tego parametru, może spowodować, że pojazd nie będzie poruszał się zgodnie z ustalonym dla niego kierunkiem ruchu, gdyż ruch w takim kierunku będzie po prostu niemożliwy. Stosowny przykład został przedstawiony w dalszej części opisu.
- **Wielkość punktu** - określa maksymalną odległość między punktami, dla której algorytm, traktuje te punkty jako identyczne (przyjmuje wartości z przedziału $(0, 1]$). Aby lepiej zrozumieć wartości przyjmowane przez ten parametr, należy wziąć po uwagę fakt, iż zarówno współrzędne wielokątów tworzących mapę, jak i wielokąta tworzącego pojazd, są współrzędnymi w układzie współrzędnych przedstawionym na poniższym obrazku i mają wartości z przedziału $[-1, 1]$.



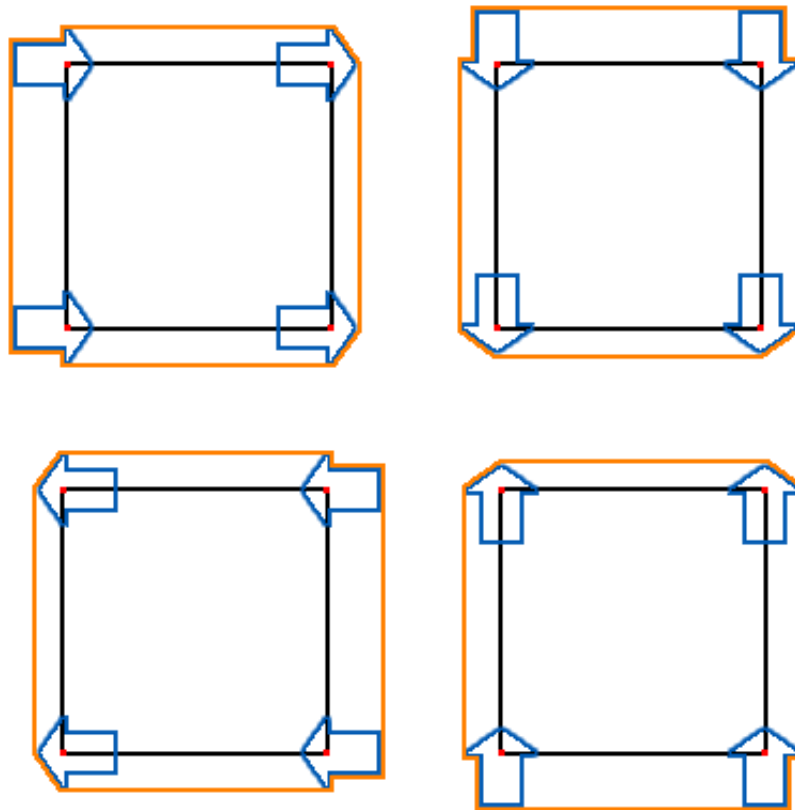
Obrazek 4.3.1.1. Układ współrzędnych

- **Waga krawędzi dla obrotu** - określa wagę krawędzi w grafie dla obrotu pojazdu o jedną jednostkę - im większa wartość, tym mniej obrotów wykona pojazd, gdyż ścieżka dłuższa wizualnie może okazać się mniej kosztowna z punktu widzenia algorytmu A^* ,
- **możliwość poruszania się do tyłu,**
- **możliwość poruszania się w dowolnym kierunku.**

Działanie algorytmu rozpoczyna się od dodania do mapy czterech sztucznych przeszkód, które posłużą jako ograniczenie obszaru mapy. Dodanie tych przeszkód nie zakłóci działania algorytmu, zapobiegnie natomiast możliwości opuszczenia przez pojazd widocznego obszaru mapy.

W następnym kroku, dla każdego możliwego obrotu pojazdu tworzona jest oddzielna kopia mapy, w której każda przeszkoda jest sumą Minkowskiego odpowiadającej jej przeszkody z mapy wejściowej i obróconego pojazdu. Utworzenie kopii mapy dla każdego możliwego obrotu pojazdu powoduje powstanie tablicy, zawierającej tyle kopii mapy wejściowej ile wynosi gęstość podziału kąta pełnego. Obiekt tej tablicy o indeksie *angle* reprezentuje mapę, dla której każda przeszkoda jest sumą Minkowskiego przeszkody z mapy wejściowej oraz pojazdu obróconego o *angle* jednostek. Jedna jednostka odpowiada obrotowi o $360/angleDensity$ stopni, gdzie *angleDensity* jest wartością odpowiadającą gęstości podziału kąta pełnego.

Przykład 4.3.1. Utworzone kopie mapy dla wszystkich możliwych obrotów pojazdu będącego strzałką (kolor niebieski) przy gęstości podziału kąta pełnego równej 4. Na czarno zaznaczona została przedstawiona przeszkoda z mapy wejściowej, natomiast na pomarańczowo suma Minkowskiego tej przeszkody i pojazdu obróconego o odpowiednią ilość jednostek.



Obrazek 4.3.1.2. Kopie mapy dla wszystkich możliwych obrotów pojazdu przy gęstości podziału kąta równej 4

Obrazek znajdujący się w lewym, górnym rogu na powyższym przykładzie przedstawia sumę Minkowskiego kwadratu i pojazdu w kształcie strzałki, obróconego o 0 jednostek, czyli o kąt 0° . Obrazek w prawym, górnym rogu przedstawia sumę Minkowskiego dla pojazdu obróconego o 1 jednostkę, czyli o 90° , a obrazki na dole odpowiednio dla pojazdu obróconego o 2 i 3 jednostki, czyli o 180° i 270° . Za każdym razem punktem przyłożenia pojazdu do przeszkody podczas obliczania sumy Minkowskiego jest punkt będący osią obrotu pojazdu (czerwony punkt).

Dzięki zastosowaniu sumy Minkowskiego dla wszystkich przeszkód znajdujących się na mapie, zostają one powiększone w taki sposób, że od tej pory pojazd można traktować jako punkt, a dokładniej jako punkt będący jego osią obrotu, gdyż względem tego punktu następuje przykładanie w algorytmie obliczającym sumę Minkowskiego. Z racji tego, iż sumy Minkowskiego zostały obliczone dla każdej kopii mapy, a więc każdego możliwego obrotu pojazdu, to pojazd został sprowadzony do punktu dla każdej możliwej jego orientacji.

Następnym krokiem działania algorytmu jest utworzenie serii grafów skierowanych. Ich liczba jest równa liczbie wszystkich możliwych obrotów pojazdów. Każdy graf odpowia-

da jednej kopii mapy, która powstała w poprzednim kroku. Wierzchołkami każdego grafu są wierzchołki przeszkód znajdujących się na odpowiadającej mu kopii mapy oraz punkt startowy i końcowy.

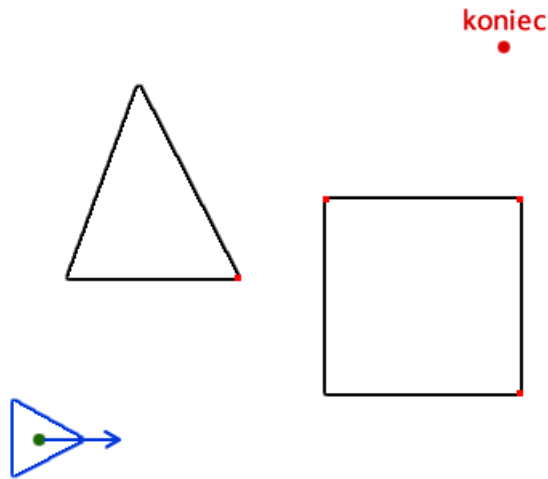
Mając utworzony graf dla każdej z kopii mapy należy dodać do niego odpowiednie krawędzie, po których będzie mógł poruszać się pojazd, który został sprowadzony do punktu. Dla danej kopii mapy przeglądamy parami wszystkie wierzchołki odpowiadającego jej grafu i dodajemy do niego krawędzie wg następujących reguł:

- jeśli dwa wybrane punkty mają identyczne współrzędne, krawędź nie zostanie dodana
- jeśli jakakolwiek współrzędna, jednego z wybranych punktów wychodzi poza zakres $[-1, 1]$, krawędź nie zostanie dodana
- jeśli jeden z wybranych wierzchołków to punkt startowy lub końcowy i punkt ten znajduje się wewnątrz którejkolwiek powiększonej przeszkody, krawędź nie zostanie dodana
- jeśli jeden z wybranych punktów jest zawarty w powiększonej przeszkodzie i nie jest jej wierzchołkiem, krawędź nie zostanie dodana
- jeśli krawędź powstała pomiędzy wybranymi punktami przecina jakąkolwiek krawędź, którejkolwiek z przeszkód i nie jest to przecinanie polegające na wspólnym początku lub końcu krawędzi, krawędź nie zostanie dodana
- jeśli środkowy punkt krawędzi powstałej pomiędzy wybranymi punktami leży wewnątrz jakiegokolwiek powiększonej przeszkody i krawędź ta, nie jest krawędzią tej przeszkody, krawędź nie zostanie dodana
- jeśli żaden z powyższych warunków nie jest spełniony, to sprawdzane są następujące warunki, zależne od parametrów algorytmu:
 - jeśli użytkownik wyraził zgodę na poruszanie się we wszystkich kierunkach, krawędź zostanie dodana z wagą równą przeskalowanej odległości pomiędzy wybranymi punktami
 - jeśli użytkownik nie wyraził zgody na poruszanie się we wszystkich kierunkach, a prosta przechodząca przez wybrane punkty jest odchylona o nie więcej niż $360/(2 \cdot angleDensity)$ stopni od prostej określającej kierunek ruchu pojazdu [patrz przykład niżej], krawędź zostanie dodana z wagą równą przeskalowanej odległości pomiędzy wybranymi punktami
 - * ponadto, jeśli użytkownik wyraził zgodę na poruszanie się do tyłu zostanie dodana krawędź powrotna pomiędzy wybranymi punktami o wadze równej przeskalowanej odległości pomiędzy nimi

Poniżej zaprezentowano przykład pokazujący w jaki sposób przebiega konstrukcja grafu dla jednej kopii mapy, czyli dla jednego obrotu pojazdu. Konstrukcja dla pozostałych kopii jest analogiczna.

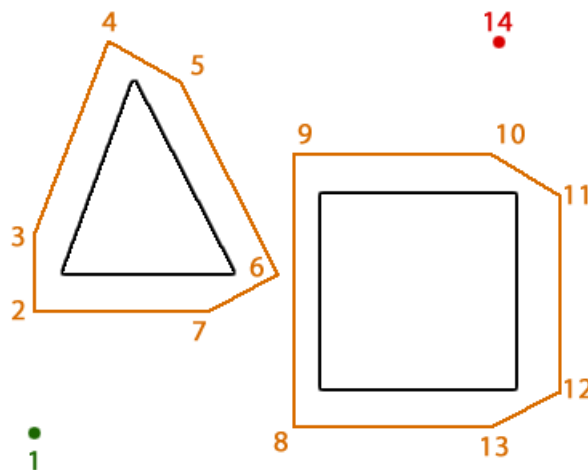
Przykład 4.3.2. Konstrukcja grafu dla jednej kopii mapy

Poniżej znajduje się obrazek przedstawiający mapę (czarne wielokąty), pojazd (niebieski wielokąt), punkt startowy (zielony) oraz punkt końcowy (czerwony). Strzałka wskazuje kierunek ruchu pojazdu.



Obrazek 4.3.1.3. Przykładowa mapa, pojazd, punkt startowy i końcowy

Po zastosowaniu sumy Minkowskiego w celu odpowiedniego powiększenia przeszkód, pojazd zostaje sprowadzony do punktu. Następnie utworzony zostaje graf odpowiadający mapie. Poniżej przedstawiono wygląd mapy po sprowadzeniu pojazdu do punktu wraz ze skojarzonymi numerami wierzchołków grafu.



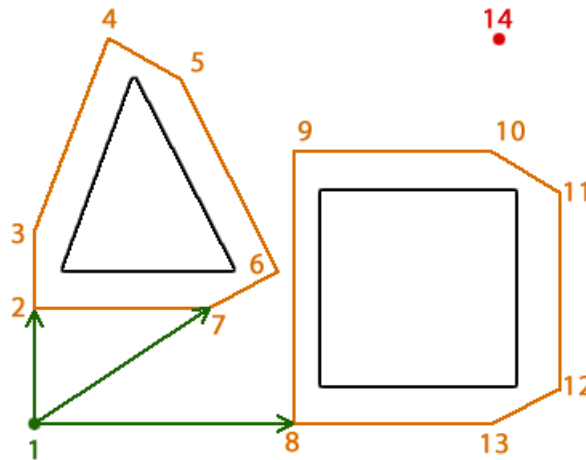
Obrazek 4.3.1.4. Przykładowa mapa po sprowadzeniu pojazdu do punktu Powiększone przeszkody zaznaczono kolorem pomarańczowym.

Kolejnym krokiem jest dodanie do grafu odpowiednich krawędzi. Przedstawiona zostanie konstrukcja grafu w trzech przypadkach: gdy pojazd może poruszać się we wszystkich kierunkach, gdy pojazd może poruszać się tylko do przodu oraz gdy dozwolone jest poruszanie się do tyłu.

Dla uproszenia zapisu krawędzie prowadzące od wierzchołka o numerze i do wierzchołka o numerze j będą oznaczane jako $i - j$.

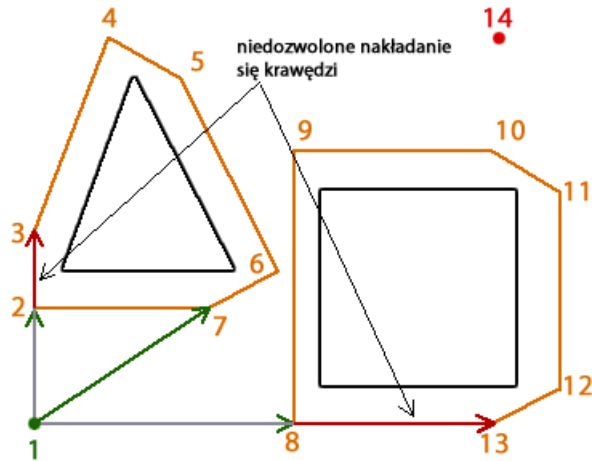
Przypadek 1. Pojazd może poruszać się w dowolnym kierunku

Jest to najprostszy przypadek, gdyż nie ma potrzeby sprawdzania czy prosta przechodząca przez wybrane punkty nie jest zbyt mocno odchylona od prostej odpowiadającej kierunkowi ruchu pojazdu. Na początek sprawdzane są wszystkie krawędzie wychodzące z wierzchołka numer 1. Dodane zostaną krawędzie do wierzchołków o numerach 2, 7, 8.



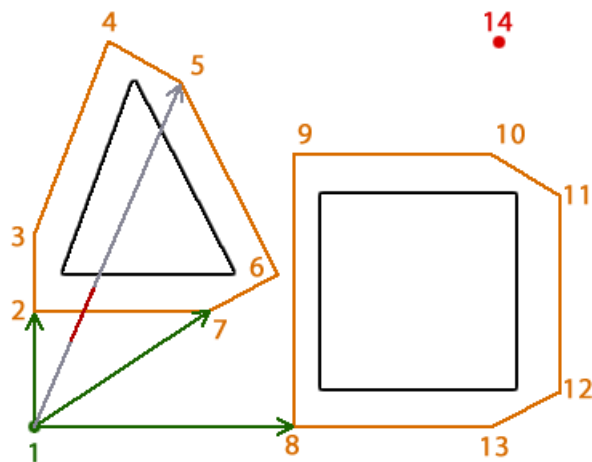
Obrazek 4.3.1.4. Dodane krawędzie 1-2, 1-7, 1-8

Krawędzie do wierzchołków numer 3 i 13 nie zostaną dodane. W przypadku krawędzi 1-3 zostanie wykryte jej przecięcie z krawędzią 2-3. Co prawda nie jest to przecięcie w dosłownym tego słowa znaczeniu, jednak częściowe nakładanie się krawędzi jest traktowane w taki sam sposób. Taka sama sytuacja ma miejsce w przypadku krawędzi 1-13 i 8-13.



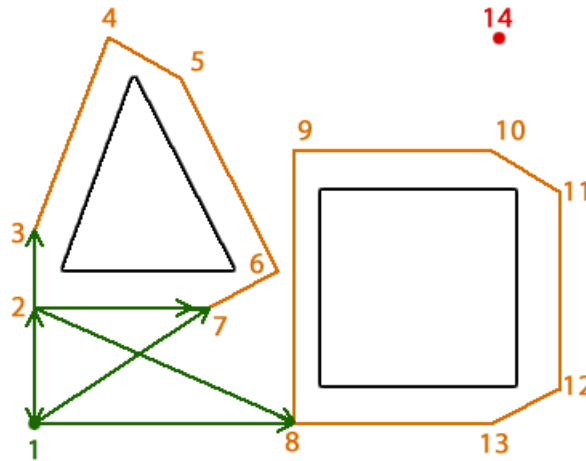
Obrazek 4.3.1.4. Próba dodania krawędzi 1-3 i 1-13

Nie zostaną dodane również krawędzie prowadzące do wierzchołków o numerach 4, 5, 6, 9, 10, 11, 12, 14. W każdym z tych przypadków, nowo dodana krawędź przecinałaby jedną z krawędzi powiększonych przeszkód.



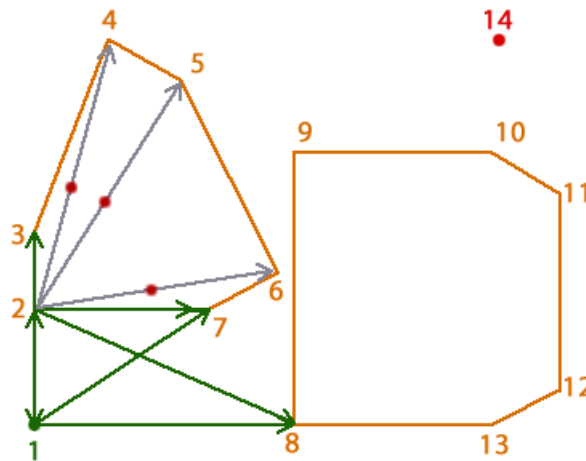
Obrazek 4.3.1.4. Próba dodania krawędzi 1-5 - niedozwolone przecięcie z krawędzią 2-7

Kiedy zakończone zostanie sprawdzanie krawędzi wychodzących z wierzchołka numer 1 następuje przejście do wierzchołka o numerze 2 i sprawdzanie wszystkich krawędzi wychodzących z tego wierzchołka. Nietrudno zauważyć, że dodane zostaną krawędzie do wierzchołków o numerach 1, 3, 7, 8. Co prawda na obrazku poniżej krawędź 2-8 przecina się z inną krawędzią grafu, jednak jest to bez znaczenia. Niedozwolone jest jedynie przecinanie się dodawanych krawędzi z krawędziami powiększonych przeszkód.



Obrazek 4.3.1.4. Dodane krawędzie 2-1, 2-3, 2-7 i 2-8

Krawędzie prowadzące do wierzchołków o numerach 4, 5, 6 nie zostaną dodane, gdyż środek każdej z nich leży wewnątrz powiększonej przeszkody. Na poniższym obrazku nie narysowano już przeszkód przed powiększeniem (kolor czarny) w celu poprawy jego czytelności. Rysowanie tych przeszkód w poprzednich krokach miało na celu pokazanie efektów działania sumy Minkowskiego i faktu, że pojazd można traktować jako punkt.

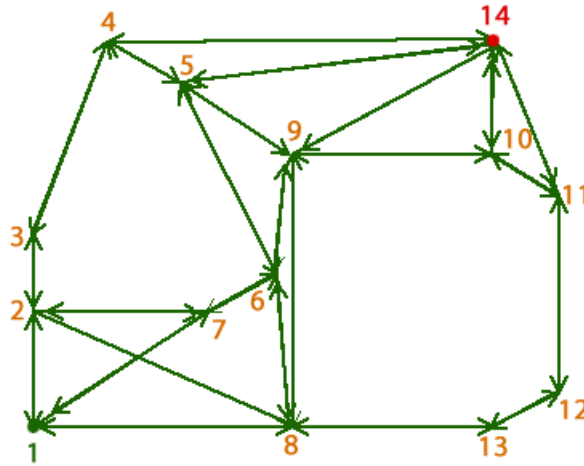


Obrazek 4.3.1.4. Próba dodania krawędzi 2-4, 2-5 i 2-6 - środki tych krawędzi leżą wewnątrz powiększonych przeszkód

Krawędzie prowadzące do wierzchołków o numerach 9, 10, 11, 12, 13, 14 nie zostaną dodane, gdyż nowo dodana krawędź przecinałaby jedną z krawędzi powiększonych przeszkód.

Analogiczne sprawdzanie, czy krawędź można dodać przeprowadzane jest dla kolejnych

par wierzchołków. Po zakończeniu tej operacji, graf odpowiadający jednemu obrotowi pojazdu jest gotowy. Został on przedstawiony na poniższym obrazku.



Obrazek 4.3.1.4. Gotowy graf dla jednego obrotu pojazdu (zgoda na poruszanie się w dowolnym kierunku)

W powyższym grafie istnieje ścieżka od punktu początkowego do punktu końcowego. Oznacza to, że możliwy jest przejazd pojazdu pomiędzy tymi punktami, w aktualnej orientacji, bez wykonywania dodatkowych obrotów.

Przypadek 2. Pojazd może poruszać się tylko do przodu

Ten przypadek jest nieco trudniejszy od poprzedniego, gdyż przy próbie dodania krawędzi pomiędzy dwoma wierzchołkami, należy dodatkowo sprawdzić, czy prosta poprowadzona przez te wierzchołki nie jest zbyt mocno odchylona od prostej odpowiadającej kierunkowi ruchu pojazdu. Maksymalne odchylenie w stopniach określone jest następującym wzorem:

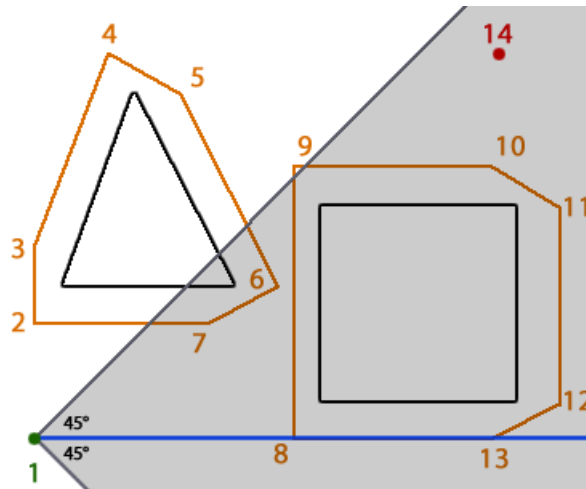
$$\frac{360}{2 \cdot angleDensity},$$

gdzie *angleDensity* jest gęstością podziału kąta pełnego.

Na potrzeby tego przykładu przyjęto, że gęstość ta wynosi 4. Zatem maksymalne odchylenie prostej przechodzącej przez dwa wierzchołki od prostej odpowiadającej kierunkowi ruchu pojazdu wynosi 45° . Przez tak małą gęstość podziału kąta pełnego, możliwa jest sytuacja, że pojazd nie będzie poruszał się idealnie w kierunku swojego ruchu, jednak znacząco upraszcza to przykład.

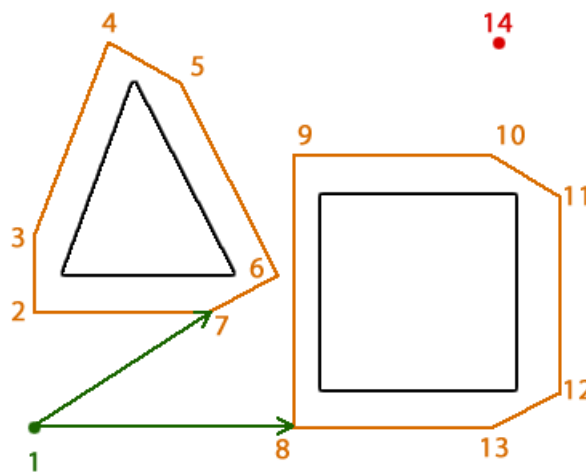
Podobnie jak w poprzednim przypadku na początku sprawdzane są wszystkie krawędzie wychodzące z wierzchołka o numerze 1. Z poprzedniego przypadku wiadomo, że jedynymi kandydatami na poprawne krawędzie są krawędzie 1-2, 1-7, 1-8. Zanim jednak którakol-

wiek z nich zostanie dodana do grafu, należy jeszcze sprawdzić, czy nie jest ona zbyt mocno odchylona od prostej odpowiadającej obrotowi pojazdu.



Obrazek 4.3.1.4. Maksymalne odchylenie od prostej wyznaczającej kierunek ruchu pojazdu

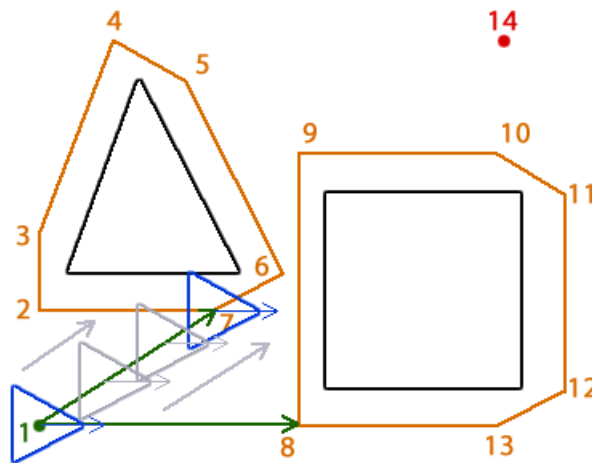
Na powyższym rysunku niebieską linią została narysowana półprosta wyznaczająca aktualny kierunek ruchu pojazdu, przyłożona do punktu numer 1. Z obu stron dorysowane zostały półproste odchylone od niej o 45° , czyli maksymalną możliwą wartość przy podziale kąta pełnego na 4 części. Szarym polem zakreslono obszar, wewnątrz którego muszą znajdować się wierzchołki, do których będzie można poprowadzić prawidłową krawędź z wierzchołkiem numer 1. Spośród trzech kandydatów, tylko dwa wierzchołki znajdują się wewnątrz szarego obszaru. Są to wierzchołki numer 7 i 8. Dodane więc zostaną krawędzie 1-7 i 1-8.



Obrazek 4.3.1.4. Dodane krawędzie 1-7 i 1-8

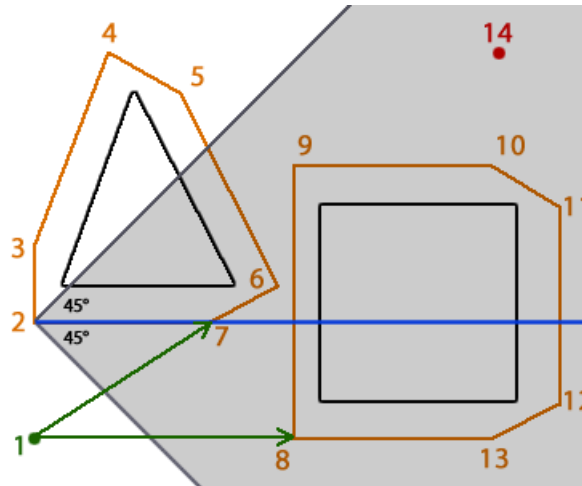
Przykład ten doskonale pokazuje problem, jaki może się pojawić przy zbyt małej gęstości

podziału kąta pełnego. Dla rozważanej kopii mapy i odpowiadającego jej grafu, pojazd jest w położeniu, w którym może poruszać się tylko po linii poziomej w prawą stronę. Nie będzie więc problemu dla krawędzi 1-8 i przejazdu z pozycji wierzchołka numer 1 do pozycji wierzchołka numer 8. Natomiast w przypadku krawędzi 1-7, przejazd z pozycji wierzchołka numer 1 do pozycji wierzchołka numer 7 będzie nienaturalny - pojazd pojedzie bokiem. Nie jest oczywiście pewne, że mimo takich krawędzi, jak krawędź 1-7 taka sytuacja wystąpi. Wszystko zależy od tego w jaki sposób grafy odpowiadające kolejnym obrotom pojazdu zostaną ze sobą połączone i jaka ścieżka zostanie znaleziona przez algorytm A*. Jeśli jednak krawędź 1-7 wejdzie w skład wyznaczonej ścieżki, to przejazd pojazdu będzie wyglądał jak na obrazku poniżej.



Obrazek 4.3.1.4. Przejazd pojazdu po krawędzi 1-7

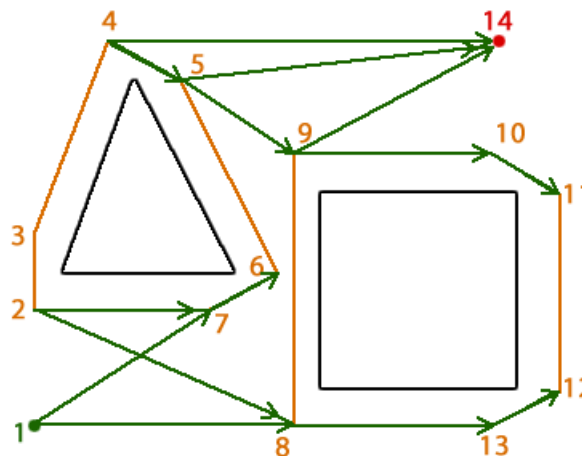
Kiedy zakończy się sprawdzanie krawędzi wychodzących z wierzchołka numer 1 następuje sprawdzenie wszystkich krawędzi wychodzących z wierzchołka o numerze 2. Podobnie jak dla wierzchołka numer 1 zakreślony został obszar pozwalający na łatwe stwierdzenie, czy daną krawędź należy dodać do grafu, czy odrzucić.



Obrazek 4.3.1.4. Maksymalne odchylenie od prostej wyznaczającej kierunek ruchu pojazdu

Z przypadku pierwszego wynika, że jedynymi kandydatami na prawidłowe krawędzie są krawędzie 2-1, 2-3, 2-7 i 2-8. Widać jednak, że tylko wierzchołki numer 7 i 8 leżą wewnątrz zakreślonego obszaru. Zostaną więc dodane tylko dwie krawędzie.

Operację sprawdzania krawędzi powtarzamy dla kolejnych wierzchołków par wierzchołków. Efekt końcowy został przedstawiony na obrazku poniżej.



Obrazek 4.3.1.4. Gotowy graf dla jednego obrotu pojazdu (zgodna na poruszanie się tylko do przodu)

Jak widać w przypadku, gdy dozwolone jest poruszanie się tylko do przodu, to graf odpowiadający jednemu obrotowi pojazdu ma znacznie mniej krawędzi. W grafie zaprezentowanym na obrazku powyżej nie istnieje ścieżka od punktu początkowego do punktu końcowego. Z tego wynika, że nie jest możliwy przejazd pojazdu w aktualnej orientacji, od punktu początkowego do punktu końcowego bez wykonania żadnego dodatkowego obrotu.

Przypadek 3. Pojazd może poruszać się do przodu i do tyłu

Przypadek ten jest bardzo podobny do poprzedniego. Jedyna różnica polega na tym, że podczas dodawania krawędzi w jedną stronę pomiędzy parą wierzchołków, dodawana jest również krawędź powrotna. Dzięki tej dodatkowej krawędzi pojazd może poruszać się do tyłu.

Po utworzeniu grafu dla każdej kopii mapy konieczne jest odpowiednie połączenie tych grafów w jeden duży graf. Dla uproszczenia graf ten nazywany będzie *grafem warstwowym*, a każdy z grafów odpowiadających kolejnym kopiom mapy, *warstwą* tworzonego grafu warstwowego. Krawędzie mogą być dodawane tylko pomiędzy sąsiednimi warstwami. Dodatkowo należy dodać odpowiednie krawędzie pomiędzy pierwszą i ostatnią warstwą grafu. Krawędzie pomiędzy warstwami grafów odpowiadają obrotom pojazdu.

Dodawanie krawędzi pomiędzy warstwami rozpoczyna się od warstwy pierwszej i drugiej. Dla każdego wierzchołka z warstwy pierwszej, jeżeli nie leży on we wnętrzu jakiejkolwiek powiększonej przeszkody, sprawdzana jest odległość do każdego z wierzchołków drugiej warstwy. Jeśli odległość ta jest mniejsza niż odległość określona w parametrze *wielkość punktu*, to sprawdzane jest położenie wierzchołka z drugiej warstwy. Jeśli, podobnie jak wierzchołek z pierwszej warstwy, nie leży on we wnętrzu żadnej powiększonej przeszkody, to krawędź pomiędzy tymi wierzchołkami jest dodawana (jest to krawędź łącząca dwie warstwy), a jej waga jest równa wartości parametru *waga krawędzi obrotu*. Następnie sprawdzane są możliwości połączeń pomiędzy warstwą drugą i trzecią, itd. Na koniec należy jeszcze w odpowiedni sposób połączyć pierwszą i ostatnią warstwę (gdyż odpowiadają one sąsiednim obrotom pojazdu).

Po zakończeniu operacji dodawania krawędzi pomiędzy warstwami, graf warstwowy jest już prawie gotowy. Należy do niego dodać jeszcze jeden wierzchołek, a następnie dodać krawędzie prowadzące od wierzchołka końcowego każdej warstwy do tego wierzchołka. Dodanie tego wierzchołka sprawia, że możliwe jest szukanie ścieżki pomiędzy wierzchołkiem startowym, z warstwy odpowiadającej początkowemu obrotowi pojazdu, a tym dodanym wierzchołkiem.

Po utworzeniu grafu warstwowego należy wyznaczyć ścieżkę od wierzchołka startowego, z warstwy odpowiadającej początkowemu obrotowi pojazdu, do wierzchołka końcowego (ostatniego dodanego do grafu wierzchołka). Ścieżka wyznaczana jest przy pomocy algorytmu A^* , który de facto sprowadza się do algorytmu Dijkstry. Dzieje się tak, ponieważ dla grafu warstwowego, skonstruowanego w poprzednich krokach bardzo ciężko zdefiniować rozsądną heurystykę. Z tego powodu przyjęte zostało, że $\forall_v h(v) = 0$, gdzie $h(v)$ - przewidywana przez heurystykę droga od wierzchołka v do wierzchołka docelowego. Złożoność czasową algorytmu A^* można oszacować jako $O(|E|)$, gdzie $|E|$ - liczba wszystkich krawędzi

grafu warstwowego.

Ostatnim etapem działania algorytmu jest wygenerowanie listy rozkazów, które posłużą do przygotowania wizualizacji ruchu pojazdu od punktu startowego do końcowego. Można ją oczywiście wygenerować tylko wtedy, gdy algorytm A* znalazł odpowiednią ścieżkę w grafie warstwowym. Pojedynczy rozkaz zawiera następujące informacje:

- punkt docelowy, do którego powinien przemieścić się pojazd,
- wartość kąta do jakiego powinien być obrócony pojazd przed rozpoczęciem ruchu w kierunku punktu docelowego. Jeżeli wartość ta jest z przedziału $[-2\pi, 0)$ to obrót nastąpi w kierunku zgodnym z ruchem wskazówek zegara, natomiast w przypadku, gdy wartość tego kąta jest z przedziału $[0, 2\pi)$, to obrót będzie przeciwny do ruchu wskazówek zegara.

Konstrukcja listy rozkazów wymaga przejrzenia wszystkich wierzchołków wchodzących w skład znalezionej ścieżki. Konstrukcja pojedynczego rozkazu przebiega w następujący sposób:

- jako punkt docelowy przypisywana jest lokalizacja badanego wierzchołka,
- odnajdywany jest numer warstwy grafu, do której należy badany wierzchołek i na tej podstawie, znając gęstość podziału kąta pełnego, możliwe jest wyznaczenie wartości bezwzględnej wymaganego obrotu, jest ona z przedziału $[0, 2\pi)$,
- wyznaczona wartość obrotu jest modyfikowana w celu wykonania obrotu w odpowiednią stronę - kąt ujemny oznacza obrót zgodnie z ruchem wskazówek zegara, natomiast kąt nieujemny, przeciwnie do ruchu wskazówek zegara. Znak kąta wyznaczany jest na podstawie poniższych reguł:
 - jeżeli bezwzględna wartość kąta obrotu dla konstruowanego rozkazu wynosi 0, a wartość kąta obrotu dla poprzedniego rozkazu jest mniejsza od $-\pi$, to kąt 0 zostanie zamieniony na -2π ,
 - jeżeli wartość kąta obrotu dla poprzedniego rozkazu jest dodatnia i jest ona większa od wartości bezwzględnej kąta obrotu dla konstruowanego rozkazu, to wartość kąta obrotu dla konstruowanego rozkazu zostanie zamieniona na ujemną poprzez odjęcie 2π ,
 - jeżeli wartość kąta obrotu dla poprzedniego rozkazu jest równa 0 i wartość bezwzględna kąta obrotu dla konstruowanego rozkazu jest większa od π , to wartość kąta obrotu dla konstruowanego rozkazu zostanie zamieniona na ujemną poprzez odjęcie 2π ,
 - jeżeli wartość kąta obrotu dla poprzedniego rozkazu jest ujemna to sprawdzone zostają następujące warunki:

- * jeżeli wartość bezwzględna kąta obrotu dla poprzedniego rozkazu jest większa od wartości bezwzględnej kąta obrotu dla konstruowanego rozkazu, to wartość kąta obrotu dla konstruowanego rozkazu zostanie zamieniona na ujemną poprzez odjęcie 2π ,
- * jeżeli wartość kąta obrotu dla poprzedniego rozkazu jest równa -2π i wartość bezwzględna kąta obrotu dla konstruowanego rozkazu jest większa od π , to wartość kąta obrotu dla konstruowanego rozkazu zostanie zamieniona na ujemną poprzez odjęcie 2π .

Po wykonaniu wszystkich powyższych kroków algorytm kończy działanie i zwraca wygenerowaną listę rozkazów.

4.3.2 Pseudokod

W tej części dokumentu zostanie przedstawiony pseudokod algorytmu wyznaczania ścieżki. Zostały z niego wyodrębnione pewne funkcje w celu poprawy czytelności.

Pseudokod 4.3.2.1. Algorytm wyznaczania ścieżki

```

dodaj do mapy przeszkody znajdujące się na krawędziach mapy
utwórz tyle kopii mapy, ile jest możliwych obrotów pojazdu
dla każdego możliwego obrotu pojazdu
    dla każdej przeszkody znajdującej się na odpowiadającej
    obrotowi kopii mapy
        oblicz sumę Minkowskiego tej przeszkody i obróconego pojazdu
dla każdej kopii mapy
    utwórz graf skierowany o liczbie wierzchołków równej liczbie
    wierzchołków wszystkich przeszkód + 2
dla każdej kopii mapy
    dla każdej pary wierzchołków
        jeżeli można dodać krawędź pomiędzy wierzchołkami
            jeżeli dozwolony ruch w dowolnym kierunku
                dodaj krawędź pomiędzy wierzchołkami
            w przeciwnym wypadku
                jeżeli prosta przechodząca przez wybrane wierzchołki
                nie jest zbyt mocno odchylona od prostej wyznaczającej
                kierunek ruchu pojazdu
                    dodaj krawędź pomiędzy wierzchołkami
                jeżeli dozwolony ruch do tyłu
                    dodaj krawędź powrotną pomiędzy wierzchołkami
utwórz graf warstwowy z grafów odpowiadających kopiom map i

```

```

dodatkowy wierzchołek
dla każdej warstwy
    dla każdego wierzchołka z danej warstwy
        jeżeli wierzchołek odpowiada punktowi końcowemu
            dodaj krawędź do ostatniego wierzchołka grafu warstwowego
        jeżeli wierzchołek nie leży wewnątrz żadnej powiększonej
        przeszkody
            dla każdego wierzchołka z następnej warstwy
                jeżeli odległość pomiędzy wybranymi wierzchołkami jest
                mniejsza niż określona przez parametr 'wielkość punktu'
                i wierzchołek z drugiej warstwy nie leży wewnątrz żadnej
                powiększonej przeszkody
                    dodaj krawędź pomiędzy wierzchołkami
wybierz wierzchołek startowy z warstwy odpowiadającej początkowemu
obrotowi pojazdu
wyznacz ścieżkę od wierzchołka początkowego do ostatniego wierzchołka
za pomocą algorytmu A*
jeżeli ścieżka została znaleziona
    utwórz listę rozkazów
zwróć listę rozkazów

```

Pseudokod 4.3.2.2. Algorytm sprawdzający czy można dodać krawędź pomiędzy x i y

```

jeżeli  $x$  jest wierzchołkiem startowym lub końcowym
    dla każdej powiększonej przeszkody mapy
        jeżeli  $x$  jest zawarte w tej powiększonej przeszkodzie
            return false
jeżeli  $y$  jest wierzchołkiem startowym lub końcowym
    dla każdej powiększonej przeszkody mapy
        jeżeli  $y$  jest zawarte w tej powiększonej przeszkodzie
            return false
jeżeli, któraś współrzędna  $x$  lub  $y$  wykracza poza zakres mapy
    return false
dla każdej powiększonej przeszkody mapy
    jeżeli krawędź  $x$ - $y$  przecina jakąkolwiek krawędź przeszkody
        jeżeli przecinanie nie polega na wspólnym początku lub końcu
            return false
    jeżeli wierzchołek  $x$  lub  $y$  jest zawarty w przeszkodzie i nie
    jest jej wierzchołkiem
        return false
    jeżeli środkowy punkt krawędzi  $x$ - $y$  jest zawarty w przeszkodzie

```

```

        jeżeli krawędź x-y nie jest krawędzią przeszkody
            return false
return true

```

Pseudokod 4.3.2.3. Algorytm konstrukcji listy rozkazów

```

utwórz rozkaz z wierzchołka startowego:
    - jako punkt docelowy przypisz lokalizację punktu startowego
    - jako kąt obrotu przypisz początkowy obrót pojazdu
dla każdego wierzchołka zawartego w wyznaczonej ścieżce
    utwórz rozkaz:
        - ustaw punkt docelowy rozkazu na lokalizację wierzchołka
        - ustaw kąt obrotu na kąt odpowiadający warstwie do której
          należy wierzchołek startowy
    jeżeli kąt obrotu rozkazu == 0
        i kąt obrotu poprzedniego rozkazu <= -pi
            kąt obrotu rozkazu -= 2*pi
    jeżeli kąt obrotu poprzedniego rozkazu > 0
        jeżeli kąt obrotu rozkazu < kąt obrotu poprzedniego rozkazu
            kąt obrotu rozkazu -= 2*pi
    jeżeli kąt obrotu poprzedniego rozkazu == 0
        jeżeli kąt obrotu rozkazu > pi
            kąt obrotu rozkazu -= 2*pi
    jeżeli kąt obrotu poprzedniego rozkazu < 0
        jeżeli wartość bezwzględna kąta obrotu rozkazu
            < wartość bezwzględna kąta obrotu poprzedniego rozkazu
            kąt obrotu rozkazu -= 2*pi
    jeżeli kąt obrotu poprzedniego rozkazu == -2*pi
        i kąt obrotu rozkazu > pi
            kąt obrotu rozkazu -= 2*pi

```

4.3.3 Złożoność algorytmu

Złożoność algorytmu wyznaczania ścieżki można oszacować poprzez oszacowanie złożoności jego operacji dominującej, czyli operacji tworzenia kolejnych warstw grafu warstwowego. Złożoność tej operacji można opisać przy pomocy następującego wzoru:

$$angleDensity \cdot vertices \cdot (vertices - 1) \cdot canTwoPointsConnectCost,$$

gdzie:

- *angleDensity* - gęstość podziału kąta pełnego

- *vertices* - liczba wierzchołków tworzących wszystkie powiększone przeszkody na mapie
- *canTwoPointConnect* - złożoność funkcji służącej do sprawdzania czy można dodać krawędź pomiędzy dwoma punktami

Pesymistyczną złożoność funkcji służącej do sprawdzania czy można dodać krawędź pomiędzy dwoma punktami można oszacować jako $O(n^2)$. Wynika to z faktu, iż w pętli przeglądane są wszystkie powiększone przeszkody tworzące mapę, sprawdzane jest, czy żaden z punktów, pomiędzy którymi ma zostać dodana krawędź nie jest zawarty w jakiejś przeszkodzie (sprawdzenie zawierania ma złożoność liniową), a w przypadku gdy algorytm wykryje zawieranie, przeglądane są wszystkie wierzchołki danej przeszkody i sprawdzane jest, czy żaden z punktów pomiędzy którymi ma być dodana krawędź nie jest jednym z wierzchołków powiększonej przeszkody (tutaj już złożoność robi się kwadratowa).

Dla oszacowania pesymistycznej złożoności operacji tworzenia kolejnych warstw grafu warstwowego można przyjąć, że $angleDensity = vertices$. Wówczas oznaczając jako n liczbę wszystkich wierzchołków tworzących powiększone przeszkody na mapie, złożoność operacji tworzenia kolejnych warstw grafu można oszacować jako $O(n^5)$.

Z powyższego oszacowania wynika, że złożoność całego algorytmu wyznaczania ścieżki można szacować jako $O(n^5)$, gdzie n jest liczbą wierzchołków tworzących powiększone przeszkody na mapie.

Idąc jeszcze dalej, można przedstawić złożoność algorytmu w zależności od liczby wierzchołków tworzących mapę wejściową oraz liczby wierzchołków tworzących kształt pojazdu. Zakładając, że nie są znane informacje na temat tego w jaki sposób funkcja *Union* zredukuje liczbę wierzchołków podczas łączenia wielokątów, można oszacować, że złożoność całego algorytmu wynosi $O((k \cdot \ell)^5)$, gdzie k - liczba wierzchołków tworzących przeszkody na mapie, a ℓ - liczba wierzchołków tworzących kształt pojazdu.

Rozdział 5

Analiza porównawcza dla algorytmu wyznaczania ścieżki

Rozdział 6

Instrukcja użytkownika

Interfejs aplikacji został przedstawiony na obrazku 6.1.



Obrazek 6.1. Interfejs aplikacji

W górnej części interfejsu umieszczone zostało menu główne. Użytkownik ma możliwość przejścia do zakładki "Start", "Konfiguracja", "Symulacja" oraz powrotu do poprzedniej zakładki za pomocą przycisku "Wstecz". Ponadto w prawym górnym rogu interfejsu znajduje się przycisk "Ustawienia" pozwalający na przejście do zakładki z ustawieniami głównymi aplikacji.

6.1 Edytor map i pojazdów

Przejdzie do edytora map lub pojazdów z ekranu startowego możliwe jest poprzez wybranie zakładki “Konfiguracja”, a następnie podzakładki “Edytor map” lub “Edytor pojazdów”.

6.1.1 Edytor map

Interfejs edytora map został przedstawiony na obrazku 6.1.1.1.



Obrazek 6.1.1.1. Interfejs edytora map

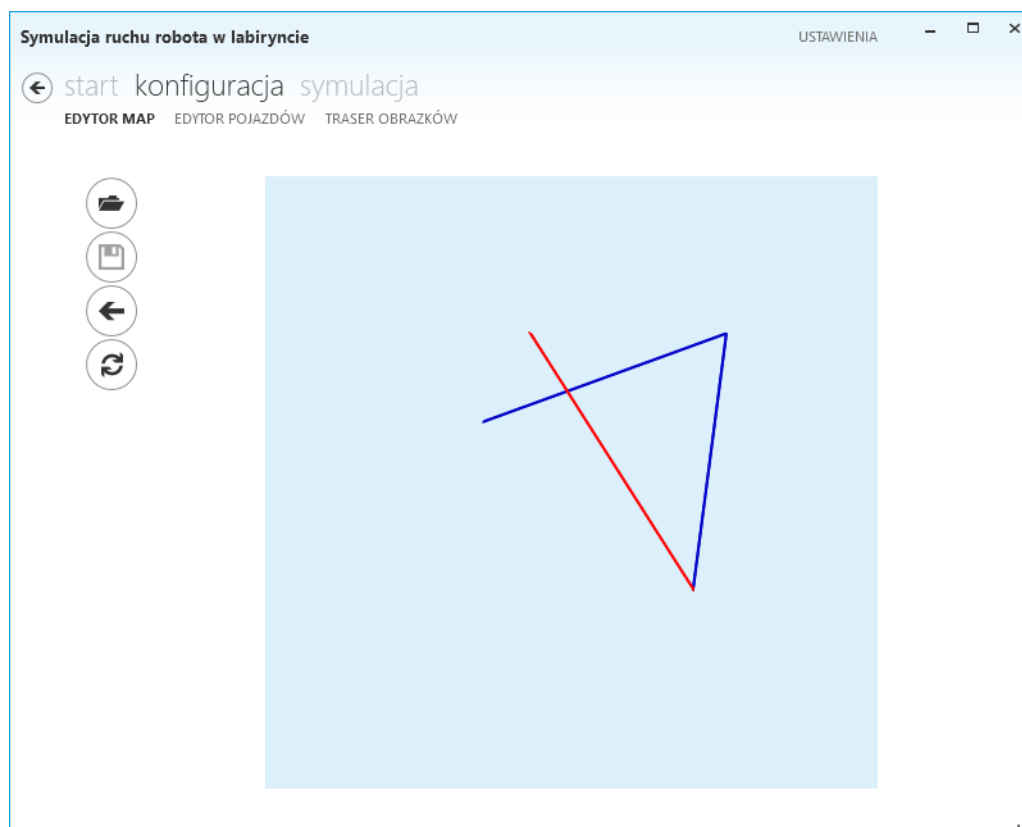
Użytkownik ma do dyspozycji następujące przyciski:

- 1 - **Wczytaj mapę** - umożliwia wczytanie i edycję uprzednio zapisanej mapy z pliku SVG.
- 2 - **Zapisz mapę** - umożliwia zapisanie mapy do pliku SVG.
- 3 - **Cofnij** - umożliwia cofnięcie ostatnio wykonanej akcji (np. usunięcie ostatniej krawędzi).
- 4 - **Wyczyść** - czyści obszar edytora.

Aby rozpocząć tworzenie nowej mapy należy dodać do niej pierwszy punkt. Można to uczynić klikając lewym przyciskiem myszy w obszarze edytora (błękitne okienko). Po dodaniu pierwszego punktu pojawi się linia, która będzie podążać za kursorem, aż do momentu zakończenia edycji wielokąta. Kolor tej linii informuje użytkownika, czy możliwe jest dodanie

nowego punktu w miejscu, w którym obecnie znajduje się kursor. Kolor zielony oznacza, że nowo dodana krawędź będzie prawidłowa, natomiast kolor czerwony informuje użytkownika że nowa krawędź przecinałaby jedną z krawędzi obecnie tworzonego wielokąta. W tym przypadku dodanie punktu nie będzie możliwe.

Dodawanie kolejnych punktów do wielokąta odbywa się za pomocą kliknięcia lewym przyciskiem myszy. Istnieje również możliwość usunięcia ostatnio dodanego punktu. Aby tego dokonać wystarczy kliknąć prawym przyciskiem myszy lub kliknąć przycisk “Cofnij”. W celu zakończenia edycji wielokąta należy kliknąć w pobliżu jego pierwszego punktu. Edytor automatycznie zamknie wielokąt, a jego kolor zostanie zmieniony na kolor czarny, co oznacza że nie jest on już aktywny.



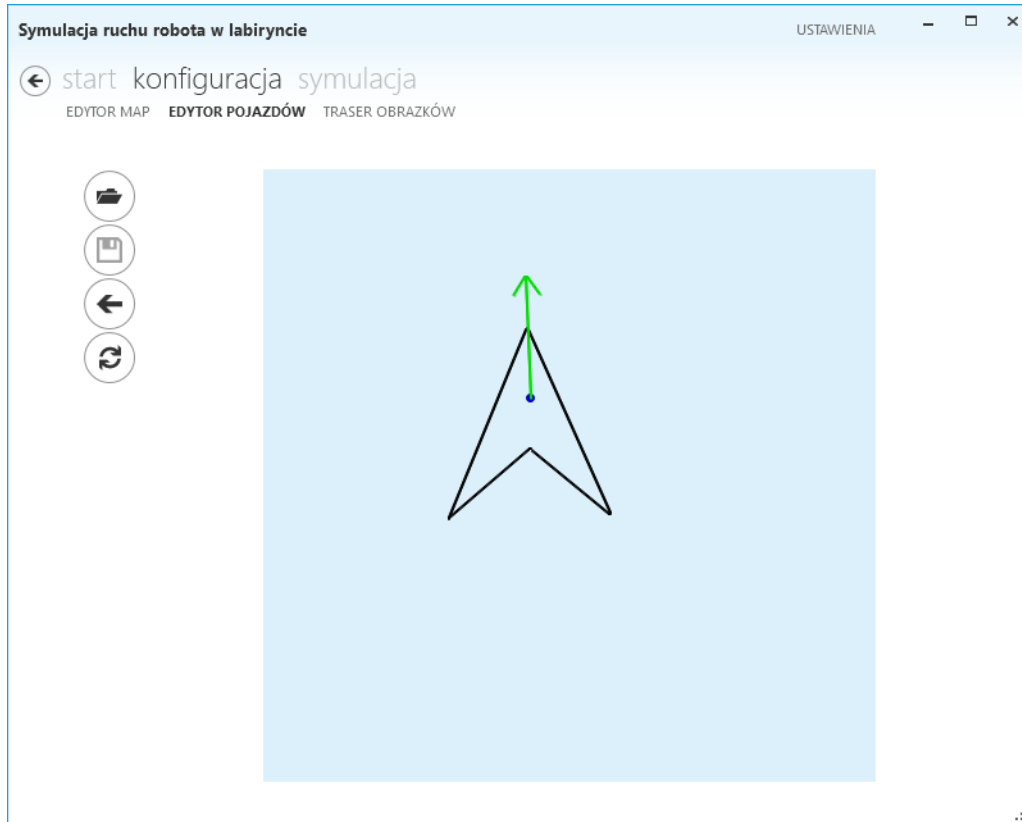
Obrazek 6.1.1.2. Próba dodania niepoprawnego punktu

Po dodaniu jednego wielokąta użytkownik ma możliwość dodania kolejnych. Nowe wielokąty mogą przecinać krawędzie utworzonych wcześniej wielokątów. Po zakończeniu tworzenia mapy należy ją zapisać, aby możliwe było jej użycie podczas tworzenia symulacji.

6.1.2 Edytor pojazdów

Interfejs edytora pojazdów wygląda identycznie jak interfejs edytora map. Tworzenie wielokąta będącego kształtem pojazdu również przebiega w taki sam sposób jak w edytorze map.

W przypadku tworzenia pojazdu można jednak utworzyć tylko jeden wielokąt. Następnym krokiem jest wybranie punktu wewnątrz tego wielokąta, który będzie jego osią obrotu. Podczas działania algorytmu wyznaczania ścieżki, to właśnie do tego punktu sprowadzany jest pojazd. Ostatnim krokiem jest wybranie kierunku jazdy pojazdu, czyli de facto jego przodu. Operacja ta została zaprezentowana na obrazku poniżej.

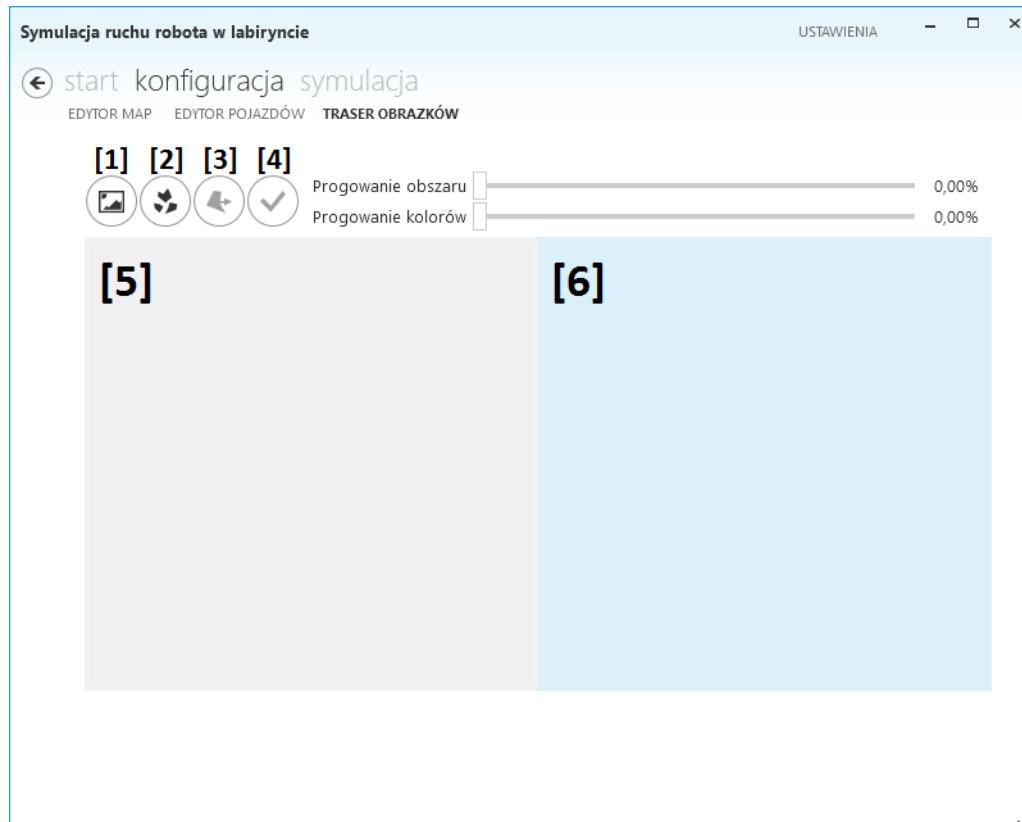


Obrazek 6.1.2.1. Ustalanie kierunku jazdy pojazdu

Po wykonaniu wymienionych wyżej operacji pojazd jest gotowy. Aby możliwe było jego użycie podczas tworzenia symulacji należy go zapisać.

6.2 Traser plików graficznych

Przejsie do traseru plików graficznych z ekranu startowego możliwe jest poprzez wybranie zakładki “Konfiguracja”, a następnie podzakładki “Traser obrazków”. Interfejs traseru plików graficznych został przedstawiony na poniższym obrazku.



Obrazek 6.2.1. Interfejs traseru plików graficznych

Główną część interfejsu stanowią dwa okna oznaczone na obrazku numerami 5 i 6. Okno po lewej stronie będzie odpowiedzialne za wyświetlanie wczytanego pliku graficznego, natomiast okno po prawej stronie prezentować będzie wyniki trasowania.

W interfejsie traseru plików graficznych użytkownik ma do dyspozycji cztery przyciski:

- 1 - **Wczytaj obrazek** - umożliwia wczytanie nowego pliku graficznego .
- 2 - **Stwórz mapę** - przycisk jest aktywny zawsze, umożliwia stworzenie mapy z wybranych wielokątów. Po kliknięciu użytkownik zostaje przeniesiony do edytora map.
- 3 - **Stwórz pojazd** - przycisk jest aktywny tylko wtedy, gdy wybrany jest jeden wielokąt, umożliwia stworzenie pojazdu. Po kliknięciu użytkownik zostaje przeniesiony do edytora pojazdów.
- 4 - **Trasuj obrazek** - rozpoczyna operację trasowania wczytanego pliku graficznego.

Poza wymienionymi przyciskami w skład interfejsu wchodzi również dwa suwaki, pozwalające na dobór następujących parametrów trasowania:

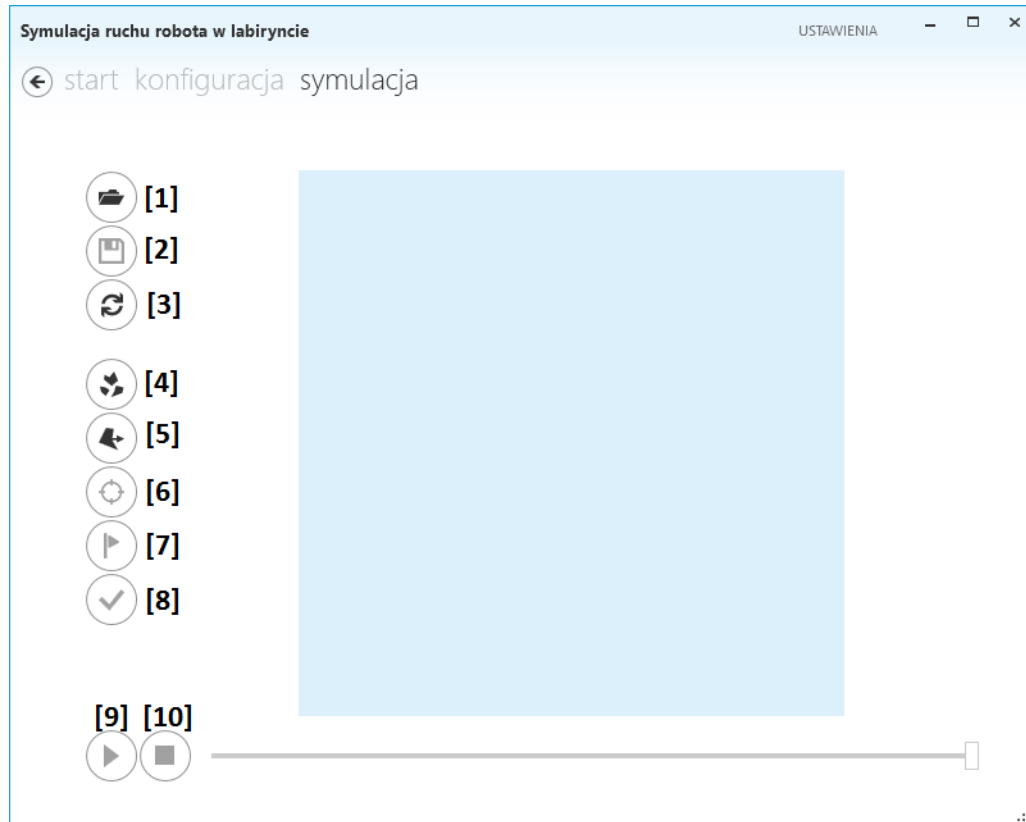
- *Wartość progowania obszaru* - procentowa (relatywnie do całego obrazka) wartość określająca maksymalną powierzchnię odrzucanych wielokątów. Niezerowa wartość pozwala na usunięcie kształtów wygenerowanych z artefaktów pliku graficznego lub po prostu odrzucenie małych, niepożądanych przez użytkownika elementów.
- *Wartość progowania kolorów* - procentowy próg określający czy grupy pikseli o danym kolorze powinny być traktowane jako kształty, czy tło. Im mniejsza wartość tego parametru, tym ciemniejszy musi być kształt, aby algorytm zinterpretował go jako wielokąt, a nie element tła.

Cały proces trasowania jest dość prosty i rozpoczyna się od wczytania pliku graficznego. Następnie użytkownik może odpowiednio dobrać parametry trasowania i kliknąć na przycisk "Trasuj obrazek". Po tych operacjach w oknie po prawej stronie pojawia się wyniki trasowania.

Jeśli użytkownik chce utworzyć mapę, może wybrać dowolną liczbę wielokątów, a następnie kliknąć przycisk "Stwórz mapę". Jeśli natomiast utworzony ma być pojazd, należy wybrać tylko jeden wielokąt i kliknąć przycisk "Stwórz pojazd". Wyboru wielokąta można dokonać za pomocą kliknięcia lewym przyciskiem myszy. Wybrany wielokąt zostaje podświetlony na niebiesko. Aby zrezygnować z wyboru danego wielokąta należy na nim kliknąć prawym przyciskiem myszy.

6.3 Moduł symulacji

Przejsie do modułu umożliwiającego przygotowanie i przeprowadzenie symulacji możliwe jest poprzez wybranie zakładki "Symulacja". Interfejs modułu symulacji został przedstawiony na poniższym obrazku.



Obrazek 6.3.1. Interfejs modułu symulacji

W interfejsie modułu symulacji użytkownik ma do dyspozycji następujące przyciski:

- 1 - **Wczytaj symulację** - umożliwia wczytanie i edycję uprzednio zapisanej symulacji z pliku SVG.
- 2 - **Zapisz symulację** - umożliwia zapisanie symulacji do pliku SVG.
- 3 - **Wyczyść** - resetuje wszystkie elementy symulacji.
- 4 - **Wczytaj mapę** - umożliwia wczytanie uprzednio zapisanej mapy z pliku SVG.
- 5 - **Wczytaj pojazd** - umożliwia wczytanie uprzednio zapisanego pojazdu z pliku SVG.
- 6 - **Wybierz punkt startowy** - umożliwia wybranie punktu w którym znajdzie się oś obrotu pojazdu, a następnie dostosowanie jego wielkości i początkowego obrotu.
- 7 - **Wybierz punkt końcowy** - umożliwia wybranie punktu końcowego, do którego będzie obliczana ścieżka, którą ma pokonać pojazd.

8 - Oblicz ścieżkę - rozpoczyna działanie algorytmu obliczania ścieżki.

9 - Odtwórz/Wstrzymaj - przycisk jest aktywny, gdy znaleziona została ścieżka od punktu startowego do punktu końcowego. Umożliwia rozpoczęcie lub wstrzymanie animacji przejazdu.

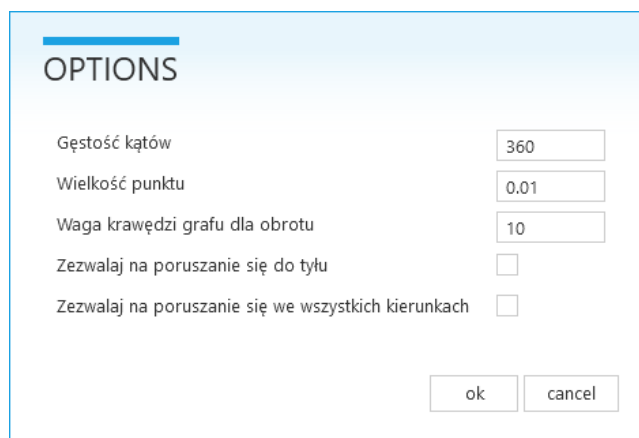
10 - Zatrzymaj - przycisk jest aktywny, gdy rozpoczęto odtwarzanie animacji. Umożliwia jej zatrzymanie i powrót pojazdu do punktu startowego.

Oprócz wymienionych wyżej przycisków, w dolnej części interfejsu znajduje się suwak umożliwiający ręczny podgląd symulacji. Możliwe jest także rozpoczęcie odtwarzania symulacji od wybranego miejsca.

Aby utworzyć nową symulację należy wykonać następujące operacje:

- wczytać mapę
- wczytać pojazd
- wybrać punkt startowy oraz dopasować rozmiar i początkowy obrót pojazdu (wielokąt reprezentujący pojazd nie może przecinać się z żadną przeszkodą)
- wybrać punkt końcowy (nie może być zawarty w żadnym z wielokątów reprezentujących przeszkody)
- kliknąć przycisk “Oblicz ścieżkę” oraz dobrać następujące parametry działania algorytmu:
 - gęstość podziału kąta pełnego - określa na ile jednostek zostanie podzielony kąt pełny (ma wpływ na długość obliczeń algorytmu)
 - wielkość punktu - określa maksymalną odległość między punktami, dla której algorytm traktuje te punkty jako identyczne
 - wagę krawędzi dla obrotu - określa wagę krawędzi w grafie dla obrotu pojazdu o jedną jednostkę - im większa wartość, tym mniej obrotów będzie chciał wykonać pojazd
 - zgoda na poruszanie się do tyłu
 - zgoda na poruszanie się w dowolnym kierunku

Okno dialogowe umożliwiające dobór wymienionych parametrów zostało przedstawione na obrazku poniżej.



OPTIONS

Gęstość kątów

Wielkość punktu

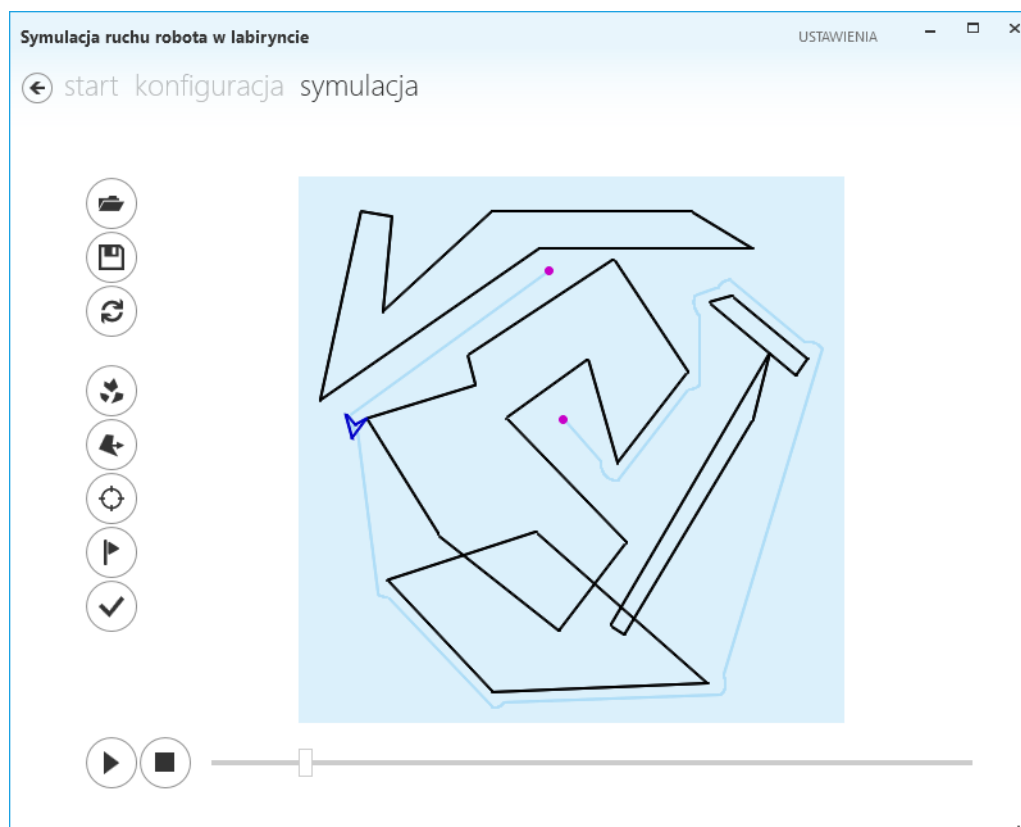
Waga krawędzi grafu dla obrotu

Zezwalaj na poruszanie się do tyłu ☐

Zezwalaj na poruszanie się we wszystkich kierunkach ☐

Obrazek 6.3.2. Opcje algorytmu obliczania ścieżki

Po wykonaniu powyższych kroków i zakończeniu obliczeń, możliwe będzie odtworzenie animacji ukazującej przejazd pojazdu od punktu startowego do punktu końcowego lub zostanie wyświetlony komunikat informujący o tym, iż ścieżka pomiędzy tymi punktami nie została znaleziona. Dodatkowo, jeśli w ustawieniach aplikacji włączono opcję prezentowania znalezionej ścieżki, zostanie ona wyświetlona w obszarze symulacji. Wynik działania algorytmu z widoczną ścieżką został zaprezentowany na poniższym obrazku.

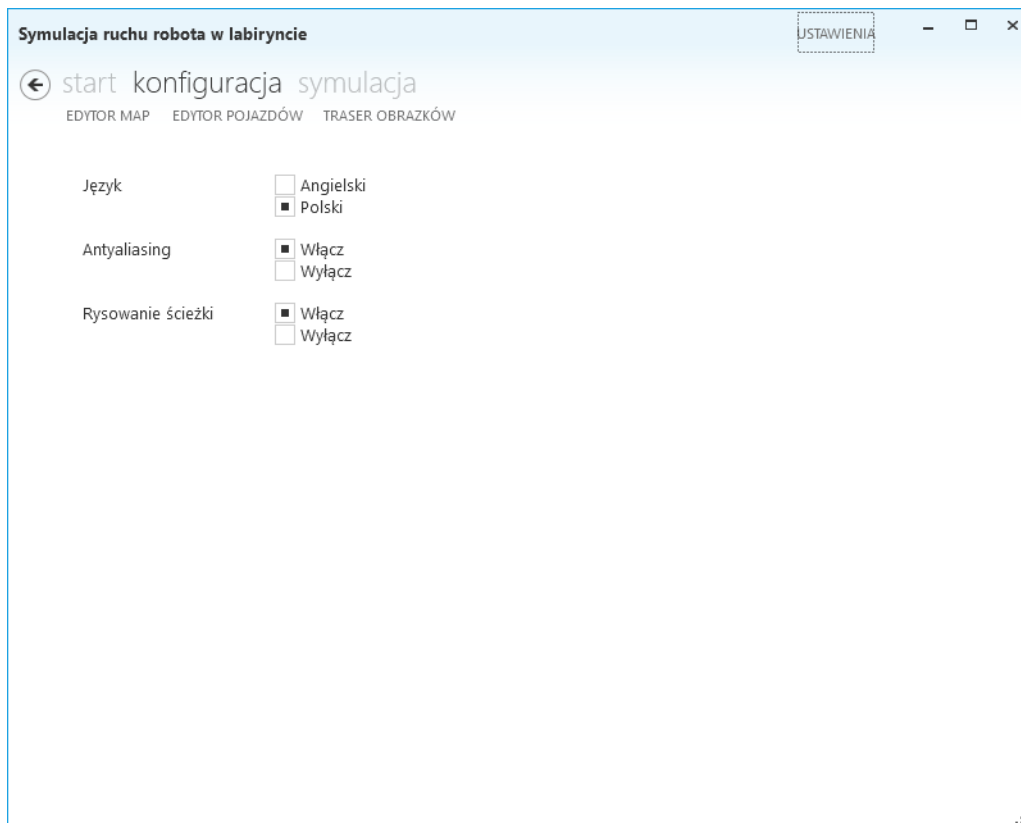


Obrazek 6.3.3. Wynik działania algorytmu

Gotową symulację można zapisać do pliku SVG, klikając uprzednio przycisk "Zapisz symulację". Można również wczytać nową mapę, nowy pojazd, na nowo wyznaczyć punkt startowy i końcowy, lub jednym kliknięciem w przycisk "Wyczyść" rozpocząć tworzenie symulacji od nowa. Zapisaną symulację można podejrzeć za pomocą przeglądarki internetowej wspierającej format SVG.

6.4 Ustawienia aplikacji

Interfejs zakładki z ustawieniami aplikacji został przedstawiony na poniższym obrazku.



Obrazek 6.4.1. Ustawienia aplikacji

W ustawieniach użytkownik może wybrać wersję językową aplikacji (polską lub angielską), włączyć lub wyłączyć rysowanie linii z antyaliasingiem oraz włączyć lub wyłączyć podgląd i przewijanie znalezionej ścieżki przy tworzeniu symulacji. Aby zmiany dotyczące języka aplikacji były widoczne, konieczny jest restart całej aplikacji.

Rozdział 7

Przebieg pracy

7.1 Model wytwórczy

Praca dyplomowa została napisana w modelu przyrostowym (ang. *incremental development*) i składała się z czterech etapów (ang. *milestones*). Model ten zakładał implementację kompletnego modułu w każdym z etapów prac. Każdy etap kończył się dostarczeniem kodu źródłowego ukończonego modułu, testów jednostkowych oraz dokumentacji technicznej modułu.

Preferencja modelu przyrostowego nad modelem kaskadowym wynikała ze struktury naszej aplikacji. Można ją podzielić na wiele odrębnych modułów, działających (do pewnego stopnia) oddzielnie i niezależnie od siebie. Ponadto, ta metodologia zakłada brak konieczności definiowania z góry szczegółowych wymagań dla poszczególnych części systemu, dzięki czemu proces tworzenia jest bardziej elastyczny niż w przypadku modelu kaskadowego.

7.2 Harmonogram i podział prac

W tej części pracy dyplomowej opisano szczegółowo podział prac w każdym z etapów oraz terminy ich zakończenia.

7.2.1 Etap 1

Termin zakończenia etapu: **17.11.2015**.

Pierwszy etap prac nad projektem obejmował przygotowanie architektury, odpowiednich modeli klas oraz ręcznego edytora map i pojazdów. Podział zadań w tym etapie wyglądał w następujący sposób:

- zaprojektowanie architektury systemu (Mateusz Pielat)
- zdefiniowanie reprezentacji pojazdu i mapy (Konrad Miśkiewicz)

- przygotowanie kontrolki umożliwiającej osadzenie MonoGame w WPF (Wojciech Kowalik)
- przygotowanie ręcznego edytora map i pojazdów (wszyscy)

7.2.2 Etap 2

Termin zakończenia etapu: **01.12.2015.**

Głównym zadaniem drugiego etapu było przygotowanie modułu umożliwiającego trasowanie plików graficznych. Ponadto dodane zostały metody umożliwiające serializację i deserializację map oraz pojazdów. Podział zadań w tym etapie:

- serializacja i deserializacja map oraz pojazdów (Wojciech Kowalik)
- przygotowanie klasy opakowującej bibliotekę D3DPotrace, umożliwiającą trasowanie plików graficznych (Mateusz Pielat)
- implementacja wielowątkowego trasowania plików z użyciem klasy opakowującej bibliotekę D3DPotrace (Konrad Miśkiewicz)

7.2.3 Etap 3

Termin zakończenia etapu: **15.12.2015.**

Celem trzeciego etapu prac nad projektem było przygotowanie narzędzia generującego losowe listy rozkazów ruchu pojazdu oraz wizualizacji ruchu pojazdu na podstawie tych rozkazów. Podział zadań podczas realizacji trzeciego etapu:

- zdefiniowanie reprezentacji rozkazów ruchu pojazdu (Mateusz Pielat)
- przygotowanie narzędzia generującego losową listę rozkazów ruchu pojazdu (Konrad Miśkiewicz)
- implementacja obiektu symulacji, wczytywania mapy, pojazdu, ustawiania pozycji i obrotu początkowego pojazdu oraz punktu końcowego (Wojciech Kowalik)
- implementacja metody przekształcającej listę rozkazów ruchu pojazdu na animację ruchu pojazdu (Mateusz Pielat)
- implementacja suwaka osi czasu wizualizacji (Wojciech Kowalik)

7.2.4 Etap 4

Planowany termin zakończenia etapu: **12.01.2016**, ostateczna data: **19.01.2016**.

Ostatni etap prac nad projektem obejmował implementację algorytmu wyszukiwania ścieżki. Do zadań tego etapu należała:

- implementacja pomocniczych algorytmów geometrycznych (wszyscy)
- implementacja algorytmu obliczającego sumę Minkowskiego (Wojciech Kowalik)
- implementacja algorytmu tworzącego graf warstwowy i wyszukującego ścieżkę od punktu początkowego do punktu końcowego (Konrad Miśkiewicz)
- eksport i mapowanie wyników obliczeń do listy rozkazów ruchu pojazdu

Ostatni etap nie został w pełni zakończony w terminie. Problemem okazała się implementacja algorytmu wyszukiwania ścieżki i wystąpienie przypadków szczególnych, które nie zostały wcześniej przez nas przewidziane.

Bibliografia

- [1] *XAML Overview*, <https://msdn.microsoft.com/en-us/library/cc189036>
- [2] *Wprowadzenie do wzorca projektowego MVVM na przykładzie aplikacji WPF*, <http://msdn.microsoft.com/pl-pl/library/wprowadzenie-do-wzorca-projektowego-model-view-viewmodel-na-przykladzie-aplikacji-wpf.aspx>
- [3] *MVVM Light Toolkit Documentation*, <http://www.mvVMLight.net/doc/>
- [4] *W3Schools SVG Tutorial*, <http://www.w3schools.com/svg/>
- [5] *MonoGame Documentation*, <http://www.monogame.net/documentation/>
- [6] *Clipper - an open source freeware library for clipping and offsetting lines and polygons.*, <http://www.angusj.com/delphi/clipper.php>
- [7] dr Jan Bródka, *Wykłady z przedmiotu "Algorytmy i Struktury Danych 2"*
- [8] dr inż. Paweł Kotowski, *Wykłady z przedmiotu "Grafika Komputerowa"*

Warszawa, dnia

Oświadczenie

Oświadczam, że pracę inżynierską pod tytułem: „Symulacja ruchu robota w labiryncie”, której promotorem jest dr Paweł Rzażewski, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....
Wojciech Kowalik
Konrad Miśkiewicz
Mateusz Pielat