POLITECHNIKA GDAŃSKA
WYDZIAŁ FIZYKI TECHNICZNEJ I MATEMATYKI STOSOWANEJ

Imię i nazwisko studenta: Patryk Pazdro
Nr albumu: 143817
Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Fizyka Techniczna
Specjalność/profil: Informatyka stosowana

PROJEKT DYPLOMOWY INŻYNIERSKI

Tytuł projektu w języku polskim: Program komputerowy służący do symulacji ruchu ulicznego

Tytuł projektu w języku angielskim: A computer program for simulating traffic flow

Potwierdzenie przyjęcia projektu	
Opiekun projektu podpis	Kierownik Katedry/Zakładu podpis
dr inż. Szymon Winczewski	

Data oddania projektu do dziekanatu:

Streszczenie

Na wstępie przedstawiono przegląd literatury dotyczącej symulacji ruchu drogowego za pomocą automatów komórkowych. Następnie ukazano jak niektóre przykłady z literatury odnoszą się do opracowanego modelu. Później zostały omówione biblioteki SFML i ImGui, których użyto przy tworzeniu aplikacji. Przedstawiono ich funkcjonalności, oraz zostały pokazane i skomentowane przykładowe kody źródłowe. Następnie opisano model automatu komórkowego, wraz z zarysem historycznym oraz przykładami użycia. Opisano również automat komórkowy używający reguły 30. Następnie zostały wyjaśnione zasady działania modelu Nagela–Schreckenberga, wraz z przykładami użycia, oraz opisano co pomógł wyjaśnić. Kolejno omówiono model VDR, oraz wyjaśniono czym się różni od modelu Nagela–Schreckenberga. Kolejną częścią pracy jest opis opracowanego modelu, który symuluje ruch drogowy na siatce dróg jednokierunkowych, które mogą się krzyżować, a wszystkie skrzyżowania są równorzędne. Przedstawiono reguły obowiązujące na skrzyżowaniu i poza nim. Później została opisana implementacja modelu. Omówiono najważniejsze klasy, graficzny interfejs użytkownika oraz jak model został zwizualizowany. Na końcu pracy został opisany przykład użycia wraz z diagramem fundamentalnym opracowanego modelu oraz dyskusją co można by udoskonalić w opracowanym modelu aby wyeliminować powstające zakleszczenia.

Słowa kluczowe:
Informatyka, Model ruchu miejskiego, Symulacja, System czasu rzeczywistego, Automaty komórkowe, Wizualizacja

Abstract

At the outset, overview of the literature on cellular automata based traffic simulation is provided. Then, it is showed how examples from the literature are related to the developed model. Later SFML and ImGui libraries which were used to create the application are discussed. Libraries functionalities are presented and source code examples are showed and commented. Then cellular automaton model is described, along with the historical outline and examples of use. Also cellular automaton rule 30 is described. Next rules of Nagel-Schreckenberg model are explained, along with examples of use. There is also described what Nagel-Schreckenberg model helped to explain. Next VDR model is discussed and how it differs from Nagel-Schreckenberg model. Another part of the work is a description of the created model, which simulates traffic on the one-way road network, in which roads can cross with each other, and all intersections follow the rule priority to the right. Rules in force at the crossing and beyond are presented. Later implementation of the model is described. The most important classes, a graphical user interface and how the model was visualized are discussed. At the end of the work an example of usage is described, along with the fundamental diagram of the model and discussion of what can be improved in the developed model to eliminate the resulting deadlocks.

Key words:
Informatyka, Urban traffic model, Simulation, Real-time system, Cellular automata, Visualisation

Spis treści

Rozdział 1

Wstęp i cel pracy

1.1 Wstęp

W gęsto zaludnionych miejscach drogi są często zapełnione do granic możliwości, co prowadzi do korków i powoduje znaczące straty ekonomiczne. W takich obszarach ciężko wybudować nowe ulice. Symulacja ruchu drogowego może pomóc podczas kontrolowania i projektowania systemów transportowych. Dzięki niej można przewidzieć krytyczne miejsca i wytyczyć inny kurs dla pojazdów, by pomóc w odciążeniu danej drogi.

W 1992 Nagel i Schreckenberg zaproponowali minimalny model służący do symulacji ruchu drogowego (NaSch). Model zawierał tylko jedną, jednopasmową drogę. Minimalny model czyli taki, że wszelkie inne uproszczenia prowadziłyby do nierealistycznych zachowań pojazdów. Bazując na modelu NaSch, Rickert *et al.* (1996) zaproponowali prosty model zawierający drogę dwupasmową. Wagner *et al.* (1997) wprowadzili zestaw reguł dla zmian pasów w symulacji ruchu drogowego zawierającej wiele pasów. Knospe *et al.* (1999) omówili wpływ wolnych pojazdów na ruch na dwupasmowej drodze. Tonguz *et al.* (2009) wprowadzili nowe podejście do symulacji ruchu drogowego, pozwalające na skonstruowanie modelu mobilności ruchu miejskiego.

Opracowany model bazuje na regułach modelu Nagela–Schreckenberga i VDR poza skrzyżowaniami. W pracy Tonguz *et al.* na skrzyżowaniach zastosowano światła, natomiast w opracowanym modelu skrzyżowania są równorzędne i nie ma na nich świateł.

1.2 Cel projektu

Celem pracy jest stworzenie programu komputerowego służącego do symulacji ruchu ulicznego. Program ma stanowić implementację modelu opartego na automacie komórkowym (model Nagela–Schreckenberga oraz pochodne) i oprócz przeprowadzenia symulacji umożliwiać także wizualizację jej przebiegu. Program ma umożliwiać przeprowadzenie symulacji dla dwuwymiarowej siatki ulic jednokierunkowych z możliwością krzyżowania się ulic. Program ma zostać napisany w języku C++, zaś do wizualizacji przebiegu symulacji oraz do stworzenia interfejsu użytkownika mają zostać wykorzystane biblioteki oparte na OpenGL.

Rozdział 2

Technologie

2.1 SFML

SFML [1] czyli prosta i szybka multimedialna biblioteka (Simple and Fast Multimedia Library) została stworzona przez Laurenta Gomila. Napisano ją w C++, do renderingu używa OpenGL. SFML jest biblioteką wieloplatformową. Można ją kompilować, a następnie uruchamiać na platformach takich jak: Windows, Linux i Mac OS X. Dostarcza interfejs umożliwiający obsługę różnych elementów komputera. Jest złożona z pięciu modułów: `system`, `window` (okno), `graphics` (grafika), `aaudio` (dźwięk) i `network` (sieć). Moduł `system` dostarcza obsługę wektorów, kodowanie tekstu w standardzie unicode, zarządzanie wątkami oraz klasy ułatwiające obsługę czasu. Moduł `window` jest odpowiedzialny za tworzenie okna bazującego na OpenGL oraz dostarczaniu funkcji do zarządzania zdarzeniami i urządzeniami wejściowymi. Moduł `graphics` udostępnia klasy do zarządzania grafiką, takie jak: `image` (obrazek), `text` (tekst), `shapes` (kształty) itp. Moduł `audio` pozwala na dodawanie i manipulowanie dźwiękiem. Moduł `network` dostarcza narzędzi do obsługi komunikacji sieciowej.

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}
```

Rysunek 2.1: Przykładowy kod korzystający z biblioteki SFML tworzący okno z narysowanym kołem.

Aby stworzyć prosty przykładowy program za pomocą biblioteki SFML (Rys. 2.1), najpierw należy zaimportować bibliotekę. Następnie w głównej funkcji programu `main` należy, stworzyć okno za pomocą `sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!")`. Konstruktor jako argumenty przyjmuje obiekt `sf::VideoMode(200, 200)`, który definijuje wielkość okna oraz nazwę okna. `sf::CircleShape shape(100.f)` pozwala stworzyć koło o zadanym promieniu. Zastosowana pętle `while` sprawia, że program będzie się wykonywał dopóki okno jest otwarte, co jest sprawdzane za pomocą metody `window.isOpen()`. Następnie należy sprawdzić, czy zostało wykonane jakieś wydarzenie, np. okno zostało zamknięte `if (event.type == sf::Event::Closed)`. Wydarzenie to informuje, że należy zamknąć okno przy użyciu metody `window.close()`. Co każde wykonanie pętli należy wyczyścić okno, za pomocą `window.clear()`. Funkcja `window.draw(shape)` tworzy w oknie, obiekt z modułu `graphics`, który przyjmuje jako argument, w tym przypadku wcześniej stworzone koło. Metoda `window.display()` wyświetla na ekran komputera wszystkie obiekty które zostały dotychczasowo stworzone w oknie za pomocą metody `window.draw()`.

2.2 ImGui

Immediate mode GUI [2] (graficzny interfejs użytkownika w trybie natychmiastowym) jest biblioteką służącą do tworzenia graficznego interfejsu użytkownika w C++. Dodaje funkcjonalność tworzenia okien w aplikacji, w których można wyświetlać różne elementy interfejsu użytkownika, między innymi: suwaki, przyciski, obrazki, teksty, itp. Biblioteka daje możliwość przesuwania, zamykania, minimalizowania i manipulowania wielkością okien. ImGui tworzy i rysuje okna w każdym kroku działania aplikacji. Nie trzeba tworzyć obiektów klasy do każdego elementu interfejsu jak w standardowych bibliotekach, lecz wywołuje się tylko metody z biblioteki. Metoda musi być wywołana aby element interfejsu istniał, stąd nazwa graficzny interfejs użytkownika w trybie natychmiastowym.

```
if (!ImGui::Begin("Menu", true, ImVec2(0,0))
{
        ImGui::End();
        return;
}
ImGui::Text("Tekst");
if (ImGui::Button("Button")) {
        printf("button");
}
ImGui::End();
```

Rysunek 2.2: Przykładowy kod korzystający z biblioteki ImGui tworzący okno z tekstem i przyciskiem.

Aby wyświetlić przykładowe okno w aplikacji za pomocą ImGui (Rys 2.2) wystarczy: posłużyć się metodą `ImGui::Begin("Menu", true, ImVec2(0,0))`, gdzie pierwszy argument konstruktora to nazwa okna. Drugi to wartość typu binarnego która mówi, czy okno jest otwarte. Ostatni argument to wektor zawierający pozycję w którym okno ma się znajdować. Jeżeli okno jest zamknięte to funkcja zwraca fałsz i należy je natychmiastowo zamknąć używając `ImGui::End()`. Natomiast jeżeli zwróci prawdę, to można wyświetlić zawartość okna. Funkcja `ImGui::Text("Tekst")` wyświetli

napis "Tekst". `ImGui::Button("Button")` wyświetli przycisk z napisem "Button"i zwraca wiadomość czy jest on wciśnięty. Jeżeli jest naduszony to w konsoli zostanie wyświetlony komunikat "button". Na koniec należy zamknąć okno używając `ImGui::End()`.

Rozdział 3

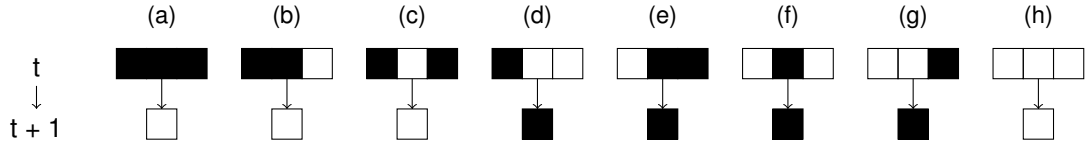
Podstawy teoretyczne

3.1 Automat komórkowy

Automat komórkowy [3] to model składający się ze zbioru komórek na siatce. Każda komórka ma określony stan. Liczba stanów jest całkowita, nieujemna i skończona. Komórki ewoluują równolegle, wedle ustalonych reguł. Reguły określają jaki wpływ na ewolucje komórki ma jej obecny stan oraz stan jej sąsiadów.

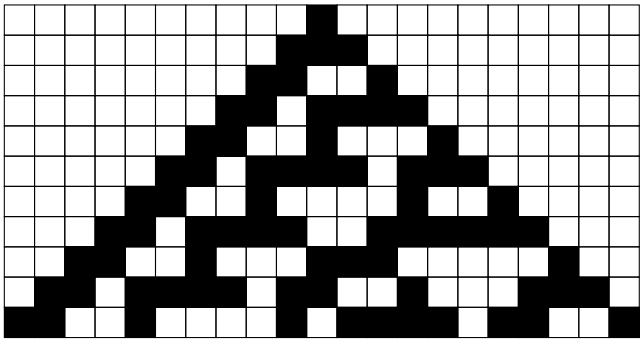
Pojęcie automatu komórkowego zostało po raz pierwszy wprowadzone przez Johna von Neumanna [4], do matematycznego opisu biologicznego zjawiska samo-replikacji. Od tamtego czasu automaty komórkowe zyskały dużą popularność, zostały rozwinięte i aktualnie za ich pomocą można przybliżyć i zrozumieć wiele złożonych zjawisk fizycznych (np. symulacja płynów), chemicznych (np. symulacja reakcji Belousov-Zhabotinsky), społecznych (np. symulacja rozwoju miast) i biologicznych (np. symulacja rozwoju nowotworów).

Najprostszy typ automatu komórkowego jest jednowymiarowym automatem (wysokość siatki jest równa 1), przyjmująca wartości binarne, w którym sąsiedztwo komórki stanowią najbliżsi sąsiedzi (sama komórka oraz jej prawy i lewy sąsiad). Istnieje 256 takich automatów [5], każdy ma przypisany numer, który jest znany jako reguła. Reguła mówi nam jaki wynik da dane sąsiedztwo. I tak na przykład w regule 30, jeżeli w kroku t weźmiemy daną trójkę bitów, z pozycji x_1, x_2, x_3, posiadające wartości $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, to w kroku $t + 1$ w pozycji x_1 otrzymamy bit o wartości 0 (patrz rys. 3.1a). Natomiast, jeżeli w kroku t, w pozycjach x_1, x_2, x_3 znajdą się wartości 0, 1, 1, to w kroku $t + 1$, w pozycji x_1 uzyskamy bit o wartości 1 (patrz rys. 3.1e).



Rysunek 3.1: Reguła 30. Białe pola reprezentują wartość binarną 0. Czarne pola reprezentują wartość binarną 1.

Na rys. 3.2 można zobaczyć jak wygląda kilkanaście kroków automatu komórkowego opisanego regułą 30. Na tym rysunku każda linia stanowi jeden krok. Pierwsza linia od góry to pierwszy krok, każda następna w dół to kolejny krok. Bity z wartością 0 przyjmują wygląd białego kwadratu, a bity z wartością 1, czarnego kwadratu. Dzięki tej regule można zaobserwować, jak proste zasady mogą produkować złożone struktury.



Rysunek 3.2: Jedenaście kroków automatu opisanego regułą 30.

3.2 Model Nagela–Schreckenberga

Model Nagela–Schreckenberga [6] został stworzony przez niemieckich fizyków Kaia Nagela i Michaela Schreckenberga. Jest on oparty na automatach komórkowych i składa się z jednowymiarowej siatki komórek, która reprezentuje jednokierunkową ulicę. Każda komórka może być zajęta przez samochód bądź pusta. Warunki brzegowe takiego modelu mogą być otwarte lub periodyczne. Każdy pojazd ma prędkość v przedstawioną za pomocą liczby całkowitej, która jest z zakresu 0 do v_{max}. Reguły opisujące model, które są wykonywane równoległe dla każdego samochodu, przyjmują następującą formę:

1. Pierwszą czynnością jest przyśpieszenie. Polega ono na dodaniu do prędkości v wartości jeden, uważając na to, by prędkość nie przekroczyła wartości maksymalnej v_{max}. Przyśpieszenie przybiera następującą formę:
$$v = min(v + 1, v_{max}),$$
gdzie funkcja min zwraca mniejszy z dwóch argumentów.
2. Następną procedurą jest hamowanie. Sprowadza się ono do sprawdzenia, czy przed danym pojazdem, w przestrzeni od pozycji pojazdu x do miejsca, w którym by się on znalazł po dodaniu do jego pozycji prędkości v nie znajduje się żaden samochód. Jeżeli pojawia się jakiś, to należy zmniejszyć prędkość do liczby komórek d między tymi pojazdami. Hamowanie przybiera następującą formę:
$$v = min(v, d).$$
3. Losowe spowolnienie z prawdopodobieństwem p polega na wylosowaniu liczby zmiennoprzecinkowej z zakresu od 0 do 1 i porównaniu jej z p (ustalona stałą). Jeżeli wylosowana liczba jest mniejsza od p, to należy od prędkości v odjąć jeden, uważając na to, by prędkość nie przyjęła

wartości ujemnej. Losowe spowolnienie z prawdopodobieństwem przybiera następującą formę:

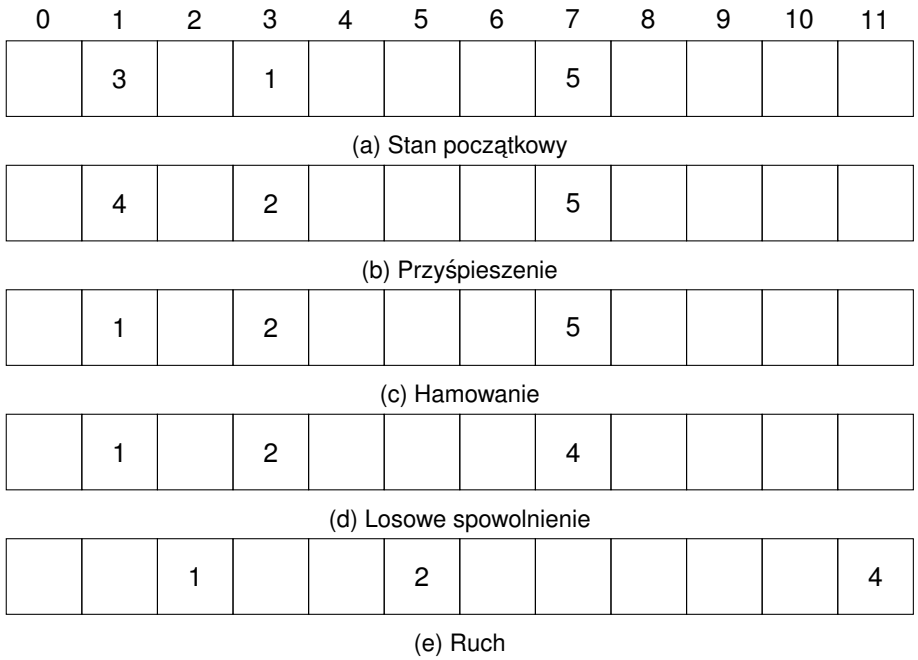
$$v = max(v - 1, 0),$$

gdzie funkcja max zwraca większy z dwóch argumentów.

4. Ostatnią czynnością jest ruch. Przemieszcza się pojazd ustalając jego nowe położenie. Do pozycji pojazdu x należy dodać wyliczoną wcześniej prędkość v. Ruch przybiera następującą formę:

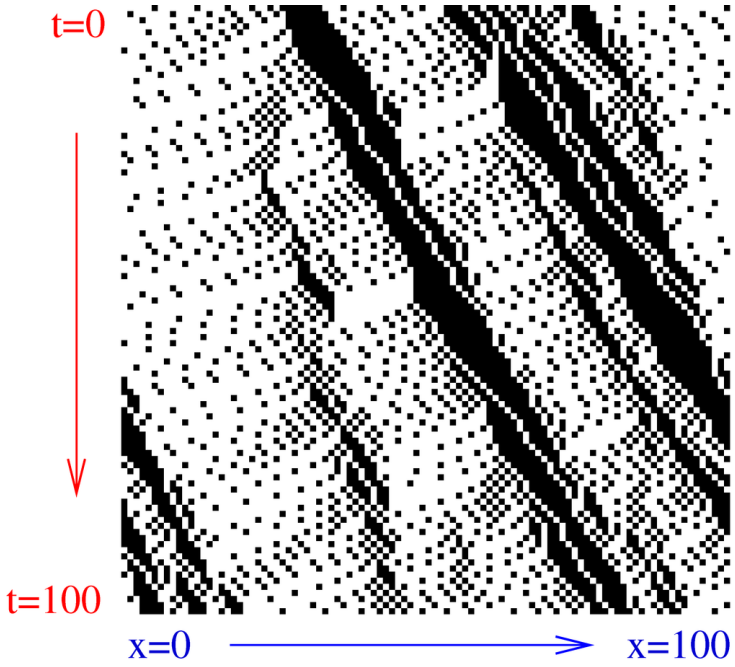
$$x = x + v.$$

Przykładowy krok modelu Nagela–Schreckenberga mógłby wyglądać następująco (rys. 3.3). Załóżmy, że droga ma wielkość 12 komórek i znajdują się na niej 3 pojazdy o pozycjach $x_1 = 1$, $x_2 = 3$, $x_3 = 7$ (rys. 3.3a). Prędkości pojazdów są równe $v_1 = 3$, $v_2 = 1$ i $v_3 = 5$. Przyjmijmy, że maksymalna prędkość v_{max} jest równa 5. Po przyśpieszeniu (rys. 3.3b), dwóch pierwszych samochodów, ich prędkość wzrośnie o 1. Prędkość trzeciego samochodu nie zmieni się, ponieważ nie może przekroczyć prędkości maksymalnej: $v_3 = min(v_3, v_{max}) = 5$. Po kroku przyśpieszenia prędkości samochodów są równe: $v_1 = 4$, $v_2 = 2$, $v_3 = 5$. Podczas czynności hamowania (rys. 3.3c) tylko pierwszy samochód musi zredukować prędkość, gdyż liczba pustych komórek przed nim $d = 1$ jest mniejsza od jego prędkości $v_1 = 4$. Pierwszy samochód będzie miał więc prędkość równą liczbie pustych komórek przed nim $v_1 = 1$, zaś prędkość pozostałych samochodów nie ulegnie zmiany: $v_2 = 2$, $v_3 = 5$. W kroku losowego spowalniania (rys. 3.3d) załóżmy, że trzeci samochód wylosował liczbę mniejszą od p, więc musi zredukować prędkość o 1: $v_3 = 4$. Ostatnią procedurą jest ruch (rys. 3.3e). Każdy pojazd zmienia swoją pozycję zgodnie z tym krokiem. Pojazd na pozycji $x_1 = 0$, z prędkością $v_1 = 1$ znajdzie się na pozycji $x_1 = 1$. Kolejne pojazdy poruszą się analogicznie: $x_2 = x_2 + v_2 = 1 + 1 = 2$, $x_3 = x_3 + v_3 = 7 + 4 = 11$.



Rysunek 3.3: Jeden krok modelu Nagela–Schreckenberga, na który składają się 4 etapy(a-d).

Przykład na Rys. 3.4 ukazuje powstawanie zatorów na drodze opisanej modelem Nagela–Schreckenberga. Każda linia zawiera 100 komórek. Czarne kwadraty reprezentują samochody, natomiast białe puste komórki. Pierwsza linia od góry obrazuje model w czasie t=0. Następna pod nią odpowiada czasowi t=1, ostania zaś czasowi t=100. Granice są periodyczne. Zagęszczenie samochodów jest równe 0.35, zaś parametr losowego spowalniania p = 0.3.



Rysunek 3.4: Model Nagela–Schreckenberga.

Ten prosty model oddaje realistyczne zachowanie pojazdów w ruchu ulicznym. Wyniki uzyskane używając modelu Nagela–Schreckenberga zostały porównane z danymi zebranymi w rzeczywistości na jednej z autostrad i okazały się bardzo zbliżone. Krok 3 jest niezbędny, gdyż wprowadza on wahania prędkości spowodowane ludzkim zachowaniem oraz zmieniającymi się czynnikami zewnętrznymi. Bez niego symulacja byłaby deterministyczna. Model NaSch pomógł wyjaśnić jak powstają zatory na drodze bez żadnych wpływów zewnętrznych.

3.3 Model VDR

Model Velocity-Dependent-Randomization [7] (losowość zależna od prędkości) jest rozszerzeniem modelu Nagela–Schreckenberga. W stosunku do modelu NaSch, algorytm VDR dodaje jeden krok przed wszystkimi czynnościami modelu, na którym bazuje: określeniem parametru losowego spowalniania (krok 0). Krok ten polega na sprawdzeniu, czy pojazd ma prędkość większą od zera. Na podstawie tego, dobierany jest parametr p, który będzie użyty w procedurze losowego spowolnienia z prawdopodobieństwem p (opisany w roz. 3.2 jako krok 3 modelu Nagela–Schreckenberga). Jeżeli prędkość v samochodu jest równa zeru, to parametr $p = p_0$. Natomiast gdy prędkość v jest większa

od zera, to $p = p_1$. Stosując model VDR, należy zatem reguły modelu NaSch (Roz. 3.2, kroki od 1 do 4) uzupełnić o krok:

0. Określenie parametru losowego spowalniania:

$$p = \begin{cases} p_0, & \text{gdy } v = 0, \\ p_1, & \text{gdy } v > 0. \end{cases}$$

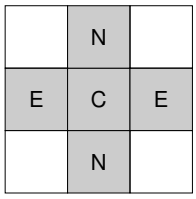
Parametry p_0 i p_1 to wartości stałe oraz $p_0 > p_1$. Dzięki tej nowej regule, jeżeli samochód zatrzyma się całkowicie, istnieje większe prawdopodobieństwo, że dłużej zajmie mu ponowne wystartowanie. Ta nowa zasada wprowadza do symulacji tendencję ludzką do rozpraszania się podczas stania w korku i co za tym idzie ruszania z opóźnieniem względem samochodu poprzedzającego.

Rozdział 4

Opracowany model

W ramach realizowanego tematu pracy opracowano model będący rozwinięciem modelu VDR, co za tym idzie również modelu Nagela–Schreckenberga. Składa się z dwuwymiarowej siatki komórek, na której są umieszczone jednokierunkowe ulice oraz pojazdy. Każdej komórce jest przypisana jedna z czterech stron świata, określająca kierunek drogi. Jeżeli dochodzi do krzyżowania się ulic, to komórka, w której następuje to zdarzenie, zostaje oznaczona jako skrzyżowanie. Na skrzyżowaniach panuje ruch równorzędny. Komórka może również zostać pusta (nie mieć przypisanego kierunku ani nie być skrzyżowaniem). Komórki puste obrazują obszar niebędący drogą, na który nie można wjechać. Granice w modelu są periodyczne. Wszystkie pojazdy mają swoją pozycję $\vec{r} = [r_x, r_y]$, prędkość $\vec{v} = [v_x, v_y]$, parametr losowego spowalniania p (potrzebny przy kroku losowego spowalniania) oraz kierunek, w którym aktualnie podążają $\overrightarrow{dir} = [dir_x, dir_y]$. Wartość $\overrightarrow{dir} = [0, 1]$ oznacza kierunek północny, $\overrightarrow{dir} = [1, 0]$ wschodni, $\overrightarrow{dir} = [0, -1]$ południowy, a $\overrightarrow{dir} = [-1, 0]$ zachodni.

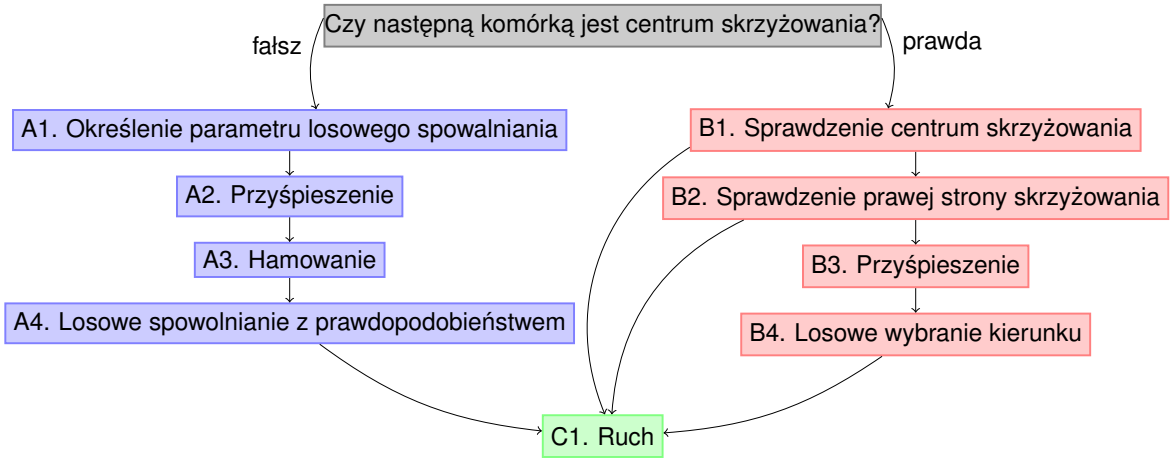
Na potrzeby opisu przyjmijmy, że centrum skrzyżowania to komórka znajdująca się pośrodku czterech komórek będących ulicami (rys. 4.1 komórka oznaczona literą C).



Rysunek 4.1: Skrzyżowanie.

Ogólny zarys reguł stosowanych w opracowanym modelu przybiera następującą formę (Rys. 4.2): na początku zostaje sprawdzone czy komórka przed aktualizowanym pojazdem jest centrum skrzyżowania. Jeżeli nie jest, to należy określić parametr losowego spowalniania. Następnie wykonać przyśpieszenie, hamowanie, losowe spowalnianie z prawdopodobieństwem według wcześniej określonego parametru i na końcu ruch. W drugim przypadku komórka przed aktualizowanym pojazdem jest centrum skrzyżowania. W tym wypadku najpierw należy sprawdzić, czy w centrum skrzyżowania znajduje się inny pojazd. Jeżeli tak, to należy przejść do czynności ruchu. Natomiast gdy w centrum skrzyżowania nie ma innego pojazdu, należy sprawdzić czy po prawej stronie skrzyżowa-

nia względem aktualizowanego samochodu nie ma innego pojazdu. Jeżeli tak, to należy przejść do czynności ruchu. Jeżeli po prawej stronie nie ma innego pojazdu to aktualizowany pojazd wykonuje przyśpieszenie, losowo wybiera następny kierunek, a na końcu wykonuje ruch.



Rysunek 4.2: Schemat reguł stosowanych w opracowanym modelu.

Dla każdego pojazdu krok aktualizacji modelu zaczyna się od sprawdzenia, czy komórka przed pojazdem jest centrum skrzyżowania. Jeżeli nie jest, to przebieg symulacji przyjmuje reguły bazujące na modelu VDR:

A1. Na początku należy określić parametr losowego spowalniania p. Jeżeli obie składowe prędkości v_x i v_y samochodu są równe zeru to parametr $p = p_0$. Natomiast gdy któraś ze składowych prędkości v_x lub v_y jest różna od zera, to $p = p_1$. Określenie parametru losowego spowalniania przybiera następującą formę:

$$p = \begin{cases} p_0, & \text{gdy } v_x = 0 \wedge v_y = 0, \\ p_1, & \text{gdy } v_x \neq 0 \vee v_y \neq 0, \end{cases}$$

A2. Operacja przyśpieszenia polega na:

1. Dodaniu do prędkości $\vec{v}$ aktualnego kierunku $\overrightarrow{dir}$. Sprawi to, że prędkość zostanie zwiększona o 1, zgodnie z kierunkiem drogi.
2. Sprawdzeniu, czy prędkość $\vec{v}$ nie przyjęła wartości wykraczającej poza wartość maksymalną v_{max}. Odpowiedzialna jest za to funkcja $clamp(\vec{v}, v_{max})$, która sprawia, że żadna ze składowych wektora $\vec{v}$ nie przyjmie wartości większych niż v_{max} i mniejszych niż $-v_{max}$.

Przyśpieszenie przybiera następującą formę:

$$\vec{v} = clamp(\vec{v} + \overrightarrow{dir}, v_{max}).$$

A3. Kolejnym krokiem jest hamowanie. Sprowadza się ono do sprawdzenia, czy przed danym pojazdem, w przestrzeni od pozycji pojazdu $\vec{r}$ do miejsca, w którym by się on znalazł po dodaniu do jego pozycji prędkości $\vec{v}$ nie znajduje się żaden samochód lub centrum skrzyżowania. Jeżeli pojawia się któreś z poprzednio wymienionych, to należy albo zmniejszyć prędkość do liczby komórek między tymi pojazdami d albo do liczby komórek między aktualizowanym samochodem

a centrum skrzyżowania c. Należy wybrać mniejszą z liczb d i c co można uzyskać przy użyciu $min(d, c)$. Hamowanie przybiera następującą formę:

$$\vec{v} = clamp(\vec{v}, min(d, c)).$$

A4. Następnie należy wykonać losowe spowolnienie z prawdopodobieństwem. Polega ono na wylosowaniu liczby zmiennoprzecinkowej z zakresu od 0 do 1 i sprawdzeniu, czy dana liczba jest mniejsza od parametru losowego spowalniania p pojazdu. Jeżeli wylosowana liczba jest mniejsza od p oraz któraś ze składowych wektora $\vec{v}$ jest różna od zera, to należy od prędkości $\vec{v}$ odjąć kierunek $\overrightarrow{dir}$. Spowoduje to, że jedna ze składowych prędkości v_x lub v_y zmniejszy się o jeden. Losowe spowolnienie z prawdopodobieństwem przybiera następującą formę:

$$\vec{v} = \begin{cases} \vec{v}, & \text{gdy } v_x = 0 \wedge v_y = 0, \\ \vec{v} - \overrightarrow{dir}, & \text{gdy } v_x \neq 0 \vee v_y \neq 0, \end{cases}$$

Z kolei, gdy komórka przed pojazdem jest centrum skrzyżowania, to reguły przybierają następującą formę:

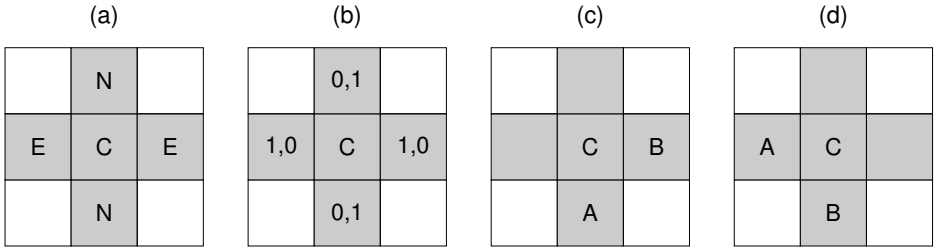
B1. Pierwszą czynnością jest sprawdzenia centrum skrzyżowania. Należy stwierdzić czy komórkę będącą centrum skrzyżowania (która znajduje się przed aktualizowanym samochodem), zajmuje jakiś pojazd. Aby się tego dowiedzieć należy ustawić pozycję testową $\overrightarrow{test} = [test_x, test_y]$ na centrum skrzyżowania. Testowa pozycja jest więc sumą pozycji aktualizowanego pojazdu $\vec{r}$ z kierunkiem aktualizowanego pojazdu $\overrightarrow{dir}$:

$$\overrightarrow{test} = \vec{r} + \overrightarrow{dir}.$$

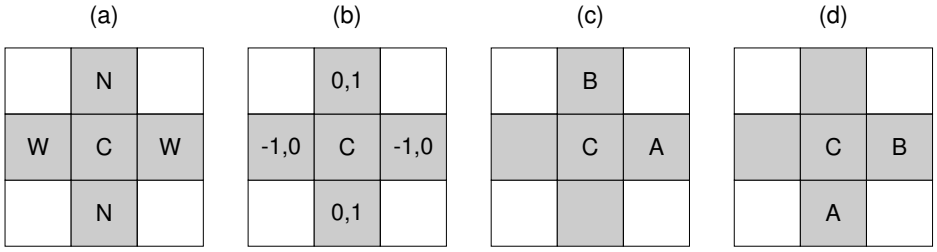
Jeżeli w testowej pozycji znajduje się jakiś samochód, to aktualizowany pojazd ustawia prędkość na zero $\vec{v} = [0, 0]$ i przechodzi do kroku ruchu (Krok oznaczony symbolem C1. Opisano go w dalszej części pracy). Uniemożliwi to przemieszczenie aktualizowanego pojazdu.

B2. Następną procedurą jest sprawdzenie prawej strony skrzyżowania. Ponieważ każde skrzyżowanie jest równorzędne, należy sprawdzić, czy komórka po prawej stronie skrzyżowania względem pozycji samochodu jest zajęta przez inny pojazd. Istnieją cztery rodzaje skrzyżowań: droga północna ze wschodnią (N-E), droga północna z zachodnią (N-W), droga południowa ze wschodnią (S-E) oraz droga południowa z zachodnią (S-W). Rysunki od 4.3 do 4.6 obrazują każde możliwe skrzyżowanie. Na rysunkach podpunkt a to opis symboliczny skrzyżowania. Podpunkt b to opis wektorowy skrzyżowania. Pokazują one kierunek każdej z dróg oraz gdzie się znajduję centrum skrzyżowania. W podpunktach c oraz d literą A oznaczono początkowe miejsca samochodu, natomiast literą B prawą stronę skrzyżowania względem początkowego miejsca samochodu. Podpunkt c obrazuje sytuacje w której nie trzeba sprawdzać prawej strony skrzyżowania. Ponieważ droga w pozycji A kieruje się do centrum skrzyżowania, a B ma kierunek przeciwny do centrum skrzyżowania. Podpunkt d obrazuje sytuacje w której trzeba sprawdzić prawą stronę skrzyżowania. Ponieważ droga w pozycji A oraz droga w pozycji B kierują się do centrum skrzyżowania. Na przykład, podczas krzyżowania się drogi północnej z drogą wschodnią, gdy samochód znajduje się na drodze o kierunku północnym (Rys. 4.3c) w pozycji A. Nie trzeba sprawdzać czy inny pojazd jest po prawej stronie skrzyżowania w pozycji B. Ponieważ droga w pozycji B ma kierunek wschodni, a więc samochód który znalazł by się w tym miejscu

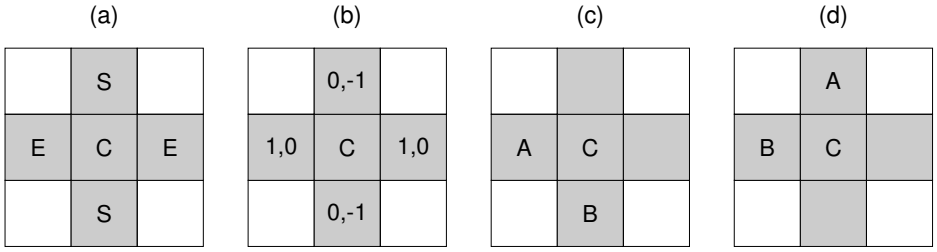
nie kierował by się w stronę skrzyżowania. W sytuacji gdy droga północna krzyżuje się z drogą wschodnią, a samochód znajduje się na drodze o kierunku wschodnim (Rys. 4.3d) w pozycji A. Trzeba sprawdzać czy inny pojazd jest po prawej stronie skrzyżowania w pozycji B. Ponieważ droga w pozycji B ma kierunek północny, a więc samochód który znalazł by się w tym miejscu kierował by się w stronę skrzyżowania.



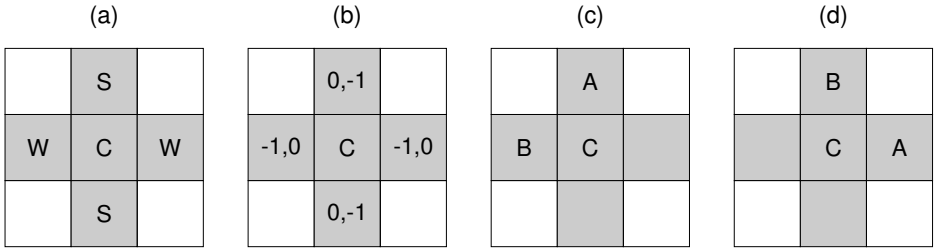
Rysunek 4.3: Skrzyżowanie drogi północnej ze wschodnią.



Rysunek 4.4: Skrzyżowanie drogi północnej z zachodnią.



Rysunek 4.5: Skrzyżowanie drogi południowej ze wschodnią.



Rysunek 4.6: Skrzyżowanie drogi południowej z zachodnią.

Aby dowiedzieć się, czy komórka po prawej stronie skrzyżowania względem pozycji samochodu, jest zajęta przez inny pojazd, należy:

1. Ustawić pozycję testową $\overrightarrow{test}$ na centrum skrzyżowania:
$$\overrightarrow{test} = \vec{r} + \overrightarrow{dir}.$$
2. Przesunąć pozycję testową na prawą stronę skrzyżowania względem aktualizowanego pojazdu. Jeżeli droga, na której znajduje się aktualizowany pojazd, ma kierunek północny $dir = [0, 1]$ lub południowy $dir = [0, -1]$. to aby przenieść pozycje testową, należy wykonać: $test_x = dir_y$ (patrz tab. 4.1). W przeciwnym wypadku (droga, na której znajduje się aktualizowany pojazd, ma kierunek zachodni $dir = [1, 0]$ lub wschodni $dir = [-1, 0]$) i należy wykonać: $test_y = -dir_x$ (patrz tab. 4.1). Przesunięcie pozycji testowej na prawą stronę skrzyżowania względem aktualizowanego pojazdu przybiera następującą formę:
$$\overrightarrow{test} = \begin{cases} \overrightarrow{test} + [dir_y, 0], & \text{gdy } dir = [0, 1] \vee dir = [0, -1], \\ \overrightarrow{test} - [0, dir_x], & \text{gdy } dir = [1, 0] \vee dir = [-1, 0], \end{cases}$$

Kierunek aktualnej drogi	N	S	W	E
Wektor kierunku aktualnej drogi	(0,1)	(0,-1)	(-1,0)	(1,0)
Pozycja przed skrzyżowaniem	(0,0)	(0,0)	(0,0)	(0,0)
Centrum skrzyżowania	(0,1)	(0,-1)	(-1,0)	(1,0)
Prawa strona skrzyżowania	(1,1)	(-1,-1)	(-1,1)	(1,-1)

Tablica 4.1: Przejście od pozycji przed skrzyżowaniem do jej prawej strony.

3. Następnie należy policzyć sumę kontrolną $s = sum(\overrightarrow{dir}, direction(\overrightarrow{test}))$. Gdzie funkcja direction zwraca wektor $\overrightarrow{dir\_test}$ będący kierunkiem ulicy w pozycji $\overrightarrow{test}$. A funkcja sum zwraca wartość równą $dir_x + dir\_test_x + dir_y + dir\_test_y$. Jeżeli kierunek pojazdu jest północny $dir = [0, 1]$ lub południowy $dir = [0, -1]$ i suma kontrolna jest równa zeru $s = 0$ albo kierunek pojazdu jest wschodni $dir = [-1, 0]$ lub zachodni $dir = [1, 0]$ i suma kontrolna jest różna od zera $s \neq 0$, to droga prostopadła do kierunku pojazdu jest równorzędna. Należy wtedy sprawdzić, czy w aktualnej pozycji testowej $\overrightarrow{test}$ znajduje się jakiś samochód (patrz tab. 4.2). Jeżeli tak, to aktualizowany pojazd ustawia prędkość na zero $\vec{v} = [0, 0]$ i przechodzi do kroku ruchu.

Kierunek aktualnej drogi	N		S		W		E	
Wektor kierunku aktualnej drogi	(0,1)		(0,-1)		(-1,0)		(1,0)	
Kierunek drogi po prawej	W	E	W	E	N	S	N	S
Wektor kierunku drogi po prawej	(-1,0)	(1,0)	(-1,0)	(1,0)	(0,1)	(0,-1)	(0,1)	(0,-1)
Suma kontrolna	0	2	-2	0	0	-2	2	0
Droga po prawej jest równorzędna	P	F	F	P	F	P	P	F

Tablica 4.2: Suma kontolna.

B3. Kolejną czynnością jest przyśpieszenie. Prędkość zostaje przyrównana do kierunku aktualnej drogi $\vec{v} = \overrightarrow{dir}$. Oznacza to, że aktualizowany pojazd w kroku ruchu, przemieści się na centrum

skrzyżowania.

B4. Ostatnią procedurą jest losowe wybranie kierunku. Zostaje wybrane losowo, czy od następnego kroku pojazd ma kierować się dalej tą samą drogą $\overrightarrow{dir} = \overrightarrow{dir}$, czy powinien wybrać drogę prostopadłą $\overrightarrow{dir} = \overrightarrow{testDir}$.

Niezależnie od tego, czy pojazd znajduje się komórkę przed centrum skrzyżowania, czy też nie, na koniec wykonuje:

C1. Ruch. Przemieszcza się pojazd ustalając jego nowe położenie. Do pozycji pojazdu $\vec{r}$ należy dodać wyliczoną wcześniej prędkość $\vec{v}$. Ruch przybiera następującą formę:

$$\vec{r} = \vec{r} + \vec{v}.$$

Rozdział 5

Implementacja

5.1 Symulacja

Na potrzeby implementacji opracowanego modelu zostały stworzone trzy podstawowe klasy:

1. `Map` (Mapa)
2. `Car` (Samochód)
3. `Simulation` (Symulacja)

5.1.1 Klasa Map

```
class Map {
    int *cellType;
    std::vector<Car *> cellState;

public:
    static const int Empty = 0;
    static const int North = 1;
    static const int East = 2;
    static const int South = 3;
    static const int West = 4;
    static const int Crossing = 5;

    int cellSize;
    int width;
    int height;
    int numberOfStreetTiles;

    Map(int width, int height, int tileSize, int horizontalStreets, int
        verticalStreets);
    ~Map();
    void init(int width, int height, int horizontalStreets, int verticalStreets);
    void render(sf::RenderTexture &renderTexture);

    int getCellType(int x, int y) const;
    Cell * getCellState(int x, int y);
    void pushCarToCell(Car &car);
    void clearCarFromCell(Car &car);
    static sf::Vector2i directionToVector(int dir);
};
```

Rysunek 5.1: Nagłówek klasy Map.

Obiekt klasy `Map` jest odpowiedzialny za przechowywanie danych dotyczących stanu i typu komórek, oraz parametrów mapy (Rys. 5.1). Wśród pól opisujących mapę znajduje się:

1. Tablica liczb całkowitych `cellsType`. Do tablicy zostają wpisane liczby od 0 do 5. Każda liczba opisuje, typ danej komórki. 0 oznacza pustą komórkę (Empty), 1 to droga w kierunku północnym (North), 2 to droga w kierunku wschodnim (East), 3 to droga w kierunku południowym (South), 4 to droga w kierunku zachodnim (West), natomiast 5 reprezentuje skrzyżowanie (Crossing). Przykładowa tablica `cellsType` mogłaby wyglądać jak na Rys. 5.2. Mapa zawiera cztery drogi o różnych kierunkach (pola o wartościach 1, 2, 3 i 4), krzyżujące się tworząc cztery skrzyżowania (pola o wartościach 5). Na mapie znajduje się dziewięć pustych, prostokątnych obszarów (pola o wartościach 0).

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	0	0	0	0	3	0	0
1	0	0	1	0	0	0	0	3	0	0
2	4	4	5	4	4	4	4	5	4	4
3	0	0	1	0	0	0	0	3	0	0
4	0	0	1	0	0	0	0	3	0	0
5	0	0	1	0	0	0	0	3	0	0
6	0	0	1	0	0	0	0	3	0	0
7	2	2	5	2	2	2	2	5	2	2
8	0	0	1	0	0	0	0	3	0	0
9	0	0	1	0	0	0	0	3	0	0

Rysunek 5.2: Przykładowe rozmieszczenie dróg na mapie.

2. Lista wskaźników `cellsState` opisujących stan każdej z komórek. Gdy wskaźnik jest pusty $ptr = null$, oznacza to, że w danej komórce nie znajduję się żaden pojazd. Natomiast gdy wskaźnik nie jest pusty $ptr \neq null$, oznacza to, że w tej komórce znajduje się samochód.
3. Liczby całkowite `width` i `height` opisujące liczbę komórek w poziomie i pionie. Wielkości te są równoznaczne z długością i wysokością mapy.
4. Liczba całkowita `cellSize` opisującą rozmiar komórki w pikselach, potrzebną do wizualizacji.
5. Liczba całkowita `numberOfStreetTiles` opisującą liczbę komórek zajmowanych przez ulice i skrzyżowania.

Obiekt klasy `Map` udostępnia też funkcje:

1. `init` wypełnia tablicę `cellsType` tak by znalazło się w niej, tyle ulic ile podanych jest w argumentach oraz żeby odstępy między ulicami były równe.
2. `render` zostanie opisana w rozdziale 5.3.
3. `getCellType` zwraca typ komórki z miejsca podanego w argumentach funkcji.
4. `getCellState` zwraca stan komórki z miejsca podanego w argumentach funkcji.
5. `pushCarToCell` ustawia stan komórki zgodnej z pozycją samochodu na dany w argumencie samochód.
6. `clearCarFromCell` ustawia stan komórki na pusty zgodnie z pozycją samochodu podanego w argumencie funkcji.
7. `directionToVector` zmienia wartość reprezentującą kierunek drogi na wektor kierunku drogi (1 (North) na [1,0], 2 (East) na [0,1], 3 (South) na [-1,0], 4 (West) na [0,-1]).

5.1.2 Klasa Car

```
class Car {
public:
    sf::Vector2i position;
    sf::Vector2i oldPosition;
    sf::Vector2i velocity;
    int target;
    float propability;

    Car(int x, int y, int target);
};
```

Rysunek 5.3: Nagłówek klasy Car.

Obiekt klasy `Car` zawiera tylko dane (Rys. 5.3). Wśród pól opisujących samochód znajdują się:

1. Wektor `position` opisujący pozycje pojazdu.
2. Wektor `oldPosition` opisujący poprzednią pozycje pojazdu. Jest on potrzebny przy wizualizacji.
3. Wektor `velocity` opisujący prędkość pojazdu.
4. Liczba całkowita `target` opisująca kierunek samochodu w postaci zgodnej z reprezentacją ulic w klasie Map.
5. Liczba zmiennoprzecinkowa `propability` opisująca parametr losowego spowalniania (potrzebny przy kroku losowego spowalniania).

5.1.3 Klasa Simulation

```
class Simulation {
    std::vector <Car> cars;
public:
    Random random;
    Map &map;
    const int vMax;
    const float propability0;
    const float propability1;

    Simulation(Map &map, int vMax, float propability0, float propability1, int
        carsDensity);
    void init(int carsDensity);
    void update();
    void straightStreetUpdate(Car &car);
    void nearCrossingUpdate(Car &car);
    void motion(Car &car);
    void render(sf::RenderTexture &renderTexture);
    bool collision(sf::Vector2i position);
    bool isNearCrossing(Car &car);
    void clamp(sf::Vector2i &velocity, int bound);
    void clampPosition(sf::Vector2i &position);
    void clampPosition(sf::Vector2f &position);
};
```

Rysunek 5.4: Nagłówek klasy Simulation.

W klasie `Simulation` została zawarta cała logika i reguły modelu (Rys. 5.4). Wśród pól opisujących symulację znajduje się:

1. Lista `cars` opisująca wszystkie samochody znajdujące się w symulacji.
2. Generator liczb pseudolosowych `random` .
3. Wskaźnik na obiekt klasy `map`.
4. Liczba całkowita `vMax` opisująca maksymalną prędkość pojazdów.
5. Liczby zmiennoprzecinkowe `p0` i `p1` potrzebne w kroku określenia parametru losowego spowalniania.

Obiekt klasy `Simulation` udostępnia też funkcje:

1. `init` rozmieszcza losowo samochody na mapie. Pojazd może zostać umieszczony tylko na komórce będącej drogą, na której nie ma jeszcze żadnego samochodu.
2. `update` (Rys. 5.5) odpowiada za zmianę stanu pojazdów zgodnie z ustalonymi regułami. W tej metodzie jest ustalane, czy każdy samochód jest komórkę przed centrum skrzyżowania (za pomocą funkcji `isNearCrossing`) i powinien zmienić stan zgodnie z regułami dla samochodu będącego przed centrum skrzyżowania (`nearCrossingUpdate`) lub czy musi zmienić stan zgodnie z regułami dla pojazdu będącego oddalonym od skrzyżowania (`straightStreetUpdate`). Następnie wykonywana jest funkcja ruchu (`motion`) na rzecz każdego samochodu.

```
void Simulation::update() {
    for (int i = 0; i < cars.size(); ++i) {
        if (isNearCrossing(cars[i]))
            nearCrossingUpdate(cars[i]);
        else
            straightStreetUpdate(cars[i]);
    }
    for (int i = 0; i < cars.size(); ++i) {
        motion(cars[i]);
    }
}
```

Rysunek 5.5: Kod odpowiedzialny za wybór wedle jakich reguł pojazd powinien zmienić stan.

3. `straightStreetUpdate` to implementacja kroków A1-A4 z rozdziału 4.

```
void Simulation::straightStreetUpdate(Car &car) {
    sf::Vector2i direction = Map::directionToVector(car.target);

    //zero one value
    car.velocity.x *= std::abs(direction.x);
    car.velocity.y *= std::abs(direction.y);

    //Rule #0 (Determination of the randomization parameter)
    if(car.velocity.x == 0 && car.velocity.y == 0)
        car.propability = propability0;
    else
        car.propability = propability1;

    //Rule #1 (Acceleration)
    car.velocity += direction;
    clamp(car.velocity, vMax);

    //Rule #2 (Deceleration)
    sf::Vector2i testPosition(car.position);
    int dist = 0;
    for (int i = 0; i < std::abs(car.velocity.x + car.velocity.y); i++) {
        testPosition += direction;
        clampPosition(testPosition);

        if (!collision(testPosition) && map.getTile(testPosition) != Map::Crossing)
            dist++;
        else
            break;
    }
    clamp(car.velocity, dist);

    //Rule #3 (Randomization)
    if (random.randomFloat() < car.propability){
        if(car.velocity.x != 0 && car.velocity.y != 0)
            car.velocity -= direction;
    }
}
```

Rysunek 5.6: Kod odpowiedzialny za zmianę stanu pojazdu zgodnie z regułami poza skrzyżowaniem.

4. `nearCrossingUpdate` to implementacja kroków B1-B4 z rozdziału 4.

```
void Simulation::nearCrossingUpdate(Car &car) {
    car.velocity.x = 0;
    car.velocity.y = 0;

    int roadDir = map.getTile(car.position);
    sf::Vector2i roadDirection = Map::directionToVector(roadDir);

    // set testPosition on the middle of crossing
    sf::Vector2i testPosition(car.position);
    testPosition += roadDirection;

    // check collision on crossing
    if (collision(testPosition))
        return;

    // set testPosition on right side of crossing
    if (roadDir == Map::North || roadDir == Map::South)
        testPosition.x += roadDirection.y;
    else
        testPosition.y -= roadDirection.x;

    int nextRoadDir = map.getTile(testPosition);
    sf::Vector2i nextRoadDirection = Map::directionToVector(nextRoadDir);

    sf::Vector2i tmp = roadDirection + nextRoadDirection;
    int sum = tmp.x + tmp.y;

    // if current road direction is N or S and control sum is equal to 0
    // or       -||-                 E or W      -||-       is not equal to 0
    // then we have to check next road right side for collision
    if (((roadDir == Map::North || roadDir == Map::South) && sum == 0)
        || ((roadDir == Map::East || roadDir == Map::West) && sum != 0)) {
        if (collision(testPosition))
            return;
    }

    car.velocity += roadDirection;

    if (random.randomFloat() <= 0.5) {
        car.target = roadDir;
    }
    else {
        car.target = nextRoadDir;
    }
}
```

Rysunek 5.7: Kod odpowiedzialny za zmianę stanu pojazdu zgodnie z regułami przed skrzyżowaniem.

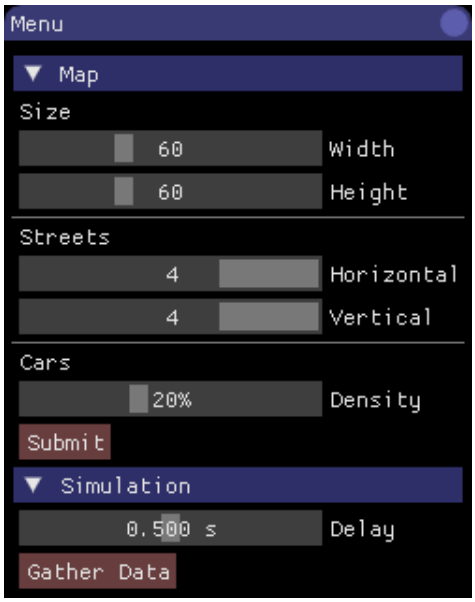
5. `motion` na początku tej funkcji samochód zostaje usunięty z komórki, w której aktualnie się znajduje za pomocą funkcji `clearCarFromCell` z klasy `Map`. Następnie wykonuje ruch opisany w rozdziale 4 w punkcie C1. Na końcu umieszczany jest w komórce, w której aktualnie się znajduje za pomocą funkcji `pushCarToCell` z klasy `Map`.
6. `render` zostanie opisana w rozdziale 5.3
7. `colliosion` sprawdza, czy w pozycji danej w argumentach nie znajduje się jakiś samochód.
8. `isNearCrossing` sprawdza, czy komórka przed pojazdem nie jest centrum skrzyżowania.
9. `clamp` sprawia, że parametry wektora będącego pierwszym argumentem nie przyjmą wartości większych niż wartość drugiego argumentu i mniejszych niż ujemna wartość drugiego argumentu.
10. `clampPosition` sprawia, że pozycja będąca argumentem funkcji nie wyjdzie poza mapę.

5.2 Graficzny interfejs użytkownika

Interfejs użytkownika to okno `Scene` zawierające wizualizacje symulacji oraz okno `Menu`, które udostępnia elementy służące do manipulacji ustawień symulacji.

5.2.1 Okno Menu

Okno `Menu` (Rys. 5.8) jest podzielone na dwie sekcje: `Map` oraz `Simulation`.



Rysunek 5.8: Okno Menu

Sekcja `Map` daje możliwość zmiany parametrów takich jak:

1. Szerokość `Width` i wysokość `Height` mapy.
2. Ilości ulic poziomych `Horizontal` i pionowych `Vertical`.
3. Zagęszczenia samochodów `Density`,

Aby zatwierdzić wybrane parametry, należy nacisnąć przycisk `Submit`.

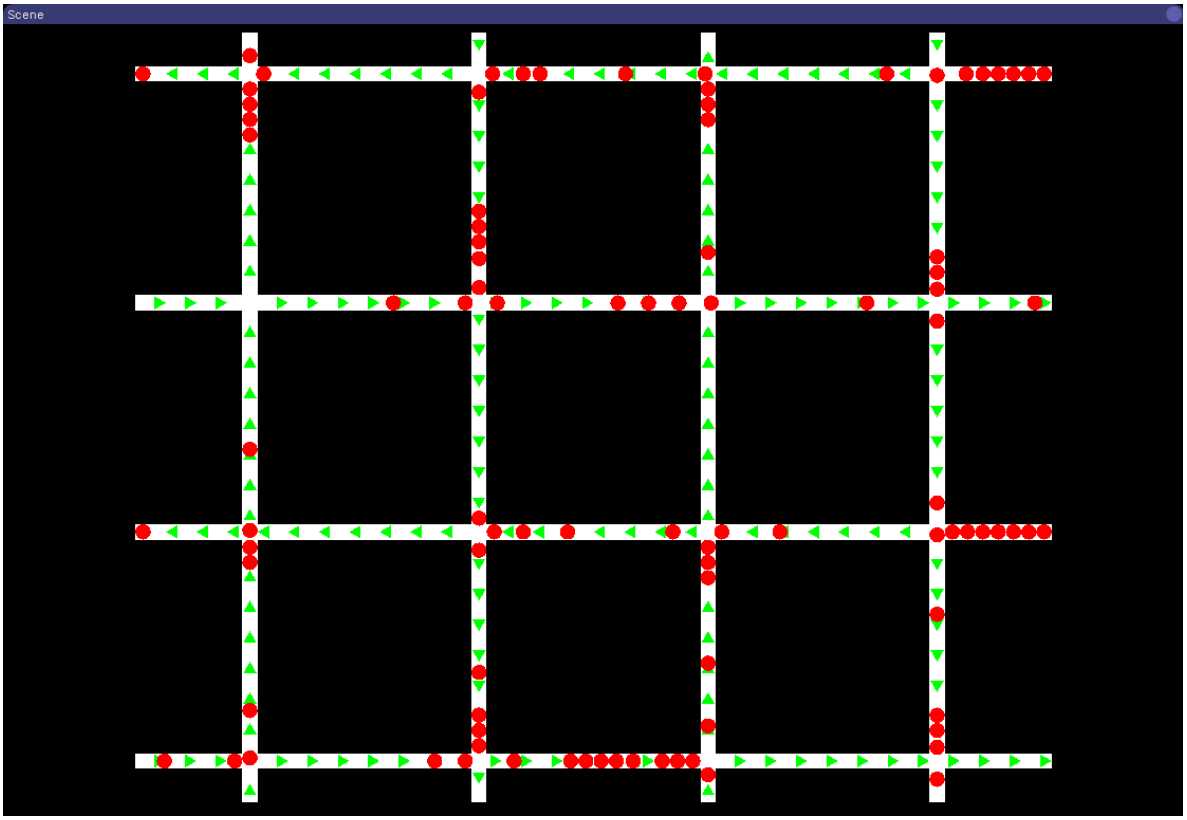
Sekcja `Scene` pozwala na manipulacje odstępem czasu między każdym krokiem symulacji `Delay` oraz posiada przycisk `Gather Data` włączający funkcje o tej samej nazwie. Funkcja `GatherData` służy do przeprowadzenia symulacji bez wizualizacji w celu zebrania danych do analizy. Dane które zbiera zostaną opisane w rozdziale 6.2.

5.2.2 Okno Scene

Okno `Scene`, w którym umieszczony jest przebieg wizualizacji, umożliwia przybliżanie i oddalanie mapy oraz przesuwanie jej za pomocą myszki komputerowej.

5.3 Wizualizacja

Wizualizacja polega na wywołaniu funkcji `render` z klasy `Map` i `Simulation` (przykład na Rys. 5.9). W klasie `Map` funkcja ta iteruje po całej tablicy `cellsType`, wyrysowując każdą komórkę jako kwadrat. Puste komórki są omijane. To znaczy, że w ich miejscu nic nie jest rysowane. Drogi są reprezentowane przez białe kwadraty. Na co drugim kwadracie jest narysowany zielony trójkąt, wskazujący kierunek drogi. Funkcja `render` z klasy `Simulation` rysuje wszystkie samochody. Przyjmują one wygląd czerwonych kółek. Aby animacja była ciągła, nie są one rysowane w pozycji pojazdu, lecz w pozycji $\overrightarrow{rendPosition} = \overrightarrow{oldPosition} + delta * \overrightarrow{velocity}$ która jest interpolacją liniową $\overrightarrow{oldPosition}$ i $\overrightarrow{position}$. Parametr $delta$ to czas przyjmujący wartości od 0 do 1 równy $delta = elapsed/delay$, gdzie $elapsed$ to czas który minął od rozpoczęcia kroku, a $delay$ to określone opóźnienie między krokami.



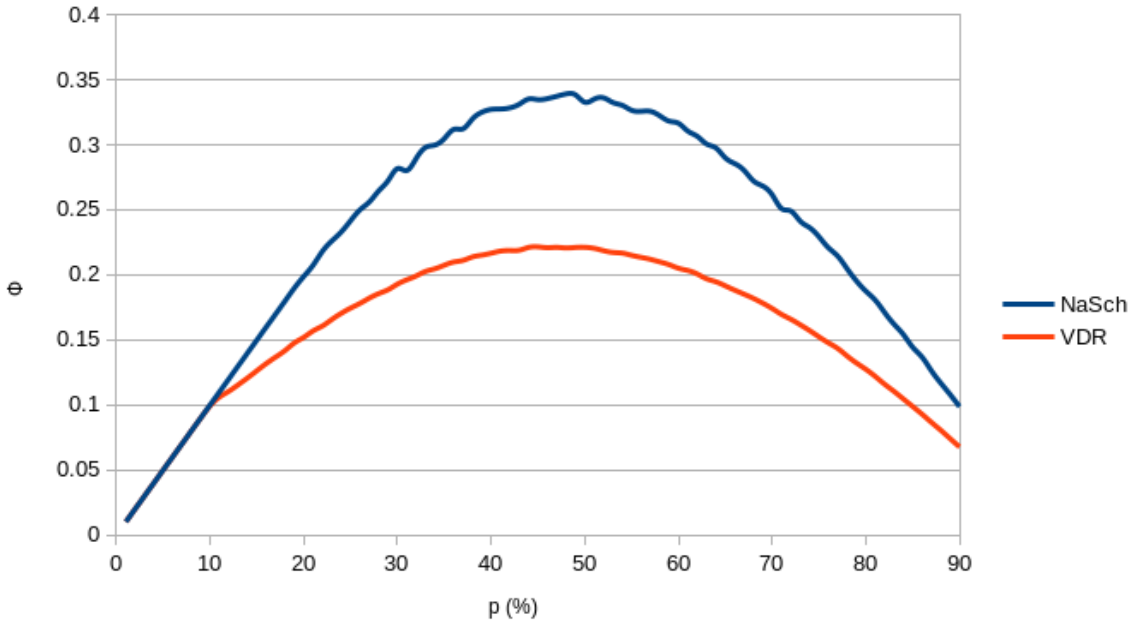
Rysunek 5.9: Okno Scene.

Rozdział 6

Przykład użycia

6.1 Pojęcia

Diagram fundamentalny ruchu drogowego jest diagramem ukazującym relacje między strumieniem ruchu a natężeniem ruchu (zagęszczeniem). Obrazuje on, że do pewnego poziomu natężenia, strumień wzrasta proporcjonalnie do natężenia. Dzieje się tak ze względu na to, że przy małym natężeniu praktycznie nie występuje interakcja między samochodami i mogą one przez większość czasu jechać z maksymalną prędkością. Od pewnego poziomu zagęszczenia interakcja zaczyna być zauważalna i mimo tego, że jest więcej pojazdów, przepływ zaczyna być mniejszy i krzywa zaczyna opadać. Wierzchołek krzywej jest uważany za pojemność drogi. Ukazuje on optymalne przypływ i zagęszczenie.



Rysunek 6.1: Diagram fundamentalny. Na osi poziomej odłożono procentową wartość zagęszczenia p, zaś na osi pionowej odłożono strumień ruchu ϕ.

Przykładowy diagram fundamentalny został zobrazowany na rys. 6.1. Pokazuje on dwie krzywe: niebieską (dla modelu Nagela–Schreckenberga) i czerwoną (dla modelu VDR). Na początku krzywe się pokrywają, ponieważ nie występują zależności między autami i żadne auto się nie zatrzymuje. Od pewnego momentu model NaSch charakteryzuje się większym strumieniem w stosunku do modelu VDR. Spowodowane jest to dodatkowym krokiem w modelu VDR. Sprawia on, że samochody, które stanęły mają mniejsze prawdopodobieństwo do ponownego ruszenia w następnym kroku. Przez to istnieje większa szansa na powstanie zatoru. Tę zależność można łatwo zaobserwować na diagramie.

Zagęszczenie ρ w kontekście problemu jest stosunkiem liczby wszystkich pojazdów N_C do liczby komórek będących ulicami lub skrzyżowaniami N_S:

$$\rho = \frac{N_C}{N_S}$$

Strumień ruchu ϕ w kontekście problemu jest stosunkiem sumy prędkości wszystkich pojazdów $V = \sum_{n=1}^{N_C} \|\vec{v}_n\|$ do liczby zajętych komórek będących ulicami lub skrzyżowaniami N_S:

$$\phi = \frac{V}{N_S}$$

6.2 Opis obliczeń

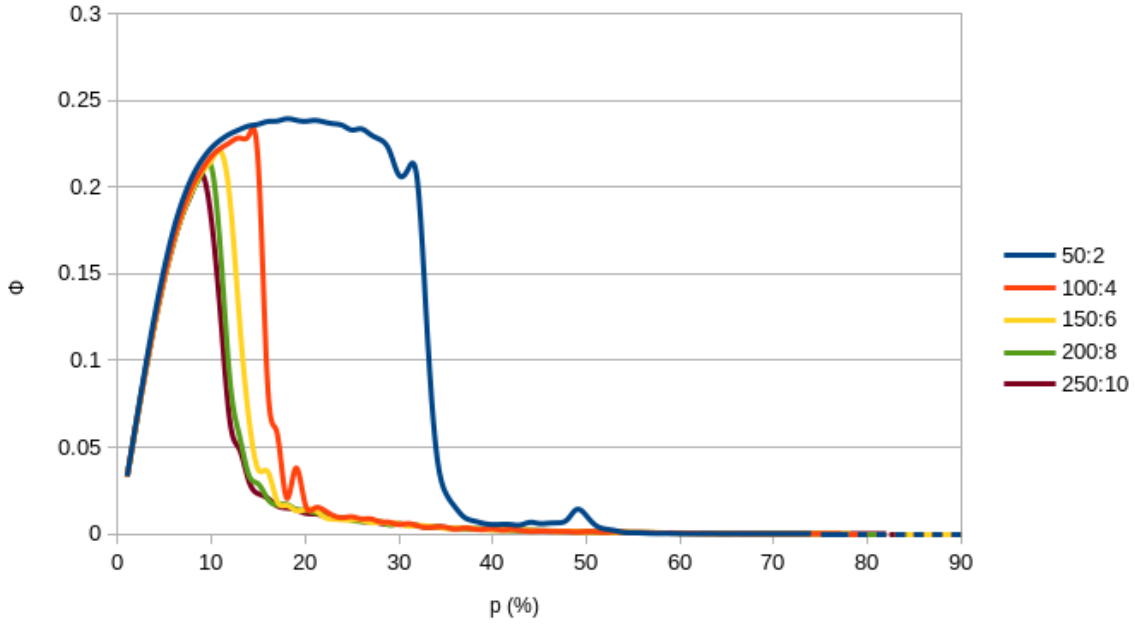
Aby wyznaczyć diagram fundamentalny przeprowadzono szereg symulacji, dla różnych gęstości z zakresu od 1% do 90%. Aby zbadać wpływ rozmiaru mapy na uzyskane wyniki rozpatrzono kilka rozmiarów: 50x50, 100x100, 150x150, 200x200, 250x250. W przypadku rozmiaru 50x50 liczbę ulic dobrano jako 2x2, zwiększano ją proporcjonalnie przy zwiększaniu rozmiaru mapy. Pojedyncza symulacja polega na wykonaniu 10000 iteracji. Żeby model osiągnął stan równowagi odrzuca się 100 pierwszych iteracji i dopiero po nich wyniki zostają zbierane. Symulacje przeprowadzono dla 10 różnych ziaren losowości, a wyniki zostały uśrednione. Wyniki zostały zebrane do pliku o formacie csv. W każdym wierszu zapisano informacje o:

1. Szerokości i wysokości mapy.
2. Ilości ulic poziomych i pionowych.
3. Zagęszczeniu samochodów.
4. Uśrednionym strumieniu samochodów z 10 symulacji dla różnych ziaren.

Do symulacji parametry zostały dobrane następująco:

1. Prędkość maksymalna $v_{max} = 5$.
2. Parametr losowego spowalniania $p_0 = 0.5$ i $p_1 = 0.3$.

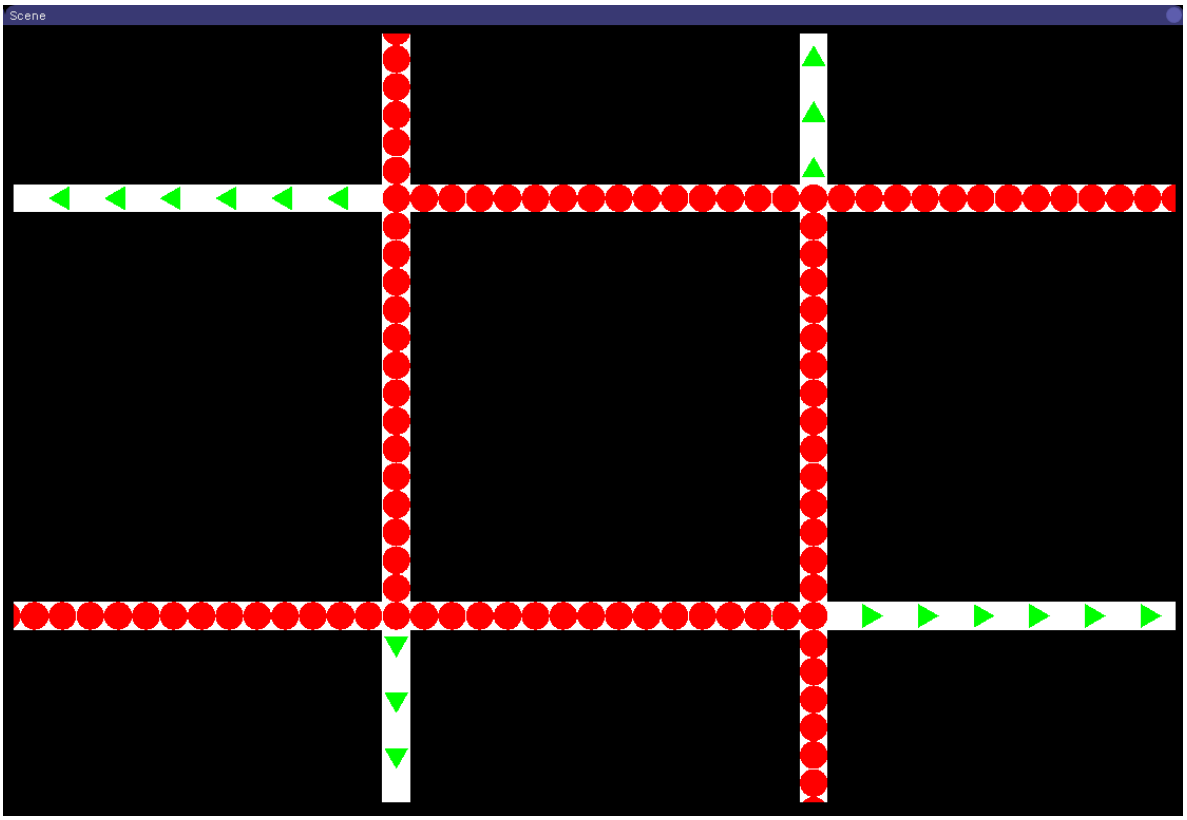
6.3 Wyniki



Rysunek 6.2: Diagram fundamentalny.

W diagramie fundamentalnym dla opracowanego modelu (Rys. 6.2) w zakresie gęstości od 1% do 8% strumień rośnie liniowo wraz ze wzrostem gęstości. Osiąga mniejszy strumień maksymalny niż modele NaSch i VDR. Jest to spowodowane interakcją samochodów ze skrzyżowaniami, która jest dodatkowym czynnikiem w stosunku do wcześniej wspomnianych modeli. Opracowany model w pewnym momencie osiąga punkt krytyczny, w którym powstaje zakleszczenie modelu (Rys. 6.3) na którymś ze skrzyżowań, przez co po pewnym czasie wszystkie samochody przestają się poruszać. W skutek tego strumień dąży do zera. Zakleszczenie modelu spowodowane jest tym, że samochód który jest w centrum skrzyżowania, mimo że ma możliwość pojechania drugą drogą, czeka aż zwolni się ta, którą wybrał wjeżdżając na skrzyżowanie. Blokuje przy tym drogę na której się znajdował wjeżdżając na skrzyżowanie. Stojące za nim samochody, w końcu zablokują inne skrzyżowanie. Sytuacja będzie się powtarzać tak długo aż żaden samochód nie będzie mógł się poruszyć. Ta sytuacja została nazwana zakleszczenie modelu.

50:2 to układ podstawowy, zaś kolejne (100:4, 150:6, 200:8, 250:10) to układy wybrane ze względu na rozmiar 2-, 3-, 4- i 5- krotny. Dla większych modeli zakleszczenie następuje szybciej ponieważ przy tym samym zagęszczeniu jest więcej pojazdów, a odstępy między skrzyżowaniami w każdym przypadku są takie same.



Rysunek 6.3: Zakleszczenie.

Aby wyeliminować zakleszczenia, można by wprowadzić warunek: gdy samochód znajduję się w centrum skrzyżowania a droga którą chce jechać jest zajęta, to powinien wybrać drugą możliwą drogę.

Rozdział 7

Podsumowanie

W niniejszej pracy zaproponowano model do symulacji ruchu drogowego opierający się na automatach komórkowych. Składa się on z dwuwymiarowej siatki komórek, na której są umieszczone jednokierunkowe ulice oraz pojazdy. Bazując na modelu VDR, został opracowany zestaw reguł dla samochodów będących poza skrzyżowaniem. Na rzecz pracy zostały opracowane również reguły obowiązujące podczas wjeżdżania na skrzyżowanie. Zakładają one, że samochód, który chce wjechać na skrzyżowanie, musi przestrzegać dwóch reguł. Musi zawsze poczekać aż centrum skrzyżowania oraz jego prawa strona będą wolne. Reguły są zgodne z zasadami panującymi na skrzyżowaniu równorzędnym.

Na rzecz pracy została opracowana aplikacja służąca do wizualizacji opracowanego modelu. Została ona napisana w języku c++ przy podejściu obiektowym. Do stworzenia wizualizacji została użyta biblioteka SFML. Natomiast do stworzenia graficznego interfejsu użytkownika została użyta biblioteka ImGui. Aplikacja wyświetla drogi, po których poruszają się samochody. Wizualizacje można przybliżać i oddalać oraz przesuwać. Interfejs użytkownika umożliwia zmianę parametrów symulacji.

Opracowany model ukazał, że za pomocą automatów komórkowych można modelować ruch drogowy na siatce dróg jednokierunkowych, które mogą się krzyżować. Dzięki modelowi można zaobserwować jaki wpływ na ruch drogowy mają skrzyżowania równorzędne oraz że samochody, które czekają na środku skrzyżowania, aż zwolni się jedna droga, zamiast wybrać drugą drogę, tworzą korki. Diagram fundamentalny opracowanego modelu ukazał, że opracowany model osiąga mniejszy strumień ruchu niż modele Nagela–Schreckenberga i VDR.

Wykaz literatury

[1] *SFML*. `http://www.sfml-dev.org/index.php`. data dostępu: 27.11.2016 r.

[2] *ImGui*. `https://github.com/ocornut/imgui`. data dostępu: 27.11.2016 r.

[3] Stephen Wolfram. “Statistical Mechanics of Cellular Automata”. W: *Reviews of Modern Physics* (1983), 601–644.

[4] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[5] Hector Zenil. “Compression-Based Investigation of the Dynamical Properties of Cellular Automata and Other Systems”. W: *Complex Systems* 19 (2010), 1–28.

[6] Kai Nagel i Michael Schreckenberg. “A cellular automaton model for freeway traffic”. W: *J. Physique* I 2 (1992), 2221–2229.

[7] Torsten Held i Michael Stefan Bittihn. “Cellular automata for traffic simulation Nagel-Schreckenberg model”. W: (2011).

Spis rysunków

Spis tablic