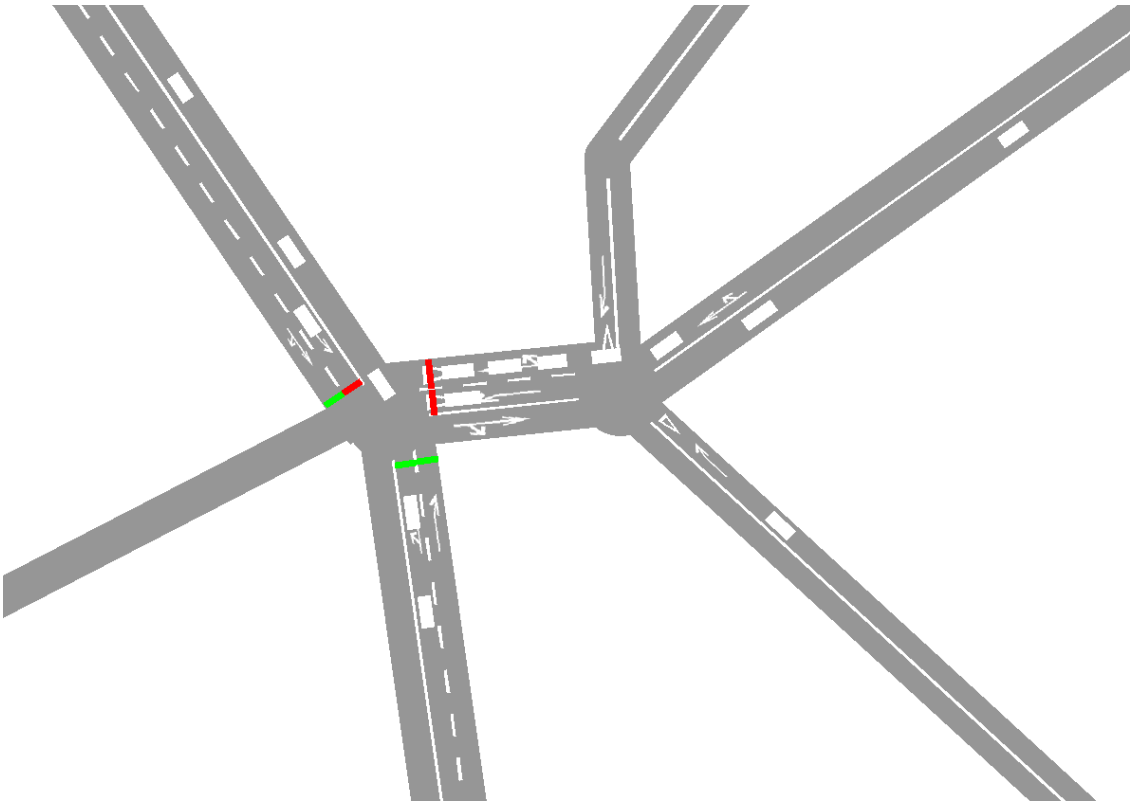
Kantonsschule Zürcher Oberland

Maturitätsarbeit 2025

Entwicklung einer mikroskopischen Verkehrssimulation

Ronny Siegenthaler, C6a



Betreuung
Albert Kern

Datum
25. November 2024

Abstract

Meine Arbeit befasst sich mit der Entwicklung einer mikroskopischen Verkehrssimulation, die den Strassenverkehr einer Kleinstadt möglichst realitätsgetreu abbildet. Das Programm wurde in Python geschrieben. Es ist aufgebaut auf existierenden Modellen wie dem Intelligent-Driver-Modell sowie eigenen Ideen. Komplizierte Details, die am Ergebnis wenig ändern, werden weggelassen. Ein GUI (Graphical User Interface) stellt die Simulation grafisch dar. Mit der Simulation wurde die Verkehrssituation der Stadt Wetzikon analysiert. Anschliessend konnten Vorschläge gemacht werden, um die Verkehrssituation der Stadt zu verbessern.

Inhaltsverzeichnis

1 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Simulation, die den Strassenverkehr möglichst realitätsnah abbilden soll. Da eine Simulation immer beliebig kompliziert werden kann, müssen sinnvolle Grenzen gesetzt werden, um die Machbarkeit im Rahmen einer Maturitätsarbeit sicherzustellen. Die Simulation beschränkt sich daher auf den motorisierten Individualverkehr als einzigen Verkehrsteilnehmer. Mathematisch oder programmiertechnisch komplizierte Details, die am Ergebnis wenig ändern, werden weggelassen oder durch einfache Schätzungen ersetzt. Ein Beispiel dafür wäre die Implementierung von Strassen mit Kurven. Des Weiteren erreiche ich mit dieser Arbeit das Ziel, meine rudimentären Programmierkenntnisse in Python auszubauen und lerne viel über das Thema der Verkehrsplanung, welches mich schon lange interessiert hat.

Mit der Simulation soll die Verkehrssituation einer Kleinstadt analysiert werden, sodass anschliessend sinnvolle Verbesserungen am Strassennetzwerk vorgenommen werden können. Diese Verbesserungen sollen dann wiederum mit der Simulation auf ihre Wirksamkeit untersucht werden. Insbesondere sollen Massnahmen untersucht werden, die den Verkehr innerhalb der Stadt verlangsamen und reduzieren. Dies kann beispielsweise durch leistungsfähige Umfahrungen um das Stadtzentrum geschehen.

2 Entwicklung der Simulation

2.1 Verwendete Tools zur Programmierung

Als Programmiersprache habe ich Python gewählt, da sich meine Programmierkenntnisse auf diese Sprache beschränken. Aus dem AM-Unterricht von Beat Jäckle war ich bereits vertraut mit dem Einsatz von Bedingungen, Schleifen, Funktionen, Classes und sogar dem Einlesen von JSON und CSV-Dateien. Im Verlauf der Arbeit stellte sich heraus, dass diese Kenntnisse bereits für einen Grossteil des Programms ausreichten. Ansonsten habe ich die offizielle Anleitung von Python[1] zu Hilfe gezogen.

Als IDE habe ich zuerst Geany[2] verwendet, da ich mit dieser Software bereits vertraut war. Später wechselte ich jedoch auf Microsoft Visual Studio Code[3], vor allem weil die Oberfläche für mich ansprechender erschien. Visual Studio ist eine vollständige Programmierumgebung, während Geany eher als erweiterter Text-Editor bezeichnet werden kann. Somit hat Visual Studio viele praktische Funktionen, die das Programmieren erleichtern. Schlussendlich ist es Geschmackssache, welche IDE man bevorzugt.

---

[1] Python Software Foundation: *The Python Tutorial*. Auf: https://docs.python.org/3/tutorial/ (abgerufen am 19.10.2024).
[2] The Geany contributors: *About Geany*. Auf: https://www.geany.org/about/geany/ (abgerufen am 19.10.2024).
[3] Microsoft: *Visual Studio Code*. Auf: https://code.visualstudio.com/ (abgerufen am 19.10.2024).

Das Programm soll, soweit sinnvoll, die Vorschriften der PEP 8 einhalten. Dies ist die allgemeine «Style Guide» für Python-Code. Dafür habe ich Pylint[4] benutzt, welches als Erweiterung sowohl in Geany als auch in Visual Studio verfügbar ist. Pylint überprüft den Code automatisch auf Unschönheiten und potenzielle Fehlerquellen und gibt sogar Vorschläge an, wie man diese beheben könnte. Mehr dazu in Kapitel 2.12.

2.2 Dokumentation der Programmierung

Ich entschied mich, auf spezialisierte Versionsverwaltungsprogramme wie Git zu verzichten, obwohl diese für meine ehemalige AM-Lehrperson sehr wichtig gewesen wären. Stattdessen legte ich ungefähr jeden Arbeitstag eine Kopie der Python-Datei an und gab ihr eine neue Versionsnummer. Für ein Projekt meiner Grösse war dies völlig ausreichend, um den Überblick zu behalten.

Neben der Versionsnummer enthält jede Version im Dateinamen auch das Datum. Zusätzlich erstellte ich ein Word-Dokument, in dem die Änderungen jeder Version protokolliert wurden. Alle Dateien wurden laufend mit Microsoft Teams synchronisiert, um mich mit Albert Kern, meiner betreuenden Lehrperson, über den Fortschritt auszutauschen.

2.3 Einführung in verschiedene Arten von Verkehrsflussmodellen

Für das Verständnis der folgenden Kapitel ist es wichtig, den grundsätzlichen Unterschied zwischen makroskopischen und mikroskopischen Verkehrsflussmodellen[5] zu kennen:

Makroskopische Modelle betrachten den Verkehrsfluss ähnlich wie eine strömende Flüssigkeit oder ein Gas. Die verwendeten Grössen sind *lokal aggregiert*, d.h. sie sind räumlich und zeitlich veränderlich. Verwendete Grössen sind Verkehrsdichte (Fahrzeuge pro Strecke), Verkehrsfluss (Fahrzeuge pro Zeit) und mittlere Geschwindigkeit. Diese Modelle können Staus und Ausbreitungsgeschwindigkeiten von Störungen sehr gut beschreiben. Vor allem benötigen sie wenig Rechenleistung und ermöglichen daher eine Auswertung in Echtzeit, z. B. zur Abschätzung der Verkehrslage auf einer Autobahn.

Mikroskopische Modelle werden auch *agentenbasierte Modelle* genannt, da sie die einzelnen Agenten, in diesem Fall die Fahrzeuge, beschreiben. Somit wird das Verhalten jedes einzelnen Fahrers in Abhängigkeit von seinen Nachbarn berechnet. Dafür werden Grössen wie die Position, Geschwindigkeit und Beschleunigung eines Fahrzeuges benötigt. Mikromodelle sind folglich genauer als Makromodelle, benötigen aber auch entsprechend mehr Rechenleistung. Ob diese Genauigkeit benötigt wird, hängt immer von der zu modellierenden Verkehrssituation ab. Beispielsweise lässt sich der Einfluss von verschiedenen Fahrzeugtypen auf den Verkehr nur schwer makroskopisch beschreiben.

---

[4] Pylint: *Star your Python code*. Auf: https://www.pylint.org/ (abgerufen am 19.10.2024).
[5] Treiber, Martin; Kesting, Arne: *Verkehrsdynamik und -simulation. Daten, Modelle und Anwendungen der Verkehrsflussdynamik*. Dresden, 2010, S. 52 ff.

Aus Mikromodellen lassen sich durch *lokale Aggregierung* makroskopische Grössen wie Dichte, Fluss oder mittlere Geschwindigkeit berechnen. Voraussetzung dafür ist, dass auf eine sinnvolle Weise mehrere Fahrzeuge gruppiert werden können.

Für meine Simulation habe ich mich für den mikroskopischen Ansatz entschieden, da der Verkehr möglichst realistisch abgebildet werden soll. Ein weiterer Grund ist, dass Makromodelle wesentlich abstrakter als Mikromodelle und daher schwieriger zu verstehen sind. Im mikroskopischen Modell kann jedes Fahrzeug genau beobachtet werden, womit sich das Programm leicht auf seine Funktionsfähigkeit überprüfen lässt.

2.4 Grundstruktur des Programms

Die Mikroskopische Verkehrssimulation eignet sich bestens für die objektorientierte Programmierung, also das Arbeiten mit Classes (Klassen)[6]. Classes sind eine Ansammlung von Variablen und Methoden (Funktionen der Class), die zusammen einen neuen Datentyp bilden. Da ich mich mit Classes bereits auskannte, begann ich direkt mit dem Schreiben einer Class für Fahrzeuge. Später kamen weitere Classes dazu, nämlich für Fahrspuren, Strassen, Kreuzungen, Lichtsignale, die Simulation selbst sowie für die grafische Darstellung. Gewisse Classes sind einer anderen Class übergeordnet und enthalten Objekte der untergeordneten Class. Beispielsweise enthält eine Fahrspur eine Liste mit allen Fahrzeugen, die sich gerade darauf befinden. Jede Class enthält mindestens zwei Methoden:

Die Methode `__init__` initialisiert das Objekt und definiert alle Parameter (Instance Variables). Zuerst kannte ich den Unterschied zwischen Instance Variables und Class Variables nicht. Class Variables werden ausserhalb von `__init__` definiert und sind für alle Instanzen der Klasse gleich. Dies führte für mich zu Problemen, da ich zuerst alle Variablen als Class Variables definierte und anschliessend versuchte, diese für jede Instanz unterschiedlich anzupassen.

Zusätzlich hat jede Class die Methode `update`, die das Objekt sowie alle darin enthaltenen Objekte für eine gewisse Zeitspanne aktualisiert.

```
'Programm zur Simulation von Strassenverkehr'

# Import von benötigten Modulen
import math
import random
import copy
import json
import pygame

class Vehicle:
    # Class für Fahrzeuge
class Lane:
    # Class für Fahrspuren
class Road:
```

[6] Python Software Foundation: *Classes*. Auf: https://docs.python.org/3/tutorial/classes.html (abgerufen am 19.10.2024).

```
    # Class für Strassen
class TrafficLight:
    # Class für Lichtsignale
class Intersection:
    # Class für Kreuzungen
class Simulation:
    # Class für die Simulation
class Window:
    # Class für die grafische Darstellung

def main():
    # Initiieren und Ausführen der Simulation

main()
```

Code 1: Grundstruktur des Codes

In der endgültigen Version befindet sich fast der gesamte Programmcode in Classes, wobei nur noch eine Funktion `main` benötigt wird, welche die Simulation erstellt und mit einer while-Schleife so lange ausführt, bis das Programm beendet wird.

2.5 Fahrzeuge

Mein erstes Ziel war es, eine Kolonne von hintereinanderfahrenden Fahrzeugen darzustellen. Dafür erstellte ich eine `class Vehicle` und definierte darin einige grundlegende Variablen wie Position, Geschwindigkeit, Fahrzeuglänge und Maximalgeschwindigkeit. Mit diesen Daten und jenen des vorausfahrenden Fahrzeuges in der Kolonne sollte das Fahrzeug seine Beschleunigung berechnen und sich entsprechend fortbewegen. Zuerst versuchte ich, aus den kinematischen Bewegungsgleichungen selbst eine sinnvolle Formel zur Berechnung der Beschleunigung zu finden. Meine Versuche ergaben jedoch kein realistisches Fahrverhalten. Deshalb musste ich nach einem einfachen Modell suchen, um das Beschleunigungsverhalten von Fahrzeugen zu berechnen. Meine Suche stiess dabei auf das Intelligent-Driver-Modell, welches im folgenden Unterkapitel genauer beschrieben wird.

2.5.1 Intelligent-Driver-Modell

Das Intelligent-Driver-Modell (IDM) ist ein einfaches und vollständig unfallfreies Fahrzeugfolgemodell, das in allen Verkehrssituationen realistische Beschleunigungswerte liefert. Das Modell wurde aus den folgenden Grundannahmen hergeleitet:[7]

1. Falls es keine Hindernisse durch andere Fahrzeuge gibt, sollte das Fahrzeug auf seine Wunschgeschwindigkeit v_0 beschleunigen.
2. Die Beschleunigung sollte abnehmen, je geringer der Abstand zum vorausfahrenden Fahrzeug ist.
3. Der Abstand zum Vorderfahrzeug sollte einen Mindestabstand s_0 und einen Sicherheitsabstand vT nicht unterschreiten. T ist eine Zeitlücke zwischen den Fahrzeugen.

[7] Treiber et al. 2010, S. 161 ff.

4. Die Bremsbeschleunigung soll, sofern die Situation unter Kontrolle ist, eine gewisse komfortable Bremsbeschleunigung b nicht unterschreiten. Um die Unfallfreiheit zu gewährleisten, darf die komfortable Bremsbeschleunigung auch überschritten werden.
5. Es sollte keine ruckartigen Änderungen in der Beschleunigung geben, sondern nur weiche Übergänge.
6. Das Modell sollte möglichst einfach sein und anschauliche Modellparameter besitzen, die mit Aspekten des menschlichen Fahrverhaltens in Verbindung gebracht werden können.

Durch die folgende Modellgleichung werden alle Grundannahmen erfüllt:

$$\frac{dv}{dt} = a\left(1 - \left(\frac{v}{v_0}\right)^{\delta} - \left(\frac{s^*(v,\Delta v)}{s}\right)^2\right)$$

$$s^*(v,\Delta v) = s_0 + vT + \frac{v\Delta v}{2\sqrt{ab}}$$

Modellparameter

a: Maximalbeschleunigung, mit der das Fahrzeug auf freier Strecke bis auf seine Wunschgeschwindigkeit beschleunigt.

b: Komfortable Bremsbeschleunigung, die in Normalsituationen beim Annähern an langsamere Fahrzeuge nicht überschritten werden soll.

v: Momentane Geschwindigkeit des Fahrzeuges

v_0: Wunschgeschwindigkeit, die auf freier Strecke erreicht werden soll.

δ: Beschleunigungsexponent, der das Beschleunigungsverhalten auf freier Strecke bestimmt. Je grösser δ, desto später wird die Beschleunigung wieder auf 0 reduziert.

s: Aktueller Abstand zum vorausfahrenden Fahrzeug

s^*: Wunschabstand, der mit der zweiten Gleichung berechnet wird.

s_0: Mindestabstand zwischen zwei Fahrzeugen bei stehendem Verkehr

T: Zeitabstand bei fahrendem Verkehr

Δv: Geschwindigkeitsunterschied zum Vorderfahrzeug (positiv, wenn dieses langsamer fährt)

Die Modellgleichung kann in zwei Teile unterteilt werden: einen Anteil für die Beschleunigung auf freier Strecke $a\left(1 - \left(\frac{v}{v_0}\right)^{\delta}\right)$ und einen Anteil für die Interaktion mit anderen Fahrzeugen $-a\left(\left(\frac{s^*(v,\Delta v)}{s}\right)^2\right)$.

Im ersten Teil wird das Verhältnis zwischen v und v_0 berechnet. Je grösser der Unterschied zwischen diesen Geschwindigkeiten, desto grösser ist die resultierende

Beschleunigung, wobei die Maximalbeschleunigung a nicht überschritten wird. Somit ist die Beschleunigung beim Losfahren am grössten und nimmt dann zunehmend ab. Der Beschleunigungsexponent δ bewirkt, dass die Beschleunigung länger konstant bleibt und erst kurz vor dem Erreichen der Wunschgeschwindigkeit gesenkt wird. Letztere wird eigentlich nie erreicht, da sich die Geschwindigkeit eines Fahrzeuges asymptotisch an v_0 annähert.

Ein Schwachpunkt des Modells zeigt sich, wenn v grösser als v_0 ist, d. h., wenn die Höchstgeschwindigkeit herabgesetzt wird. Dafür ist das Modell nicht ausgelegt, da in diesem Fall unrealistisch grosse negative Beschleunigungen entstehen. Dies werde ich später berücksichtigen.

Der zweite Teil ist ebenfalls als Verhältnis ausgedrückt, und zwar zwischen dem Wunschabstand s^* und dem tatsächlichen Abstand s. Er wirkt gewissermassen dem ersten Teil entgegen, da die berechnete Beschleunigung negativ ist. Je grösser s, desto kleiner wird die resultierende Bremsbeschleunigung. Der zweite Teil kann somit ganz weggelassen werden, wenn kein Vorderfahrzeug vorhanden ist. Der Wunschabstand s^* wird einzeln berechnet und ermöglicht die Einhaltung des Sicherheitsabstandes sowie ein dynamisches Bremsverhalten.

2.5.2 Implementierung des IDM in der Simulation

Lösung der Gleichungen

Beim IDM handelt es sich um gekoppelte Differentialgleichungen, da die Gleichung für ein Fahrzeug jeweils mit der des vorausfahrenden Fahrzeuges verbunden ist (durch die Werte s und Δv). Daraus resultiert ein Gleichungssystem, welches eine Differentialgleichung für jedes Fahrzeug in der Kolonne enthält. Eine exakte Lösung dieses Gleichungssystems würde meine mathematischen Fähigkeiten übersteigen und wäre nicht zielführend. Stattdessen wird ein numerisches Update-Intervall Δt definiert, während dessen eine konstante Beschleunigung angenommen wird. Je kleiner Δt, desto genauer ist die Lösung. Mit einfachen kinematischen Gleichungen kann der neue Ort und die neue Geschwindigkeit des Fahrzeuges ausgerechnet werden:[8]

Neue Geschwindigkeit: $$v(t + \Delta t) = v(t) + \frac{dv}{dt}\Delta t$$

Neuer Ort: $$x(t + \Delta t) = x(t) + \frac{1}{2}\frac{dv}{dt}(\Delta t)^2$$

Aufgrund der kleinen Ungenauigkeit dieser Lösung kann es passieren, dass ein Fahrzeug leicht über sein Ziel hinausfährt, was anschliessend zu einer negativen Geschwindigkeit führen würde. Da dies nicht passieren soll, muss sichergestellt werden, dass negative Geschwindigkeiten nicht auftreten können.

[8] Treiber, Martin: *The Intelligent-Driver Model and its Variants*. Auf: https://traffic-simulation.de/info/info_IDM.html (abgerufen am 12.10.2024).

Umsetzung in Python

Um die Beschleunigung berechnen zu können, wurde in der Fahrzeug-Class die Methode `IDM_acceleration` erstellt, die genau die Berechnungen der IDM-Formel durchführt. Zu Beginn wählte ich sinnvolle Modellparameter und setzte diese direkt in die Formeln ein. Später ersetzte ich diese durch Variablen, die für jedes Fahrzeug zufällig definiert werden.

```
def IDM_acceleration(self, distance: float, next_vehicle_speed: float) -> float:
    'Berechnung der IDM-Beschleunigung'
    d = distance
    free_road_term = self.max_acceleration * (1 - (self.speed / self.speed_limit) ** 4)
    # Begrenzung der Bremsbeschleuningung bei Reduktion der Maximalgeschwindigkeit
    if free_road_term < -self.deceleration:
        free_road_term = -self.deceleration
    s = (self.minimum_distance + self.speed * self.safety_time
        + (self.speed * (self.speed - next_vehicle_speed))
        / (2 * (self.max_acceleration * self.deceleration) ** 0.5))
    interaction_term = self.max_acceleration * ((s / d) ** 2)
    a = free_road_term - interaction_term
    return a
```

Code 2: Endgültige Methode IDM_acceleration

Die zuvor erwähnten zwei Teile der Modellgleichung werden hier separat berechnet, um für den ersten Teil noch eine Bedingung anwenden zu können. Im späteren Verlauf stellte sich nämlich heraus, dass bei einer Reduktion der Höchstgeschwindigkeit `self.speed_limit` das IDM eine extrem grosse negative Beschleunigung berechnet, mit der viel zu schnell auf die neue Höchstgeschwindigkeit heruntergebremst wird. Da dieses Verhalten nicht realistisch ist, fügte ich die Bedingung hinzu, dass der erste Teil nicht kleiner als die komfortable Bremsbeschleunigung werden darf.

In der Methode `update` werden dann die neue Geschwindigkeit und der neue Ort nach dem Zeitabschnitt `dt` berechnet:

```
def update(self, next_vehicle, dt: float):
    'Update des Fahrzeuges für eine bestimmte Zeit'
    self.travel_time_counter += dt  # Zähler für die Fahrzeit
    self.path_finder_timer += dt    # Zähler für die Routenplanung
    self.lane_change_timer += dt    # Zähler für den Spurwechsel

    # Berechnung der Beschleunigung
    a = self.acceleration(next_vehicle)
    # Vorwärtsbewegung mit zuvor berecherter Beschleunigung
    self.speed += a * dt
    if self.speed < 0:
        self.speed = 0
    else:
            self.position += (self.speed * dt) + (a * (dt ** 2)) / 2
```

Code 3: Berechnung des neuen Ortes und der neuen Geschwindigkeit eines Fahrzeuges

Hier fällt auf, dass nicht direkt die Methode `IDM_acceleration`, sondern die Methode `acceleration` ausgeführt wird. Diese wurde später als ein «Zwischenschritt» hinzugefügt, um das Fahrzeug allenfalls vor einer Kreuzung anhalten zu können und den Abstand zum nächsten Fahrzeug zu berechnen. Die Bestimmung des nächsten Fahrzeuges erfolgt nicht in der `class Vehicle`, sondern in der übergeordneten `class Lane`.

2.5.3 Zufällige Bestimmung der Fahrzeugparameter

Für das IDM müssen Parameter gesetzt werden, die menschliches Fahrverhalten möglichst akkurat abbilden. Da es unterschiedliche Fahrzeugtypen gibt und jeder Fahrer einen anderen Fahrstil hat, sollten die Parameter nicht für jedes Fahrzeug gleich sein. Ich entschied mich dafür, zwischen zwei grundsätzlichen Fahrzeugtypen zu unterscheiden: Autos und Lastwagen. Mit dem Modul `random` sollten zum Typ passende Werte aus einem bestimmten Bereich ausgewählt werden. Der folgende Code wird direkt in der Methode `__init__` ausgeführt:

```
# Zufällige Bestimmung von Fahrzeugparametern je nach Typ
if vehicle_type == 'car':
    self.vehicle_type: str = 'car'
    self.lenght: float = random.uniform(4, 5)
    self.max_speed: float = 200/3.6
    self.max_acceleration: float = random.triangular(1, 2)
    self.deceleration: float = random.triangular(1.5, 3)
    self.safety_time: float = random.triangular(1.2, 1.7)
    self.politeness: float = random.triangular(0.3, 0.7)
    self.width: float = 1.8
    self.lane_change_time: float = 2
elif vehicle_type == 'truck':
    self.vehicle_type: str = 'truck'
    self.lenght: float = random.uniform(10, 18.75)
    self.max_speed: float = 80/3.6
    self.max_acceleration: float = random.triangular(0.75, 1.25)
    self.deceleration: float = random.triangular(1, 1.75)
    self.safety_time: float = random.triangular(1.3, 1.8)
    self.politeness: float = random.triangular(0.3, 0.7)
    self.width: float = 2.3
    self.lane_change_time: float = 3
else:
    raise ValueError('Ungültiger Fahrzeugtyp!')
self.speed_limit: float = self.max_speed
```

Code 4: Zufällige Bestimmung der Fahrzeugparameter

Die Werte habe ich aus dem Text von M. Treiber[9] entnommen und nach eigenem Ermessen angepasst. Lastwagen haben lange Bremswege und kleine Beschleunigungen, daher wurden kleinere Werte für maximale Beschleunigung und Bremsbeschleunigung sowie ein etwas grösserer Zeitabstand gewählt. Die Maximalgeschwindigkeit für Lastwagen wurde auf 80 km/h begrenzt, da diese in der Schweiz nicht schneller fahren dürfen.[10]

Wichtig ist der Unterschied zwischen `self.speed_limit` und `self.max_speed`: Die Variable `self.speed_limit` ist die aktuelle Wunschgeschwindigkeit des Fahrzeuges, während `self.max_speed` die maximale Geschwindigkeit ist, welche das Fahrzeug erreichen kann. Mit der Methode `change_speed_limit` kann die Wunschgeschwindigkeit des Fahrzeuges

[9] Treiber et al. 2010, S. 163.
[10] Touring Club Schweiz: *Fahren auf der Autobahn – das sollten Sie wissen*. Auf: https://www.tcs.ch/de/testberichte-ratgeber/ratgeber/auto/autobahn-fahren.php (abgerufen am 15.11.2024)

verändert werden, wobei überprüft wird, ob dessen Maximalgeschwindigkeit dabei nicht überschritten wird.

Die Funktion `random.triangular` generiert zufällig einen Wert im angegebenen Bereich, wobei der Mittelwert am häufigsten vorkommt und die Wahrscheinlichkeit gegen aussen linear abnimmt. Dies hat zum Vorteil, dass Ausreisser zwar möglich sind, aber nicht so häufig sind wie der Durchschnitt.

2.5.4 Erster Testversuch

Um mein erstes Programm zu testen, welches noch nicht alle bereits beschriebenen Funktionen enthielt, initiierte ich zwei Objekte der Fahrzeug-Class und definierte für das erste Fahrzeug das zweite als Vorderfahrzeug. Die Position des zweiten Fahrzeuges setzte ich auf 500. Das erste Fahrzeug wurde losfahren gelassen, während das zweite nicht aktualisiert wurde. Somit blieb dieses stehen. Nach jedem Update-Intervall wurde der Abstand vom ersten zum zweiten Fahrzeug im Terminal angezeigt. Ich konnte beobachten, dass das erste Fahrzeug im Abstand von ca. 2m (dem definierten Mindestabstand) stehen blieb. Somit war bestätigt, dass das Programm bis dahin korrekt funktionierte.

2.6 Strassen und Fahrspuren

Nun musste eine Möglichkeit gefunden werden, um die Fahrzeuge zu gruppieren und in einer bestimmten Reihenfolge hintereinander anzuordnen. Zunächst speicherte ich die Fahrzeuge einfach in einer Liste, wobei das erste Fahrzeug in der Kolonne das letzte Element in der Liste ist. Somit können neue Fahrzeuge am Anfang der Liste hinzugefügt werden und Fahrzeuge, die das Ende der Strasse erreicht haben, am Ende der Liste entfernt werden. Das Vorderfahrzeug eines Fahrzeuges kann gefunden werden, indem das Element mit dem nächsthöheren Index in der Liste bestimmt wird. Wenn dieses Element nicht existiert, d. h., wenn das Fahrzeug bereits das letzte Element der Liste ist, hat es kein vorausfahrendes Fahrzeug und befindet sich auf freier Strecke.

2.6.1 Class für Strassen

Die Idee mit der Liste erwies sich als geeignet, jedoch musste es mehrere Strassen geben können, die unabhängig voneinander funktionieren. Deshalb verlegte ich die Liste `vehicles` in eine neue `class Road`. In dieser Class erstellte ich Methoden, um Fahrzeuge zur Strasse hinzuzufügen, aus der Strasse zu entfernen und um das Vorderfahrzeug eines Fahrzeuges zu bestimmen. Letztere wird im Folgenden etwas genauer betrachtet:

```
def next_vehicle(self, vehicle):
    'Bestimmung des vorausfahrenden Fahrzeuges'
    try:
        next_vehicle = self.vehicles[self.vehicles.index(vehicle)+1]
    except IndexError:
        next_vehicle = None
    return next_vehicle
```

Code 5: Erste Version der Methode next_vehicle

In dieser Methode wird mit `try` und `except` gearbeitet. Damit können Fehlermeldungen aufgefangen werden. In diesem Fall wird versucht, das Element mit dem nächsthöheren Index in der Liste `self.vehicles` zu bestimmen. Existiert dieses nicht, tritt ein `IndexError` auf. Dieser würde das Programm beenden, wird jedoch durch `except IndexError` aufgefangen. Dann wird das nächste Fahrzeug als `None` definiert. Die Fahrzeug-Class würde in diesem Fall davon ausgehen, dass die Strasse frei ist. In der Methode `update` werden alle Fahrzeuge der Strasse für das vorgegebene Zeitintervall aktualisiert.

2.6.2 Zweidimensionales Koordinatensystem

Um mehrere Strassen in der Simulation darstellen zu können, musste ein zweidimensionales Koordinatensystem eingeführt werden. Fahrzeugfolgemodelle wie das IDM funktionieren aber grundsätzlich nur in einem eindimensionalen Koordinatensystem, nämlich auf der Strasse bzw. der Fahrspur, auf der sich das Fahrzeug gerade befindet. Daher musste ich eine Möglichkeit finden, um die Positionen der Fahrzeuge auf der Strasse in zweidimensionale Koordinaten umzurechnen. Zuerst erstellte ich in der Strassen-Class zwei Variablen `self.pos_start` und `self.pos_end`. Diese haben den Typ `tuple`, sind also Koordinaten der Form (x, y). Daraus kann die Länge der Strasse mit dem Satz des Pythagoras berechnet werden. Diese Berechnung wird direkt in `__init__` durchgeführt:

```
self.lenght: float = ((pos_end[0]-pos_start[0])**2 + (pos_end[1]-pos_start[1])**2)**0.5
```

Code 6: Berechnung der Strassenlänge

Koordinaten von Fahrzeugen

Damit die Fahrzeuge später an der korrekten Position angezeigt werden, erstellte ich folgende Methode, um eine Position auf der Strasse in zweidimensionale Koordinaten umzurechnen:

```
def get_pos(self, position: float) -> tuple:
    'Berechnung der Koordinaten eines Fahrzeuges auf der Fahrspur'
    posx = (self.pos_start[0]
        + (self.pos_end[0] - self.pos_start[0]) / self.lenght * position)
    posy = (self.pos_start[1]
        + (self.pos_end[1] - self.pos_start[1]) / self.lenght * position)
    return (posx, posy)
```

Code 7: Umrechnung einer Position in zweidimensionale Koordinaten

Das Argument `position` ist die Position des Fahrzeuges auf der Fahrspur, also eine Zahl. Die Variablen `self.pos_start` und `self.pos_end` hingegen sind die Anfangs- und Endpositionen der Strasse in zweidimensionalen Koordinaten. Zur besseren Verständlichkeit betrachten wird diese als Vektoren. Zuerst wird der Vektor zwischen dem Start- und Endpunkt berechnet, dann wird dieser mit dem Verhältnis der Fahrzeugposition zur Gesamtlänge der Strasse skaliert. Der verkürzte Vektor wird anschliessend zum Startvektor addiert, somit erhält man die Position des Fahrzeuges in zweidimensionalen Koordinaten.

Nach jedem Update-Intervall wird mit dieser Methode für alle Fahrzeuge in der Strasse die neue Position berechnet und jeweils im Fahrzeug unter `screen_pos` gespeichert.

2.6.3 Generator für neue Fahrzeuge

Bis anhin hatte ich die Fahrzeuge am Ende des Programms manuell erzeugt. Um dies zu automatisieren, erstellte ich die Methode `generate_vehicles`:

```
def generate_vehicles(self):
    'Provisorischer Fahrzeug-Generator'
    if self.vehicles[0].position - self.vehicles[0].lenght > 50:
        if random.choice([0, 0, 0, 0, 0, 0, 0, 0, 0, 1]):
            type = 'truck'
        else:
            type = 'car'
        self.add_vehicle(Vehicle(type=type, v=80/3.6))
```

Code 8: Provisorische Methode generate_vehicles

Die Methode stellt sicher, dass das erste Fahrzeug in der Liste mindestens 50 Meter entfernt ist. Ist dies der Fall, wird zufällig der Fahrzeugtyp `'car'` oder `'truck'` (in diesem Fall mit Wahrscheinlichkeit $^1/_{11}$) ausgewählt. Anschliessend wird ein neues Objekt der `class Vehicle` mit dem entsprechenden Fahrzeugtyp und einer Geschwindigkeit von 80 km/h initiiert. Dar Fahrzeugtyp wird hier noch `type` genannt. Später habe ich ihn zu `vehicle_type` umbenannt, da `type` eine systeminterne Variable ist, die nicht neudefiniert werden sollte. Für alle Grössen werden in der Simulation SI-Einheiten verwendet, deshalb müssen Geschwindigkeiten in km/h in die Einheit m/s umgerechnet werden.

2.6.4 Mehrere Fahrspuren auf einer Strasse

Das nächste Ziel war die Implementierung einer Strasse mit mehreren Fahrspuren in beide Richtungen. Mir fiel auf, dass die bisherige `class Road` ziemlich genau den Funktionsumfang der gewünschten Class für Fahrspuren hatte, deshalb wurde sie kurzerhand in `class Lane` umbenannt. Aufgrund der Flexibilität des objektorientierten Programmierens ist eine solche Umbenennung mit sehr wenig Aufwand verbunden. In der neuen `class Road` sollten alle Fahrspuren einer Strasse gespeichert werden. Zusätzlich sollte sie Methoden des Hinzufügens von neuen Spuren und des Spurwechsels beinhalten.

Zur besseren Unterscheidung zwischen den beiden Fahrtrichtungen entschied ich mich dazu, die Speicherung der Fahrspuren in zwei Listen aufzuteilen: `self.lanes_0` und `self.lanes_1`. Die Zahl 0 bedeutet «entlang der Strasse», d. h. von `pos_start` bis `pos_end`, die Zahl 1 «in die entgegengesetzte Richtung».

Gewisse Parameter, die für alle Fahrspuren einer Strasse gleich sind, werden in der Strassen-Class festgelegt und an die Fahrspuren weitergegeben. Dazu gehören Länge, Höchstgeschwindigkeit und Winkel. Winkel sind zu diesem Zeitpunkt noch nicht von Bedeutung, werden aber später thematisiert.

Hinzufügen von neuen Fahrspuren

Die grösste Schwierigkeit beim Hinzufügen einer neuen Fahrspur ist die genaue Bestimmung von deren Anfangs- und Endposition. Die Positionen können nicht einfach von der

Strasse übernommen werden, da sonst alle Spuren in der Strassenmitte aufeinanderliegen würden. Stattdessen müssen sie um die Breite der Spur verschoben werden.

Die Methode `add_lane` erstellt die neue Fahrspur mit den angegebenen Parametern und speichert sie je nach angegebener Richtung in `self.lanes_0` oder `self.lanes_1`. Zur Berechnung von `pos_start` und `pos_end` der Fahrspur benutzt sie die Methode `get_lane_pos`. Diese wiederum verwendet die Methode `offset`.

```
def offset(self, position: tuple, offset: float) -> tuple:
    'Verschiebung einer Position senkrecht zur Strassenmitte'
    posx = position[0] + offset * math.sin(self.angle)
    posy = position[1] + offset * math.cos(self.angle)
    return (posx, posy)

def get_lane_pos(self, lane) -> tuple:
    'Berechnung der Position einer Fahrspur'
    if lane in self.lanes_0:
        offset = lane.width / 2
        for other_lane in self.lanes_0:
            if other_lane == lane:
                break
            offset += other_lane.width
        pos_start = self.offset(self.pos_start, offset)
        pos_end = self.offset(self.pos_end, offset)
    elif lane in self.lanes_1:
        offset = -lane.width / 2
        for other_lane in self.lanes_1:
            if other_lane == lane:
                break
            offset -= other_lane.width
        pos_start = self.offset(self.pos_end, offset)
        pos_end = self.offset(self.pos_start, offset)
    else:
        raise ValueError('Fahrspur gehört nicht zu dieser Strasse!')
    return (pos_start, pos_end)
```

Code 9: Berechnung der Position einer neuen Fahrspur

Neue Fahrspuren werden immer am Ende der Liste gespeichert. Daher werden mit einer for-Schleife die Breiten der anderen Fahrspuren als `offset` zusammengezählt, bis die neue Fahrspur erreicht wird, dann wird die Schleife mit `break` abgebrochen. Die Hälfte der Breite der neuen Spur wird noch dazugezählt, da die neuen Start- und Endpositionen genau in der Mitte der Spur liegen sollen. Wenn die Richtung der Spur 1 ist, also entgegengesetzt der Strassenrichtung, werden die Start- und Endpositionen vertauscht und der `offset` negativ gemacht.

Die Methode `offset` verschiebt die Start- und Endpositionen, die in der Strasse gespeichert sind, um den angegebenen Wert. Dafür verwendete ich Sinus und Cosinus des Strassenwinkels, da der Winkel zu diesem Zeitpunkt bereits implementiert war. Das Vorzeichen beim Sinus müsste eigentlich negativ sein, allerdings zeigt bei meinem Koordinatensystem die y-Achse nach unten. Der Grund dafür wird in Kapitel 2.10.3 erklärt.

2.6.5 Spurwechsel

Mehrspurige Strassen sind nur dann sinnvoll, wenn Fahrzeuge auch die Fahrspur wechseln können. Bei einem Spurwechsel muss ein Fahrzeug aus seiner aktuellen Spur entfernt und an der richtigen Stelle in die benachbarte Spur eingesetzt werden. Es muss auch eine Möglichkeit geben zu prüfen, ob ein Spurwechsel überhaupt sinnvoll ist. Zuerst beschäftigte ich mich aber damit, den Spurwechsel überhaupt zu ermöglichen.

Methode für den Spurwechsel

```
def change_lanes(self, vehicle, direction: int=1):
    'Spurwechsel für ein Fahrzeug'
    current_lane = vehicle.current_lane
    if current_lane in self.lanes_0:
        lanes = self.lanes_0
    elif current_lane in self.lanes_1:
        lanes = self.lanes_1

    index = lanes.index(current_lane) + direction
    if index < 0:
        raise IndexError
    new_lane = lanes[index]
    current_lane.remove_vehicle(vehicle)
    try:
        new_lane.add_vehicle(vehicle)
    except ValueError as exc:
        current_lane.add_vehicle(vehicle)
        raise ValueError('Fahrzeug passt nicht in die Lücke!') from exc
```

Code 10: Methode change_lanes für den Spurwechsel

Um den Spurwechsel zu ermöglichen, erstellte ich in der `class Road` die Methode `change_lanes`. Zuerst wird herausgefunden, in welcher Liste sich die aktuelle Fahrspur des Fahrzeuges (`vehicle.current_lane`) befindet. Der Index der aktuellen Fahrspur wird dann mit der angegebenen Richtung `direction` addiert. Da die Fahrspuren in der Liste von links nach rechts sortiert sind, bedeutet die Richtung 1 einen Spurwechsel nach rechts und die Richtung -1 einen Spurwechsel nach links. Theoretisch wären auch Zahlen grösser als 1 möglich, dann würden allerdings Spuren übersprungen werden. Wenn die Spur mit dem neuen Index nicht existiert, tritt ein `IndexError` auf. Dieser wird bewusst nicht aufgefangen, da der Spurwechsel in diesem Fall nicht möglich ist. Beim Testen merkte ich, dass negative Indizes keinen Fehler hervorrufen, sondern vom letzten Element der Liste herabzählen. Das war in diesem Fall ein Problem, da ohne Fehlermeldung von der ganz linken in de ganz rechte Spur gewechselt wurde. Um dies zu unterbinden, fügte ich eine Bedingung ein, sodass bei einem negativen Index manuell ein `IndexError` hervorgerufen wird. Nachdem die neue Fahrspur bestimmt worden ist, wird das Fahrzeug aus seiner aktuellen Fahrspur entfernt und es wird versucht, das Fahrzeug zur neuen Fahrspur hinzuzufügen. Wenn dies nicht funktioniert, d. h., wenn die Funktion `new_lane.add_vehicle(vehicle)` einen Fehler `ValueError` erzeugt, wird das Fahrzeug wieder auf seine ursprüngliche Spur zurückversetzt und es wird eine neue Fehlermeldung erzeugt.

Einfügen des Fahrzeuges an richtiger Stelle

Die Methode `add_vehicle` existierte bereits in der `class Lane`, jedoch wurden neue Fahrzeuge immer am Anfang der Liste und mit der Position 0 eingefügt. Die neuen Anforderungen an diese Methode waren, dass Fahrzeuge an jeder beliebigen Position eingefügt werden können und eine Fehlermeldung herausgegeben wird, wenn sich an dieser Position bereits ein Fahrzeug befindet. Dies habe ich folgendermassen umgesetzt:

```
def add_vehicle(self, vehicle):
    'Hinzufügen eines neuen Fahrzeuges zur Fahrspur'
    last_vehicle = None
    for other_vehicle in self.vehicles:
        if other_vehicle.position - other_vehicle.lenght >= vehicle.position:
            self.vehicles.insert(self.vehicles.index(other_vehicle), vehicle)
            break
        last_vehicle = other_vehicle
    if vehicle not in self.vehicles:
        self.vehicles.append(vehicle)
    if last_vehicle and last_vehicle.position > vehicle.position-vehicle.lenght:
        self.vehicles.remove(vehicle)
        raise ValueError('Fahrzeug passt nicht auf die Fahrspur!')

    vehicle.current_lane = self
    vehicle.angle = self.angle
    vehicle.change_speed_limit(self.speed_limit)
```

Code 11: Einfügen eines Fahrzeuges an richtiger Stelle einer Fahrspur

Der Hauptbestandteil der Methode ist eine for-Schleife, die alle Fahrzeuge in der Liste `self.vehicles` durchgeht. Sobald ein anderes Fahrzeug mit einer grösseren Position als das neu einzufügende Fahrzeug erreicht wird, wird die Schleife mit `break` unterbrochen und das neue Fahrzeug an dieser Stelle in die Liste eingesetzt. Wichtig ist, dass von der Position des vorausfahrenden Fahrzeuges noch dessen Länge abgezogen wird. Danach wird überprüft, ob das neue Fahrzeug bereits in die Liste eingefügt werden konnte. Falls nicht, wird es am Ende der Liste angehängt. In diesem Fall wäre es das vorderste Fahrzeug in der Kolonne. Schliesslich muss noch überprüft werden, ob sich das neue Fahrzeug nicht mit dem hinterherfahrenden überlappt. Sonst wird es wieder aus der Liste entfernt und ein `ValueError` erzeugt.

2.6.6 Spurwechselmodell MOBIL

Die Entscheidung, ob ein Spurwechsel durchgeführt wird, kann mit dem Spurwechselmodell MOBIL (**M**inimizing **O**verall **B**raking decelerations **I**nduced by **L**ane changes)[11] getroffen werden. Für die Entscheidung müssen nicht nur die Fahrzeuge auf der aktuellen Spur, sondern alle benachbarten Fahrzeuge berücksichtigt werden. Das Spurwechselmodell hat zwei Kriterien, die beide erfüllt werden müssen, damit der Spurwechsel stattfindet. Mit der folgenden Grafik soll dies etwas besser illustriert werden:

[11] Treiber, Martin; Kesting, Arne; Helbing, Dirk: *MOBIL: General Lane-Changing Model for Car-Following Models*. Dresden, 2006. Auf: https://mtreiber.de/publications/MOBIL_TRB.pdf (abgerufen am 14.10.2024).

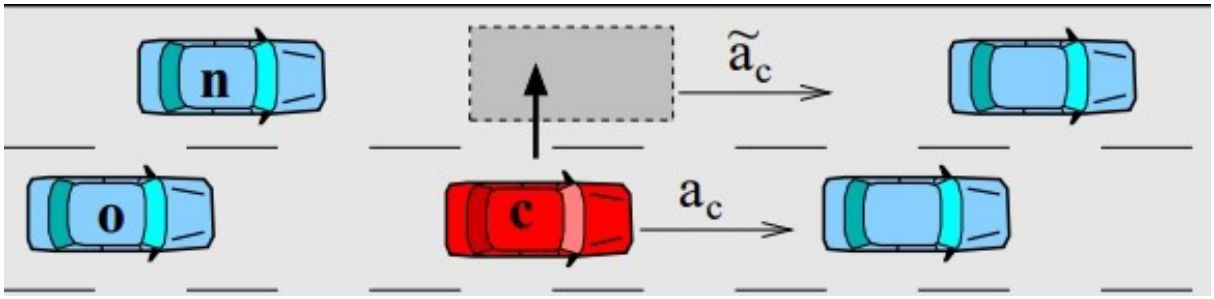


Abbildung 1: Darstellung eines Fahrzeuges c, das einen Spurwechsel nach links in Betracht zieht. Ebenfalls dargestellt sind die alten und neuen nachfolgenden Fahrzeuge o resp. n und die Beschleunigungen von c vor und nach dem Spurwechsel.

Sicherheitskriterium

Das Sicherheitskriterium stellt sicher, dass das nachfolgende Fahrzeug auf der neuen Spur die gegebene sichere Bremsbeschleunigung b_{safe} nicht überschreitet. Dies kann durch folgende mathematische Ungleichung formuliert werden:

$$a'_n \geq -b_{safe}$$

Alle Beschleunigungen a, die in diesem Modell verwendet werden, werden mit dem IDM berechnet. Theoretisch wäre aber auch die Nutzung eines anderen Fahrzeugfolgemodells möglich. Durch das Sicherheitskriterium werden spurwechselbedingte Kollisionen ausgeschlossen. Wenn der Abstand zum neuen hinteren Fahrzeug n zu klein ist, würde dieses nach dem Spurwechsel zu stark abbremsen und somit das Sicherheitskriterium nicht erfüllen.

Anreizkriterium

Das Anreizkriterium überprüft, ob ein Spurwechsel die individuelle Situation des Fahrzeuges c verbessert, d. h. ob es danach eine grössere Beschleunigung hat. Zusätzlich werden noch die direkt betroffenen Nachbarn o und n berücksichtigt. Der «Höflichkeitsfaktor» p bestimmt, wie stark die nachfolgenden Fahrzeuge die Entscheidung beeinflussen sollen. Die Gleichung des Anreizkriteriums sieht folgendermassen aus:

$$a'_c - a_c + p(a'_n - a_n + a'_o - a_o) > \Delta a_{th}$$

Für jedes Fahrzeug wird die Differenz der Beschleunigung vor und nach dem Spurwechsel ausgerechnet, wobei die Beschleunigungen der beiden nachfolgenden Fahrzeuge mit p gewichtet werden. Realistische Werte für p liegen zwischen 0 und 1, wobei 0 ein vollständig egoistisches Fahrverhalten bedeuten würde. Jedoch muss auch bei egoistischem Fahrverhalten immer noch das Sicherheitskriterium eingehalten werden. Das Anreizkriterium ist erfüllt, wenn der gesamte Beschleunigungsgewinn grösser als ein gewisser Schwellenwert Δa_{th} ist. Der Schwellenwert verhindert, dass wegen minimalen Beschleunigungsunterschieden zwischen den Spuren hin- und hergewechselt wird.

Rechtsfahrgebot

Im Gegensatz zu den USA ist in der Schweiz (und in den meisten europäischen Ländern) die Nutzung der Fahrspuren asymmetrisch geregelt, d. h. es gilt das Rechtsfahrgebot. Die linke Spur soll also nur zum Überholen genutzt werden. Dies kann durch die Einführung einer Tendenz Δa_{bias} bewerkstelligt werden. Die Tendenz sollte einen Spurwechsel von links nach rechts begünstigen und in die umgekehrte Richtung erschweren. Des Weiteren ist es sinnvoll, das nachfolgende Fahrzeug o bei einem Spurwechsel von rechts nach links nicht zu berücksichtigen, da dieses fast immer eine positive Beschleunigungsdifferenz hat und deshalb den Spurwechsel begünstigen würde. Schliesslich sollten langsame Fahrzeuge auf der rechten Spur nicht durch schnellere Fahrzeuge auf die linke Spur «verdrängt» werden. Daraus lässt sich das Anreizkriterium folgendermassen anpassen:

$$L \to R: \quad a'_c - a_c + p(a'_n - a_n + a'_o - a_o) > \Delta a_{th} - \Delta a_{bias}$$

$$R \to L: \quad a'_c - a_c + p(a'_n - a_n) > \Delta a_{th} + \Delta a_{bias}$$

Diese Anpassung sorgt grundsätzlich dafür, dass langsame Fahrzeuge auf der rechten Spur bleiben, während links überholt wird. Sie verhindert jedoch nicht das Rechtsüberholen, was in der Schweiz aber mittlerweile bei stockendem Verkehr erlaubt ist.

2.6.7 Implementierung der Spurwechselkriterien

Zur Umsetzung der Kriterien erstelle ich in der Strassen-Class die Methode `check_lane_change`, die leider zu lang ist, um hier abgebildet zu werden. Als Argumente müssen das Fahrzeug und die Richtung des Spurwechsels (1 oder -1) angegeben werden. Die Methode sollte dann einen Wert des Typs `bool`, also entweder `True` oder `False`, zurückgeben. Die Schwierigkeit war nicht, die Ungleichungen des Sicherheits- und Anreizkriteriums in Python-Syntax zu übersetzen, sondern die Beschleunigungen der Fahrzeuge c, o und n vor und nach dem Spurwechsel zu bestimmen. Insgesamt mussten also sechs Beschleunigungen bestimmt werden.

Berechnungen aller Beschleunigungen

Ich kam auf die Idee, den Spurwechsel für das Fahrzeug c einmal durchzuführen, um die Beschleunigungswerte nach dem Spurwechsel zu bestimmen. Danach kann der Spurwechsel wieder rückgängig gemacht werden, indem die Richtung einfach umgekehrt wird. Somit wird das Fahrzeug wieder in seinen ursprünglichen Zustand versetzt. Wenn die Funktion `self.change_lanes(c, direction)` einen Fehler erzeugt, d. h., wenn das Fahrzeug nicht in die Lücke der neuen Spur passt oder diese nicht existiert, kann direkt `False` ausgegeben werden.

Zur Bestimmung der nachfolgenden Fahrzeuge o und n wurde in der `class Lane` die Methode `last_vehicle` erstellt. Diese funktioniert genau gleich wie die Methode `next_vehicle`, ausser dass der Index des Fahrzeuges um 1 verringert statt vergrössert wird. Hier musste auch wieder aufgepasst werden, dass keine negativen Indizes auftreten können.

Alle Beschleunigungen werden mit der Methode `acceleration` des jeweiligen Fahrzeug-Objektes berechnet. Dafür wird jeweils das nächste Fahrzeug benötigt, welches mit der bereits beschriebenen Methode `next_vehicle` bestimmt wird. Falls ein Fahrzeug (o und n) nicht existiert, wird seine Beschleunigung gleich 0 gesetzt. Somit hat es keinen Einfluss auf die Entscheidung.

Der gesamte Ablauf kann wie folgt zusammengefasst werden:

1. Berechnung der Beschleunigungen a_c und a_o
2. Wechsel auf die neue Spur. Falls dabei ein Fehler auftritt, wird bereits hier mit `return False` abgebrochen und wieder zurück gewechselt.
3. Berechnung der Beschleunigungen a'_c, a'_n und a'_o
4. Wechsel zurück auf die alte Spur
5. Berechnung der Beschleunigung a_n
6. Anwendung der Kriterien und Ausgabe von `True` oder `False`

Festlegen der Modellparameter

Für die Wahl der Modellparameter orientierte ich mich an der Tabelle von Treiber, Kesting und Helbing[12], da dort ebenfalls das IDM als Fahrzeugfolgemodell verwendet wird.

Wie bereits beschrieben sollte die Tendenz Δa_{bias} die rechte Spur begünstigen, deshalb muss sie beim Spurwechsel von links nach rechts negativ und in die umgekehrte Richtung positiv sein. Dies lässt sich durch `bias = -0.4 * direction` sehr einfach realisieren. Es wurde ein Wert von 0.4 m/s² gewählt. Im späteren Verlauf fügte ich noch weitere Bedingungen ein, um den Wechsel auf die richtige Spur vor dem Abbiegen zu ermöglichen (dazu folgt später mehr).

Der Höflichkeitsfaktor p ist ein individueller Wert, der für jeden Fahrer unterschiedlich sein kann. Ich empfand es daher als sinnvoll, diesen zufällig in der Methode `__init__` der Fahrzeug-Class zu bestimmen. Ausgewählt werden Werte zwischen 0.3 und 0.7.

Für den Schwellenwert Δa_{th} wählte ich 0.2 m/s² und für die sichere Bremsbeschleunigung b_{safe} 4 m/s².

Überprüfen der Kriterien

Im letzten Teil der Methode werden die beiden Kriterien überprüft:

```
# Überprüfung der Beschleunigungskriterien
safety_criterion = new_acc_n >= -safe_deceleration
if bias <= 0:
    total_acc = ((new_acc_c - acc_c)
        + politeness * (new_acc_n - acc_n + new_acc_o - acc_o)
        - threshold - bias)
else:
    total_acc = ((new_acc_c - acc_c)
        + politeness * (new_acc_n - acc_n)
        - threshold - bias)
```

[12] Treiber et al. 2006, S. 10.

```
incentive_criterion = total_acc > 0
if safety_criterion and incentive_criterion:
    return total_acc
else:
    return False
```

Code 12: Überprüfung der Sicherheits- und Anreizkriterien des Spurwechselmodells

Das Sicherheitskriterium wird zuerst berechnet, danach das Anreizkriterium. Wenn der Bias positiv ist, werden die Beschleunigungen `acc_o` und `new_acc_o` weggelassen. Somit wird das zuvor beschriebene Rechtsfahrgebot eingehalten. Die Parameter `threshold` und `bias` wurden auf die linke Seite der Ungleichung verschoben, damit sie auch im totalen Beschleunigungsgewinn `total_acc` enthalten sind. Am Ergebnis der Entscheidung ändert diese Anpassung nichts. Wenn der Beschleunigungsgewinn grösser als 0 ist, ist das Anreizkriterium erfüllt. Wenn das Sicherheitskriterium auch erfüllt ist, wird der Beschleunigungsgewinn als Return Value zurückgegeben, ansonsten wird `False` zurückgegeben.

Überprüfen des Spurwechsels in beide Richtungen

Mit der Methode `check_lane_change` kann entschieden werden, ob ein Spurwechsel in eine bestimmte Richtung durchgeführt werden soll. Bei mehr als zweispurigen Strassen müssen jedoch beide Richtungen untersucht und der Spurwechsel allenfalls durchgeführt werden. Dafür erstellte ich eine weitere Methode, die `check_lane_changes` genannt wurde. Diese überprüft die beiden Spurwechselmöglichkeiten nach links und rechts einzeln mit `self.check_lane_change`. Wenn der Output für eine Richtung nicht `False` ist, wird der Spurwechsel in diese Richtung mit `self.change_lanes` ausgeführt. Wenn beide Richtungen möglich sind, wird diejenige mit dem grösseren Beschleunigungsgewinn ausgewählt.

Zusätzlich führte ich noch den Timer `vehicle.lane_change_timer` ein, der sicherstellt, dass ein Spurwechsel bei Autos nur alle 2 Sekunden und bei Lastwagen alle 3 Sekunden stattfinden kann. Der Timer wird bei jedem Update-Intervall um die entsprechende Länge des Intervalls vergrössert und bei einem Spurwechsel wieder auf 0 zurückgesetzt. Er ist realistisch gesehen sinnvoll, da Fahrzeuge eine bestimmte Zeit benötigen, um die Spur zu wechseln und erst danach den nächsten Spurwechsel in Betracht ziehen können.

2.7 Kreuzungen und Strassennetzwerke

Die bisher beschriebenen Classes für Fahrzeuge, Fahrspuren und Strassen erlauben zwar die gleichzeitige Simulation mehrerer Strassen, jedoch sind diese nicht miteinander verbunden. Fahrzeuge können somit auch nicht von einer Strasse auf die nächste wechseln. Hier kommen Kreuzungen als «Bindeglied» zwischen den Strassen ins Spiel. Mehrere Kreuzungen, die mit Strassen verbunden sind, bilden ein Strassennetz. Die Hauptaufgabe der Kreuzungen besteht darin, einfahrende Fahrzeuge aufzunehmen und auf eine nächste Strasse weiterzuleiten.

2.7.1 Class für Kreuzungen

Auch für Kreuzungen erstellte ich eine Class, die `class Intersection` genannt wurde. Die erste Version dieser Class konnte nicht viel bewirken. Sie hatte lediglich eine Position, eine Liste aller Strassen, mit der sie verbunden ist und eine Liste aller Fahrzeuge, die sich in dem Moment auf der Kreuzung befinden. Mein Ziel war dabei nicht, die Fahrzeuge mit einer komplizierten Kurve durch die Kreuzung fahren zu lassen, sondern nach dem Hinzufügen zur Kreuzung direkt auf die nächste Strasse weiterzuleiten. Die Liste der Strassen teilte ich später in zwei Listen auf: `self.incoming_lanes` und `self.outgoing_lanes`. Dies aus dem einen Grund, um klar zu unterscheiden, welche Fahrspuren in die Kreuzung hineinführen und welche aus der Kreuzung heraus. Beispielsweise hat eine Einbahnstrasse nur eine Spur, die entweder bei der Kreuzung anfängt oder dort endet.

In der Class erstellte ich eine Methode `add_vehicle`, die ein neues Fahrzeug in der Liste `self.vehicles` speichert. Mit der Methode `remove_vehicle` sollten die Fahrzeuge direkt wieder entfernt und an die nächste Fahrspur weitergegeben werden. Dafür musste jedoch zuerst eine Möglichkeit gefunden werden, um die nächste Fahrspur für ein Fahrzeug zu bestimmen. Um dies überhaupt zu ermöglichen, muss das Fahrzeug ein bestimmtes Ziel haben. Dann muss ein Algorithmus den kürzesten Weg dorthin berechnen.

2.7.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus[13] ist vermutlich die einfachste Methode zur Lösung des Problems des kürzesten Pfades zwischen zwei unterschiedlichen Knoten in einem gewichteten Graphen. Ein Graph ist eine Ansammlung von Knoten, die mit Kanten verbunden sind. Die Kanten können gewichtet sein, d. h. einem bestimmten Wert zugeordnet sein. Das Strassennetz kann als Graph betrachtet werden, wobei die Kreuzungen Knoten und die Strassen Kanten sind. Die Kantengewichte könnten beispielsweise die Fahrzeiten zwischen den Kreuzungen darstellen. Beim Dijkstra-Algorithmus dürfen diese nicht negativ sein, was in meinem Fall aber kein Problem darstellt. Der kürzeste Weg von A nach B ist folglich der mit der kleinsten Summe der Kantengewichte. Der Algorithmus berechnet dabei nicht nur den kürzesten Weg von A nach B, sondern von A zu allen anderen Punkten/Knoten. Es kann mathematisch bewiesen werden, dass dieser Algorithmus in jedem Fall den kürzesten Weg findet. Darauf wird in dieser Arbeit aber nicht weiter eingegangen.

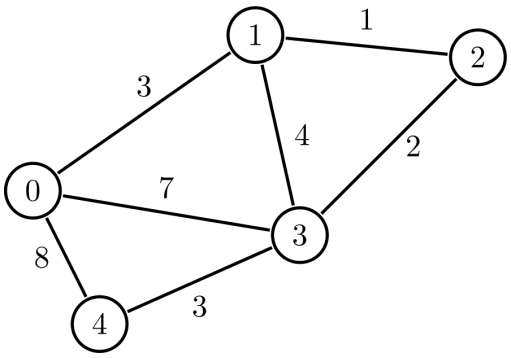


Abbildung 2: Beispiel eines gewichteten Graphen

[13] Floss, Theresa: *Die Mathematik hinter einem Navigationssystem – Wie Routenplaner ihre Wege berechnen*. Innsbruck, 2023, S. 26 ff. Auf: https://www.uibk.ac.at/mathematik/algebra/media/teaching/bachelorarbeit_floss_mathematik_navigation.pdf (abgerufen am 16.10.2024).

Ablauf des Algorithmus

Zuerst werden zwei Listen erstellt: eine mit allen unbesuchten Kreuzungen und eine mit allen besuchten Kreuzungen. Die Liste der besuchten Kreuzungen ist am Anfang noch leer, während die Liste der unbesuchten Kreuzungen alle Kreuzungen der Simulation enthält. Für jede Kreuzung werden zwei Variablen definiert: die bisher berechnete kürzeste Fahrzeit dorthin und der Vorgänger. Erstere wird zu Beginn für alle Kreuzungen auf unendlich gesetzt, ausser für die Startkreuzung A, wo sie auf 0 gesetzt wird. Die folgenden Schritte werden dann so lange wiederholt, bis die Zielkreuzung B besucht wird:

1. Wähle aus der Liste der unbesuchten Kreuzungen die Kreuzung C mit der kürzesten Fahrzeit.
2. Finde alle benachbarten Kreuzungen von C.
3. Berechne für jede benachbarte Kreuzung die neue Fahrzeit ausgehend von der Fahrzeit der Kreuzung C. Falls diese kleiner ist als die bisherige Fahrzeit der Nachbarkreuzung, überschreibe die alte Fahrzeit mit der neuen und definiere C als Vorgänger.
4. Verschiebe die Kreuzung C von der Liste der unbesuchten Kreuzungen in die Liste der besuchten Kreuzungen.

Sobald die Zielkreuzung B erreicht worden ist, kann der Weg von A nach B über die Vorgänger zurückverfolgt und in einer Liste gespeichert werden.

Rechenbeispiel

Als Beispiel betrachten wir den Graphen in Abbildung 2. Ziel ist es, den kürzesten Weg von Knoten (2) zu Knoten (4) zu finden. Dazu wird der Ablauf Schritt für Schritt ausgeführt:

1. Für den Startknoten (2) wird die Fahrzeit auf 0 gesetzt, alle anderen Knoten erhalten die Fahrzeit unendlich.
2. Die Nachbarknoten von (2) sind (1) und (3). Die neuen Fahrzeiten für diese Knoten sind 0+1=1 resp. 0+2=2. Bei beiden Knoten sind die neuen Fahrzeiten kleiner als die aktuelle Fahrzeit (unendlich), daher werden sie mit den neuen Fahrzeiten überschrieben. Nun kann (2) aus der Liste der unbesuchten Knoten entfernt werden.
3. Als Nächstes wird Knoten (1) betrachtet, da dieser die kleinste Fahrzeit von allen unbesuchten Knoten hat. Der Nachbarknoten (0) erhält die neue Fahrzeit 1+3=4. Für Knoten (3) wäre die neue Fahrzeit 1+4=5, diese ist allerdings grösser als die bisherige Fahrzeit von 2 und kann daher ignoriert werden. Ebenso beim Nachbarknoten (2).
4. Nun ist Knoten (3) mit einer Fahrzeit von 2 das kleinste Element in der Liste. Von hier aus hat Knoten (4) eine Wegzeit von 2+3=5. Die anderen Nachbarknoten von (3) können über diese Route nicht schneller erreicht werden.
5. Knoten (0) hat eine Fahrzeit von 4. Für den Nachbarknoten (4) wäre die neue Fahrzeit 4+8=12, grösser als die bisherige Fahrzeit 5.
6. Es bleibt nur noch der Zielknoten (4) mit der Fahrzeit 5 übrig. An dieser Stelle kann der Algorithmus abbrechen. Der kürzeste Weg wird anhand der Vorgänger zurückverfolgt (diese wurden in der Beschreibung ausgelassen). Er führt von (2) über (3) nach (4).

2.7.3 Implementierung des Dijkstra-Algorithmus

Ich entschied mich dazu, die Methode für den Dijkstra-Algorithmus in der neuen `class Simulation` zu erstellen, da dort alle Strassen und Kreuzungen gespeichert werden. Den Code schrieb ich selbst, obwohl im Internet einige vorprogrammierte Versionen von diesem Algorithmus verfügbar sind.

```
def path_finder(self, start, end, current_lane) -> list:
    'Dijkstra-Algorithmus zur Bestimmung der schnellsten Route vom Start zum Ziel'
    unvisited: list = self.intersections.copy()
    found_path: bool = False
    # Definition der Variablen Fahrzeit und Vorgänger für jede Kreuzung
    for intersection in unvisited:
        intersection.predecessor = None
        if intersection == start:
            intersection.travel_time = 0
        else:
            intersection.travel_time = math.inf

    while unvisited:
        # Auswahl der Kreuzung mit der kürzesten Fahrzeit
        unvisited.sort(key=lambda x: x.travel_time)
        current_intersection = unvisited[0]
        # Abbruchkriterien
        if current_intersection.travel_time == math.inf:
            return
        if current_intersection == end:
            found_path = True
            break
        # Bestimmung der aktuellen Fahrspur
        if current_intersection.predecessor:
            for lane in current_intersection.incoming_lanes:
                if lane.start_connection == current_intersection.predecessor:
                    current_lane = lane
                    break

        # Vergleichen der Fahrzeiten aller Nachbarkreuzungen
        for neighbour_intersection in (current_intersection.get_neighbour_intersections(
            current_lane)):
            old_time = neighbour_intersection[0].travel_time
            new_time = current_intersection.travel_time + neighbour_intersection[1]
            if new_time < old_time:
                neighbour_intersection[0].travel_time = new_time
                neighbour_intersection[0].predecessor = current_intersection
        # Entfernen der aktuellen Kreuzung aus der Liste
        unvisited.remove(current_intersection)
        current_lane = None

    # Zurückverfolgen der berechneten Fahrtroute
    if found_path:
        path: list = []
        while current_intersection.predecessor:
            path.insert(0, current_intersection)
            current_intersection = current_intersection.predecessor
        return path
```

Code 13: Methode für den Dijkstra-Algorithmus

Die Methode benötigt die zwei Argumente `start` und `end`, die beide Objekte des Typs `Intersection` sein müssen. Das dritte Argument `current_lane` ist zu diesem Zeitpunkt noch nicht wichtig, wird jedoch später wieder aufgegriffen.

Im ersten Schritt wird die Liste `unvisited` erstellt, indem eine Kopie der Liste `self.intersections` angelegt wird. Auf eine Liste aller besuchten Kreuzungen wurde verzichtet, da diese nicht zwingend benötigt wird. Dann werden für jede Kreuzung die zuvor beschriebenen Variablen Fahrzeit und Vorgänger definiert. Die Unendlichkeit kann in Python durch `math.inf` ausgedrückt werden.

Die Schleife `while unvisited` wiederholt die zuvor beschriebenen Schritte 1-4 des Algorithmus so lange, bis die Liste `unvisited` leer oder ein Abbruchkriterium erfüllt ist. Zur Bestimmung der aktuellen Kreuzung `current_intersection` (Schritt 1) wird die Liste nach Fahrzeit sortiert und anschliessend das erste Element ausgewählt. Die benachbarten Kreuzungen werden mit `current_intersection.get_neighbour_intersections` bestimmt (Schritt 2). Diese Methode habe ich in der `class Intersection` angelegt. Sie erzeugt eine Liste von Tupeln mit allen benachbarten Kreuzungen und den entsprechenden Fahrzeiten, um diese zu erreichen. Wie die Fahrzeiten genau berechnet werden, wird im nächsten Kapitel thematisiert. Schritt 3 kann ganz einfach mit einer for-Schleife durchgeführt werden. Bei Schritt 4 wird die Kreuzung schliesslich aus der Liste `unvisited` entfernt.

Zum Schluss wird noch der Pfad zurückverfolgt, sofern er überhaupt gefunden werden konnte. Dies ist der Fall, wenn die Variable `found_path` den Wert `True` hat. Der Fall, dass kein Pfad gefunden wird, kann nur auftreten, wenn die Zielkreuzung nicht mit dem Strassennetzwerk verbunden ist.

2.7.4 Berechnung der Fahrzeit

Die benötigte Fahrzeit für eine Strasse kann ungefähr abgeschätzt werden, indem ihre Länge durch ihre Höchstgeschwindigkeit geteilt wird. Dies war auch mein erster Ansatz, den ich so in der Methode `get_travel_time` der `class Lane` implementierte. Der Ansatz geht jedoch davon aus, dass die Fahrzeuge mit der entsprechenden Höchstgeschwindigkeit fahren. Bei einem Stau ist dies nicht der Fall. Intelligente Fahrer würden in diesem Fall versuchen, mit einer alternativen Route den Stau zu umfahren. Dieses Verhalten wollte ich auch in meine Simulation einbauen. Die eben beschriebene Methode sollte deshalb erweitert werden, sodass sie die aktuelle durchschnittliche Fahrzeit der Fahrzeuge auf der Fahrspur berechnen kann. Nach langem Herumprobieren kam ich auf folgendes Ergebnis:

```
def get_travel_time(self) -> float:
    'Berechnung der benötigten Zeit, um die Strasse zu befahren'
    t1 = self.lenght / self.speed_limit
    try:
        last_vehicle = self.vehicles[-1]
        t2 = last_vehicle.travel_time_counter
    except IndexError:
        t2 = 0
    if self.travel_times:
```

```
        t3 = 0
        for time in self.travel_times:
            t3 += time[0]
        t3 /= len(self.travel_times)
    else:
        t3 = 0
    t = max(t1, t2, t3)
    return t
```

Code 14: Berechnung der benötigten Fahrzeit für eine Fahrspur

Es werden drei unterschiedliche Fahrzeiten t_1, t_2 und t_3 berechnet und danach die grösste davon ausgewählt. t_1 ist die eben beschriebene Schätzung der Fahrzeit bei Höchstgeschwindigkeit, t_2 ist die aktuelle Fahrzeit des vordersten Fahrzeuges, das sich noch auf der Fahrspur befindet und t_3 ist die durchschnittliche Fahrzeit aller Fahrzeuge, die die Strasse in den letzten fünf Minuten verlassen haben. Die Daten für t_3 stammen aus der Liste `self.travel_times`, die in der Methode `update` laufend aktualisiert wird. Dabei werden alle Daten entfernt, die älter als fünf Minuten sind. Somit wird sichergestellt, dass die durchschnittliche Fahrzeit nach der Auflösung eines Staus wieder gesenkt wird und der Dijkstra-Algorithmus diese Route wieder bevorzugt.

2.7.5 Abzweigungsrichtungen

Vor Kreuzungen gibt es häufig mehrere Einspurstreifen, von denen aus man in nur eine bestimmte Richtung abbiegen darf, d. h. nach links, rechts oder geradeaus. Ebenso gibt es Kreuzungen, wo das Abbiegen nicht in alle Richtungen erlaubt ist. Für den Stadtverkehr sind diese Abzweigungsrichtungen sehr wichtig, deshalb mussten sie auch in meine Simulation eingebaut werden.

In der Class für Fahrspuren erstellte ich die Variable `self.turning_directions`, die standardmässig den Wert `[-1, 0, 1]` hat. Jede Zahl steht dabei für eine Abzweigungsrichtung (-1 für links, 0 für geradeaus und 1 für rechts), ähnlich wie die Richtungen beim Spurwechsel. Der Standardwert bedeutet also, dass man von dieser Fahrspur aus in alle Richtungen abbiegen darf. Werden eine oder zwei Zahlen aus der Liste entfernt, sind die erlaubten Abzweigungsrichtungen entsprechend eingeschränkt. Mehr als drei verschiedene Richtungen sind nicht möglich.

Die Methode `get_turning_direction` in der Class für Kreuzungen wurde erstellt, um die Abzweigungsrichtung von einer Fahrspur `incoming_lane` in eine Fahrspur `outgoing_lane` zu bestimmen. Sie gibt entweder -1, 0 oder 1 zurück. Dafür vergleicht sie die Winkel der beiden Fahrspuren. Unterschiede kleiner als 30 Grad werden als «geradeaus» klassifiziert.

Auf einer mehrspurigen Strasse, bei der jede Spur unterschiedliche Abzweigungsrichtungen hat, muss ein Fahrzeug auf die richtige Spur wechseln. Dafür wurde die Spurwechselmethode `check_lane_change` erweitert, sodass die Tendenz Δa_{bias} diesen Spurwechsel begünstigt. Ebenfalls wird verhindert, dass wieder in die falsche Spur zurück gewechselt wird, beispielsweise um andere Fahrzeuge zu überholen.

Ebenfalls musste der Dijkstra-Algorithmus so angepasst werden, dass er keine unmöglichen Routen generiert, die Abbiegeverbote an einer Kreuzung missachten. Dafür wurde vor allem die Methode `get_neighbour_intersections` angepasst, sodass diese nur noch die Nachbarkreuzungen zurückgibt, die mit den erlaubten Abzweigungsrichtungen erreicht werden können. Dies ist aber nur möglich, wenn die Methode weiss, von welcher Fahrspur das Fahrzeug kommt. Daher musste ich das Argument `current_lane` einführen. Nebenbei konnte damit das Problem, dass an jeder beliebigen Kreuzung U-Turns durchgeführt wurden, behoben werden.

2.7.6 Nächste Fahrspur eines Fahrzeuges

Nun hatte ich die Methode `path_finder` erstellt, die den kürzesten Weg eines Fahrzeuges zu seinem Ziel berechnet. Als Output erhielt man jedoch nur die Liste aller Kreuzungen, die das Fahrzeug auf seinem Weg besuchen soll. Die Kreuzung, auf der es sich gerade befindet, sollte aber genau wissen, auf welche Fahrspur sie das Fahrzeug weiterleiten muss.

Dazu schrieb ich die Methode `get_next_lane`, die sich, im Gegensetzt zu `path_finder`, in der `class Intersection` befindet. Sie führt zuerst den Dijkstra-Algorithmus mit `self` als Start und `vehicle.destination` als Ziel aus. Letztere ist die Zielkreuzung, die bei der Generation des Fahrzeuges ausgewählt worden ist. Die nächste Kreuzung ist dann das erste Element der zurückgegebenen Liste `path`. Nun werden mit einer for-Schleife alle Fahrspuren aus `self.outgoing_lanes` gefunden, die zu dieser Kreuzung führen.

Wenn mehrere Spuren möglich sind, d. h., wenn die kommende Strasse mehrspurig ist, werden zunächst diejenigen Spuren aussortiert, die die Abzweigungsrichtung der Route an der nächsten Kreuzung nicht erlauben. Somit wird bereits hier sichergestellt, dass das Fahrzeug in der richtigen Spur für das nächste Abbiegemanöver ist.

Wenn danach immer noch mehrere Spuren möglich sind, wird diejenige ausgewählt, deren Index am nächsten zum Index der vorherigen Spur ist. Der Index wird in diesem Fall von rechts nach links gezählt und beginnt bei der rechtsliegenden Spur einer Strasse mit der Zahl 1. Durch diese Indexierung wird sichergestellt, dass Fahrzeuge, die aus einer mehrspurigen Strasse kommen und in eine mehrspurige Strasse weitergeleitet werden, nach der Kreuzung in der gleichen Spur bleiben.

Die ausgewählte nächste Spur wird direkt im Fahrzeug als `vehicle.next_lane` gespeichert. Somit muss sie nicht bei jedem Update-Intervall neu berechnet werden, was vorteilhaft ist, da der Dijkstra-Algorithmus ziemlich viel Rechenleistung benötigt. Diese Änderung führte ich ein, als meine immer komplizierter werdende Simulation zu langsam wurde, und ich konnte tatsächlich eine merkliche Verbesserung feststellen.

Die Methoden `next_vehicle` und `last_vehicle` in der Class für Fahrspuren passte ich ebenfalls an, sodass auch Fahrzeuge nach der nächsten bzw. vor der letzten Kreuzung für die IDM-Beschleunigung und den Spurwechsel berücksichtigt werden. Damit konnte ich gefährliche Bremsmanöver nach der Kreuzung verhindern.

2.7.7 Verbindung von Strassen und Kreuzungen

Damit die Fahrzeuge von einer Strasse in eine Kreuzung und dann in die nächste Strasse fahren können, müssen die Strassen und Kreuzungen miteinander verbunden werden. Dafür erstellte ich die folgende Methode in der `class Simulation`:

```
def connect(self, road, intersection, position: str):
    'Erstellen einer Verbindung zwischen Strasse und Kreuzung'
    if position == 'start':
        road.start_connection = intersection
        for lane in road.lanes_0:
            intersection.outgoing_lanes.append(lane)
            lane.start_connection = intersection
        for lane in road.lanes_1:
            intersection.incoming_lanes.append(lane)
            lane.end_connection = intersection
    elif position == 'end':
        road.end_connection = intersection
        for lane in road.lanes_0:
            intersection.incoming_lanes.append(lane)
            lane.end_connection = intersection
        for lane in road.lanes_1:
            intersection.outgoing_lanes.append(lane)
            lane.start_connection = intersection
    else:
        raise ValueError('Position must be "start" or "end"')
    if road.width > intersection.width:
        intersection.width = road.width
    for lane in intersection.incoming_lanes:
        lane.stop_position = lane.lenght - intersection.width / 2
```

Code 15: Verbinden von Strassen und Kreuzungen

Die Methode stellt die Verbindung für jede einzelne Fahrspur der Strasse her, indem diese in der Kreuzung in der Liste `intersection.incoming_lanes` bzw. `intersection.outgoing_lanes` referenziert wird. Zusätzlich wird die Kreuzung in der Fahrspur als `lane.start_connection` bzw. `lane.end_connection` referenziert. Am Schluss wird die Breite der Kreuzung noch an die Breite der Strasse angepasst und für jede Spur die Anhalteposition festgelegt, wo Fahrzeuge später anhalten sollen, sofern sie vor der Kreuzung warten müssen.

Die Verbindung ist nur dann sinnvoll, wenn sich die Kreuzung und der Anfang / das Ende der Strasse an der gleichen Position befinden. Bei meinen ersten Testversuchen erstellte ich alle Verbindungen manuell, später erstellte ich dann die Methode `connect_roads`, die diesen Prozess automatisiert. Sie orientiert sich dabei an den Positionen im Koordinatensystem und verbindet alle Strassen/Kreuzungen, welche die gleichen Koordinaten haben.

2.8 Vortritt

Nun konnte meine Simulation bereits komplexe Strassennetze mit mehrspurigen Strassen darstellen, in denen Fahrzeuge eigenständig den besten Weg zum Ziel finden. Ein grosses Problem galt es allerdings noch zu beseitigen: An den Kreuzungen fuhren alle Fahrzeuge einfach durch, ohne die anderen heranfahrenden Fahrzeuge zu berücksichtigen, auch wenn sich deren Wege kreuzten. Für diese Fälle gibt es in der Realität Vortrittsregeln, die genau festlegen, welches Fahrzeug die Kreuzung zuerst überqueren darf. Im Gegensatz zu den anderen Problemen, denen ich bei der Entwicklung meiner Simulation begegnet war, fand ich für die Vortrittsregeln kein einfaches Modell, das die Entscheidung, ob ein Fahrzeug Vortritt hat, beschreibt. Ich war also auf mich allein gestellt, um ein Konzept für die Implementierung der Vortrittsregeln zu entwickeln. Im Folgenden werden die Vortrittsregeln, die in der Schweiz gelten, kurz zusammengefasst. Die Zusammenfassung basiert auf meinem eigenen Wissen und der Broschüre des TCS[14]. Anschliessend wird erklärt, wie ich diese Regeln in der Simulation umgesetzt habe.

2.8.1 Vortrittsregeln in der Schweiz

Die Vortrittsregeln kommen grundsätzlich immer dann zur Anwendung, wenn sich die Wege zweier Fahrzeuge kreuzen, also wenn das gleichzeitige Überqueren einer Kreuzung zu einer Kollision führen würde.

Rechtsvortritt

Wenn nichts anderes signalisiert ist, haben von rechts kommende Fahrzeuge Vortritt, unabhängig davon, in welche Richtung sie abbiegen. Wenn gleichzeitig Fahrzeuge von allen Richtungen kommen, muss der Vortritt mit Handzeichen geregelt werden.

Vortritt des Gegenverkehrs

Beim Linksabbiegen hat grundsätzlich der Gegenverkehr Vortritt, es sei denn, man folgt einer Hauptstrasse, die die Richtung ändert, und der Gegenverkehr kommt aus einer Nebenstrasse.

Signal «Kein Vortritt»

Bei diesem Signal muss man den Fahrzeugen auf der vortrittsberechtigten Strasse den Vortritt lassen. Dies kann auch durch andere Signale wie «Stop» oder durch bauliche Massnahmen wie Trottoirüberfahrten signalisiert werden. Mehrere Fahrzeuge, die alle aus einer Strasse ohne Vortritt kommen, wenden untereinander den Rechtsvortritt an.

Hauptstrassen

Auf Hauptstrassen hat man immer Vortritt, ausser man verlässt die Hauptstrasse und überquert dabei den Gegenverkehr auf der Hauptstrasse.

---

[14] Touring Club Schweiz: *Der Vortritt und ich*. Auf: https://www.tcs.ch/mam/Verkehrssicherheit/PDF/Booklets/der-vortritt-und-ich.pdf (abgerufen am 17.10.2024).

2.8.2 Konzept zur Umsetzung der Regeln

Wenn ein Fahrer in eine Kreuzung einfährt, schaut er auf alle anderen Strassen, die ebenfalls an der Kreuzung enden und überprüft, ob sich dort vortrittsberechtigte Fahrzeuge befinden. Sobald mindestens ein vortrittsberechtigtes Fahrzeug in Sicht ist, muss er anhalten. Ich gehe in meiner Simulation davon aus, dass alle Fahrer immer genau wissen, wo sich alle anderen Fahrzeuge befinden und wohin diese abbiegen möchten. Somit sind Unfälle ausgeschlossen.

Rechtsvortritt und Vortritt des Gegenverkehrs

Um den Rechtsvortritt umzusetzen, muss das Programm irgendwie wissen, welche Strasse von rechts kommt. Dafür werden die Strassen im Gegenuhrzeigersinn sortiert und nummeriert. Man kann sich die Kreuzung als Kreis, der im Gegenuhrzeigersinn befahrend wird, vorstellen. Jede Strasse, an der dabei vorbeigefahren wird, ist vortrittsberechtigt. Im abgebildeten Beispiel biegt das Fahrzeug aus der Strasse 1 nach links in die Strasse 4 ab. Dabei fährt es an den Strassen 2 und 3 vorbei. Die Fahrzeuge auf diesen Strassen haben daher Vortritt. Mit diesem Konzept werden die beiden Regeln «Rechtsvortritt» und «Vortritt des Gegenverkehrs» vereint.

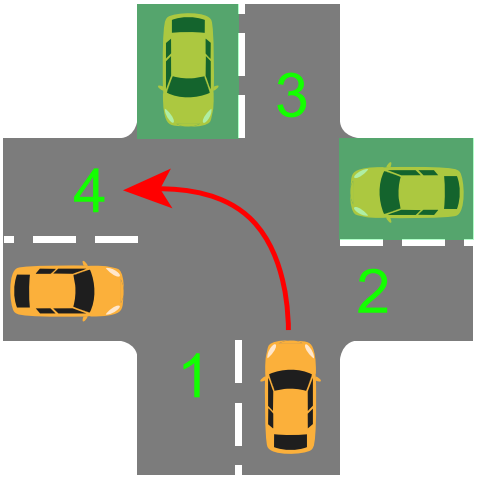


Abbildung 3: Ein Fahrzeug möchte an einer Kreuzung links abbiegen. Dabei muss es den beiden grün markierten Fahrzeugen den Vortritt gewähren.

Bis jetzt wird allerdings noch nicht berücksichtigt, in welche Richtungen die anderen Fahrzeuge abbiegen möchten. Beispielsweise könnte das Fahrzeug 2 rechts abbiegen. Dann hätte es zwar Vortritt gegenüber dem Fahrzeug 1, jedoch könnten die Fahrzeuge die Kreuzung ohnehin gleichzeitig überqueren. Dieser Fall kann durch folgende Bedingung erkannt werden: Wenn die Zielstrasse von Fahrzeug 2 (Strasse 3) ebenfalls in der Liste der Strassen liegt, an denen Fahrzeug 1 vorbeifährt, kann die Kreuzung gleichzeitig befahren werden. Somit könnten auch die Fahrzeuge 1 und 3 gleichzeitig fahren, sofern Fahrzeug 3 links abbiegt und Fahrzeug 2 bereits abgebogen ist. Natürlich muss die Kreuzung hierfür genügend Platz bieten. In meiner Simulation gehe ich aber immer davon aus, dass dies der Fall ist.

Bei Kreuzungen mit Rechtsvortritt kann es zur Situation kommen, dass Fahrzeuge gleichzeitig aus allen Richtungen kommen. Dann darf streng gesehen kein Fahrzeug zuerst fahren, da jedes Fahrzeug dem von rechts kommenden Fahrzeug den Vortritt gewähren muss. In der Realität würde der Vortritt mit Handzeichen geregelt werden, aber in der Simulation gibt es keine Handzeichen. Deshalb muss die Simulation eine solche Situation erkennen und auflösen, indem ein zufällig ausgewähltes Fahrzeug den Vortritt erhält.

Vortrittsberechtige Strassen

Wenn Strassen vorkommen, die gegenüber anderen Strassen vortrittsberechtigt sind, wird den Fahrzeugen auf diesen Strassen natürlich der Vortritt gewährt. Aber auch hier muss wieder geprüft werden, ob das Kreuzen nicht gleichzeitig möglich ist. Wir betrachten nochmals die Abbildung, gehen aber davon aus, dass die Strassen 2 und 4 Vortritt haben. Zuerst wird das eben erarbeitete Prinzip des Rechtsvortritts angewandt, womit die Fahrzeuge 2 und 3 bereits als vortrittsberechtigt erkannt und aussortiert werden. Nun muss aber noch das Fahrzeug 4 berücksichtigt werden, da es sich auf der vortrittsberechtigten Strasse befindet. Somit hat es Vortritt gegenüber dem Fahrzeug 1. Wenn das Fahrzeug 4 aber nicht an der Strasse 1 vorbeifährt, sondern rechts abbiegt, können die Fahrzeuge trotzdem gleichzeitig losfahren.

Die Berücksichtigung der vortrittsberechtigten Strassen ist also nur eine Erweiterung des Rechtsvortritts. Somit kann in jedem Fall sichergestellt werden, dass der Vortritt nur als solcher erkannt wird, wenn das gleichzeitige Befahren der Kreuzung in einer Kollision resultieren würde.

Das vorgestellte Modell ist sehr flexibel, funktioniert also bei jeder Kreuzung. Auch Verkehrskreisel lassen sich durch die Kombination mehrerer Kreuzungen realisieren. Trotzdem kann es sein, dass ich gewisse Situationen nicht bedacht habe, bei denen das Modell versagen würde.

2.8.3 Umsetzung des Konzepts im Programm

Der Vortritt wird grundsätzlich in der `class Intersection` überprüft. Dafür sind mehrere Methoden zuständig, die im Folgenden vorgestellt werden:

```
def get_priority_vehicles(self, incoming_lane, outgoing_lane) -> list:
    'Gibt alle Fahrzeuge zurück, die Vortritt haben'
    priority_vehicles: list = []
    crossing_lanes: list = self.get_crossing_lanes(incoming_lane, outgoing_lane)

    for other_lane in self.incoming_lanes:
        if other_lane == incoming_lane:
            continue
        try:
            other_vehicle = other_lane.vehicles[-1]
        except IndexError:
            continue

        # 1. Vortritt, wenn anderes Fahrzeug rot hat
        # 2. Kein Vortritt für anderes Fahrzeug, wenn dessen Fahrspur keinen Vortritt hat
        # 3. Vortritt für Fahrzeuge auf der gleichen Strasse, wenn die Spur gewechselt wird
        # 4. Rechtsvortritt
        # 5. Vortritt für Fahrzeuge aus vortrittsberechtigten Strassen

        if other_lane.traffic_light_status == 'red':
            continue
        elif (incoming_lane.priority and not other_lane.priority
            and incoming_lane.traffic_light_status == 'inactive'):
            continue
        elif other_lane.road == incoming_lane.road:
            if other_vehicle.next_lane == outgoing_lane:
                if incoming_lane.index != outgoing_lane.index:
```

```
                    if other_lane.index != outgoing_lane.index:
                        if other_lane.index > incoming_lane.index:
                            continue
                    priority_vehicles.append(other_vehicle)
            elif other_lane in crossing_lanes:
                if other_vehicle.next_lane:
                    if other_vehicle.next_lane in crossing_lanes:
                        continue
                priority_vehicles.append(other_vehicle)
            elif (other_lane.priority and not incoming_lane.priority
                and incoming_lane.traffic_light_status == 'inactive'):
                if other_vehicle.next_lane:
                    if not (other_vehicle.next_lane in crossing_lanes
                        or other_vehicle.next_lane.road == outgoing_lane.road):
                        continue
                priority_vehicles.append(other_vehicle)

    return priority_vehicles
```

Code 16: Methode get_priority_vehicles für die Vortrittsregeln

Die Methode `get_priority_vehicles` ist der Kernbestandteil, da dort die eigentlichen Vortrittsregeln angewendet werden. Sie benötigt die Argumente `incoming_lane` und `outgoing_lane`, das sind die Fahrspuren, auf denen sich das Fahrzeug vor und nach der Kreuzung befindet. Als Resultat gibt sie eine Liste mit allen Fahrzeugen zurück, denen bei diesem Abbiegemanöver der Vortritt gewährt werden muss. Dabei wird noch nicht berücksichtigt, wie weit diese Fahrzeuge von der Kreuzung entfernt sind.

Mit einer for-Schleife wird jede Fahrspur berücksichtigt, die an der Kreuzung endet. Das vorderste Fahrzeug, das sich auf der Fahrspur befindet, wird jeweils als `other_vehicle` definiert. Falls dieses nicht existiert, kann mit dem Befehl `continue` direkt zur nächsten Iteration gesprungen werden. Anschliessend wird mit fünf Kriterien geprüft, ob das andere Fahrzeug Vortritt hat oder nicht. Sobald ein Kriterium zutrifft, werden die weiteren Kriterien nicht mehr geprüft. Wenn das Fahrzeug keinen Vortritt hat oder sich die Wege nicht kreuzen, wird mit `continue` zur nächsten Iteration gesprungen, ansonsten wird es zur Liste der vortrittsberechtigten Fahrzeuge `priority_vehicles` hinzugefügt.

Das erste Kriterium hängt bereits mit Lichtsignalen zusammen, die im Kapitel 2.9 eingeführt werden. Wenn das andere Fahrzeug am Rotlicht steht, hat es auf keinen Fall Vortritt und muss nicht weiter berücksichtigt werden.

Das zweite Kriterium ist ebenfalls ein einfaches Ausschlusskriterium. Wenn die Fahrspur `other_lane` gegenüber der Fahrspur `incoming_lane` nicht vortrittsberechtigt ist, muss erstere nicht weiter berücksichtigt werden. Dies wird durch die Variable `priority` in der Class für Fahrspuren bestimmt, die entweder `True` oder `False` sein kann.

Das dritte Kriterium wurde später hinzugefügt und stellt sicher, dass es keine Konflikte bei mehrspurigen Strassen gibt, die nach der Kreuzung weniger Fahrspuren haben. Mit dieser Umsetzung bin ich nicht ganz zufrieden, da die Fahrzeuge erst vor der Kreuzung anhalten und warten, bis eine Lücke in der anderen Spur frei wird. In der Realität würde man in einer solchen Situation einen Reissverschluss bilden, d. h. die Fahrzeuge würden

hereingelassen werden. Dieses Verhalten wäre allerdings zu kompliziert in der Umsetzung.

Im vierten und fünften Kriterium wird schliesslich mein Konzept umgesetzt. Dafür ist die Methode `self.get_crossing_lanes` zentral. Sie gibt alle Fahrspuren zurück, an denen beim Abbiegen von `incoming_lane` nach `outgoing_lane` links vorbeigefahren wird. Dies funktioniert, da die Strassen, mit denen die Kreuzung verbunden ist, in einer Liste nach Winkel sortiert werden. Damit kann der Rechtsvortritt gemäss der vorherigen Beschreibung überprüft werden. Das andere Fahrzeug wird nur dann zur Liste `priority_vehicles` hinzugefügt, wenn sich die Wege kreuzen.

Entscheidung des Vortritts

Die Methode `check_priority` kann schliesslich entscheiden, ob ein Fahrzeug anhalten muss oder nicht. Dazu führt sie die Methode `get_priority_vehicles` aus, um die Liste mit allen potenziell vortrittsberechtigten Fahrzeugen zu erhalten. Dann überprüft sie für jedes dieser Fahrzeuge, ob es noch weit genug von der Kreuzung entfernt ist, um trotzdem durchfahren zu können. Sobald ein Fahrzeug gefunden wird, das diese Bedingung nicht erfüllt, ist der Output `False`, d. h., das Fahrzeug muss anhalten. Zusätzlich kann diese Methode sicherstellen, dass die Kreuzung bei einem Stau freigehalten wird. Das Fahrzeug wird also auch dann angehalten, wenn es auf der nächsten Fahrspur keinen Platz hat.

Anhalten des Fahrzeuges

In der Methode `update` der `class Lane` wird während jedem Update-Intervall der Vortritt für jedes Fahrzeug auf der Fahrspur geprüft. Allenfalls werden die Fahrzeuge dann mit `vehicle.stop(position=self.stop_position)` angehalten. Diese Funktion hält das Fahrzeug aber nur an, wenn die IDM-Beschleunigung eine gewisse maximale Bremsbeschleunigung nicht übersteigt. Somit wird verhindert, dass die Fahrzeuge kurz vor der Kreuzung ihre Entscheidung ändern und stark abbremsen.

2.8.4 Abbremsen der Fahrzeuge vor dem Abbiegen

Wenn die Fahrzeuge nicht vortrittsbedingt anhalten müssen, fahren sie mit voller Geschwindigkeit durch die Kreuzung, auch wenn sie abbiegen. Dieses Verhalten ist unrealistisch, deshalb sollten sie vor der Kreuzung abgebremst werden. Um dies zu erreichen, definierte ich in der Class für Kreuzungen den Parameter `turning_speed`, der eine maximale Geschwindigkeit für das Überqueren der Kreuzung festlegen sollte. Die Fahrzeuge sollten aber nur dann abbremsen, wenn sie abbiegen, d. h. die Richtung ändern. Da ich die Abzweigungsrichtungen bereits implementiert hatte, konnte ich einfach die Bedingung einfügen, dass die Abzweigungsrichtung des Fahrzeuges nicht 0 sein darf. Das wird ebenfalls in der Methode `update` der `class Lane` gemacht. Abhängig von der Geschwindigkeit des Fahrzeuges wird dann die Distanz berechnet, bei der seine Geschwindigkeitsbegrenzung auf die der nächsten Kreuzung heruntergesetzt wird. Das Fahrzeug bremst dann mit seiner komfortablen Bremsbeschleunigung ab und erreicht genau bei der Kreuzung die neue Wunschgeschwindigkeit.

2.9 Lichtsignale

Viele Kreuzungen mit hohem Verkehrsaufkommen haben Lichtsignalanlagen (LSA), um den Verkehr besser steuern zu können. Diese wollte ich auch in meine Simulation einbauen. Ich stellte fest, dass die Implementierung der Lichtsignale ziemlich einfach war, da sie im Gegensatz zu den Vortrittsregeln nach einem festen Zeitplan gesteuert werden. In der Realität gibt es zwar häufig Sensoren, um den Zeitplan je nach Verkehrsaufkommen dynamisch einzustellen, damit wollte ich mich aber noch nicht weiter beschäftigen.

2.9.1 Class für Lichtsignalanlagen

Ich entschied mich dazu, für die LSA eine neue Class anzulegen, statt den Code direkt in die Class für Kreuzungen zu integrieren, damit der Code übersichtlich bleibt.

```
class TrafficLight:
    'Class für Lichtsignale'
    def __init__(self, intersection, phases: list):
        self.intersection = intersection
        self.lanes: list = intersection.incoming_lanes
        self.phases: list = phases
        index: int = 0
        for phase in phases:
            phase['index'] = index
            index += 1
        self.timer: float = 0
        self.current_phase: dict = self.phases[0]
        self.active: bool = True
        self.set_lane_status()

    def get_next_phase(self) -> dict:
        'Ermittlung der nächsten Phase'
        try:
            next_phase = self.phases[self.current_phase['index'] + 1]
        except IndexError:
            next_phase = self.phases[0]
        return next_phase

    def set_lane_status(self):
        'Anpassung des Status für alle Fahrspuren entsprechend der aktuellen Phase'
        for lane in self.lanes:
            if self.active:
                if lane in self.current_phase['active_lanes']:
                    lane.traffic_light_status = 'green'
                else:
                    lane.traffic_light_status = 'red'
            else:
                lane.traffic_light_status = 'inactive'

    def update(self, dt: float):
        'Update des Lichtsignalzyklus für eine bestimmte Zeit'
        if self.active:
            self.timer += dt
            if self.timer > self.current_phase['time']:
                self.timer = 0
                self.current_phase = self.get_next_phase()
                self.set_lane_status()
        else:
            self.set_lane_status()
```

Code 17: Class für Lichtsignalanlagen (LSA)

Die Lichtsignalanlage hat eine Liste mit allen Fahrspuren, die von ihr kontrolliert werden. Das sind normalerweise alle Fahrspuren, die an der Kreuzung enden, deshalb wird einfach die Liste `intersection.incoming_lanes` referenziert. Referenzieren bedeutet, dass keine Kopie der Liste angelegt wird, sondern auf die ursprüngliche Liste zurückgegriffen wird. Eine Referenz ist wesentlich effizienter als eine Kopie, da sie keinen neuen Speicherplatz belegt. Der Nachteil ist, dass bei einer Änderung (Hinzufügen oder entfernen von Elementen) auch die ursprüngliche Liste verändert wird. In Python wird mit dem Operator «=» standardmässig eine Referenz angelegt. Wenn man eine Kopie möchte, muss man dies explizit mit `copy.copy(list)` oder `list.copy()` angeben.

Zusätzlich braucht die Lichtsignalanlage einen Phasenplan, der genau festlegt, welche Fahrspuren für wie lange grün bzw. rot haben. Dieser ist als Liste von Dictionaries organisiert, wobei die Dictionaries jeweils eine Phase beschreiben. Jede Phase hat zwei Elemente: `'time'` beschreibt, wie lange (in Sekunden) die Phase läuft und `'active_lanes'` ist eine Liste mit allen Fahrspuren, die während dieser Phase grün haben. Alle anderen Fahrspuren haben jeweils rot. Auf das gelbe Licht habe ich verzichtet, stattdessen erlaube ich den Fahrzeugen, über rot zu fahren, sofern sie nicht mehr rechtzeitig abbremsen können. Zusätzlich erhält jede Phase in der Methode `__init__` einen Index, der die genaue Reihenfolge der Phasen festhält.

Die Methode `get_next_phase` ermittelt die nächste Phase und speichert sie unter `self.current_phase`. Wenn sie am Ende der Liste angelangt ist, beginnt sie erneut von vorne.

Mit `set_lane_status` wird der Status für alle Fahrspuren angepasst. Dafür wurde in der Class für Fahrspuren eine neue Variable `traffic_light_status` erstellt. Diese kann die Zustände `'red'`, `'green'` und `'inactive'` annehmen. Letzterer ist der Standardwert und bedeutet, dass kein Lichtsignal existiert oder dieses ausgeschaltet ist.

Die Methode `update` zählt die Zeit mit `self.timer` und wechselt gegebenenfalls zur nächsten Phase.

2.9.2 Berücksichtigung der Lichtsignale beim Vortritt

Da der Lichtsignalstatus direkt in der Fahrspur gespeichert ist, lässt er sich sehr einfach bei den Vortrittsregeln berücksichtigen. Die Methode `check_priority` hält das Fahrzeug an, wenn seine Fahrspur ein Rotlicht hat. In diesem Fall müssen die weiteren Vortrittsregeln nicht mehr berücksichtigt werden. Wenn das Fahrzeug aber grün hat, werden die anderen Fahrzeuge, die auch grün haben, weiterhin mit dem Rechtsvortritt berücksichtigt. So sind Lichtsignale möglich, bei denen das Linksabbiegen bei aktivem Gegenverkehr erlaubt ist.

2.10 Grafische Darstellung

Schon sehr früh in der Entwicklung brauchte ich eine einfache Möglichkeit, um den geschriebenen Programmcode zu testen. Der erste Testversuch konnte zwar noch im Terminal durchgeführt werden, für das zweidimensionale Koordinatensystem war dieses allerdings nicht mehr geeignet. Dafür brauchte ich ein Graphical User Interface (GUI). Als Lösung stiess ich schnell auf Pygame[15], eine Python-Bibliothek, die primär zur Erstellung von Videospielen gedacht ist. Da ich mich auf diesem Gebiet noch nicht auskannte, musste ich oft auf die Dokumentation von Pygame zurückgreifen. Das GUI wurde während der Programmierung laufend erweitert, um die neuen Elemente der Simulation anzeigen zu können.

Um die Bibliothek nutzen zu können, muss sie zuerst im Terminal mit `pip install pygame` installiert werden. Im Programm müssen die Module dann mit `import pygame` importiert werden.

2.10.1 Initialisierung des Fensters

Auch für die grafische Darstellung der Simulation erstellte ich eine Class, die `class Window` genannt wurde. In der Methode `__init__` wird das Pygame-Fenster erstellt:

```
def __init__(self, size: tuple, name: str='Traffic Simulation', FPS: int=60):
    'Initialisierung von Pygame & Erstellung der Simulation'
    self.scaling: float = 5
    self.offsetx: int = 0
    self.offsety: int = 0
    self.show_text: bool = False
    self.show_signs: bool = True
    self.show_visualisation: bool = True
    self.FPS: int = FPS
    self.time_acceleration: float = 1

    self.simulation = Simulation(name=name)
    pygame.init()
    self.screen = pygame.display.set_mode(size, pygame.RESIZABLE)
    self.screen.fill(self.white)
    self.clock = pygame.time.Clock()
    pygame.display.set_caption(self.simulation.name)
    pygame.display.update()
```

Code 18: Initialisierung der grafischen Darstellung mit Pygame

Zuerst werden einige Variablen definiert. Die Variablen `self.scaling`, `self.offsetx` und `self.offsety` dienen später dazu, dass die Darstellung skaliert und verschoben werden kann. `self.FPS` legt fest, wie oft de Simulation pro Sekunde aktualisiert wird. Höhere Werte bedeuten mehr Bilder pro Sekunde (frames per second, FPS) und somit eine flüssigere Darstellung. Damit wird auch die Länge der Update-Intervalle definiert. Die restlichen Variablen dienen dazu, gewisse Elemente der Anzeige ein- oder auszublenden.

[15] Pygame: *Pygame Front Page*. Auf: https://www.pygame.org/docs/ (abgerufen am 19.10.2024).

Dann wird die Simulation erstellt, indem ein neues Objekt der `class Simulation` initiiert wird. Die `class Window` ist also der `class Simulation` übergeordnet, da sie die Simulation enthält.

Im letzten Schritt wird schliesslich das Pygame-Fenster initiiert. Die Methode `pygame.init()` startet Pygame. Dann wird ein Fenster mit einer bestimmten Grösse `size` erstellt. Dank `pygame.RESIZABLE` kann dieses auch später noch in der Grösse verändert werden. Als nächstes wird der Inhalt des Fensters `self.screen` mit der Farbe Weiss gefüllt. Die Farben sind Tupeln der Form `(255, 255, 255)`, wobei die Zahlen für die Farben Rot, Grün und Blau stehen und Werte von 0 bis 255 annehmen können. Alle Farben, die ich benötige, habe ich ausserhalb von `__init__` als Class Variables definiert. Mit `pygame.time.Clock()` wird ein Objekt erstellt, das den Zeitablauf des Programms kontrollieren kann. Schliesslich bekommt das Fenster noch einen Namen und wird zum ersten Mal aktualisiert.

Der Grundablauf funktioniert dann so, dass bei jedem Update-Intervall der Bildschirm zuerst weiss gefüllt wird und anschliessend alle Elemente (Fahrzeuge, Strassen, Markierungen, etc.) neu aufgezeichnet werden.

2.10.2 Darstellung von Fahrzeugen

Das erste Ziel war die Darstellung von Fahrzeugen. Zu diesem Zeitpunkt gab es in der Simulation erst eine Strasse, deshalb wurde die Fahrzeugkolonne einfach in der Mitte des Fensters dargestellt. Auf schöne Bilder von Autos habe ich verzichtet, stattdessen werden die Fahrzeuge einfach als weisse Rechtecke dargestellt, wobei ihre Grösse natürlich berücksichtigt wird. Daran hat sich auch bis zum Schluss nichts verändert.

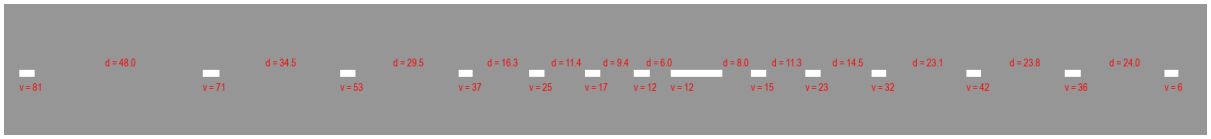


Abbildung 4: Erste Version der grafischen Darstellung. Zu sehen ist eine Fahrzeugkolonne, in der sich eine Stauwelle gebildet hat (inkl. Geschwindigkeiten und Abstände zwischen den Fahrzeugen).

Für die Anzeige von Fahrzeugen ist die Methode `draw_vehicle` zuständig. Sie rechnet die Position des Fahrzeuges in Bildschirmkoordinaten um und benutzt anschliessend die Methode `draw_rect`, um ein gedrehtes Rechteck an der entsprechenden Position zu zeichnen. Die Rechtecke müssen gedreht werden, da die Strassen im zweidimensionalen Koordinatensystem unterschiedliche Winkel haben können und die Fahrzeuge auch in diesen Winkeln angezeigt werden sollen. Die beiden Methoden habe ich aufgeteilt, da `draw_rect` später auch für andere Elemente benutzt werden kann. Letztere wird nun etwas genauer betrachtet:

```
def draw_rect(
    self,
    lenght: float,
    width: float,
    pos_center: tuple,
    angle: float,
    color: tuple=white
) -> pygame.Rect:
    'Zeichnen eines (gedrehten) Rechtecks um einen Mittelpunkt'
    surface = pygame.Surface((lenght, width), pygame.SRCALPHA)
    surface.fill(color)
    surface = pygame.transform.rotate(surface, math.degrees(angle))
    rect = surface.get_rect()
    rect.center = pos_center
    self.screen.blit(surface, rect)
    return rect
```

Code 19: Zeichnen eines gedrehten Rechtecks

Die Methode erstellt eine rechteckige Fläche mit der angegebenen Länge und Breite und füllt diese mit der angegebenen Farbe. Dann dreht sie die Fläche mit `pygame.transform.rotate`. Da die trigonometrischen Funktionen von Python den Winkel im Bogenmass berechnen, muss dieser noch umgerechnet werden. Schliesslich bestimmt sie den neuen rechteckigen Rahmen um die gedrehte Fläche mit `surface.get_rect()`. Der Objekttyp `pygame.Rect` hat zum Vorteil, dass man das Rechteck anhand seines Mittelpunktes verschieben kann. Der Mittelpunkt ist der einzige Punkt, dessen Position beim Drehen nicht verändert wird, daher muss er als Argument angegeben werden. Zum Schluss wird die gedrehte Fläche mit `self.screen.blit(surface, rect)` auf dem Bildschirm abgebildet. Das Rechteck `rect` definiert dabei nur die Position und wird selbst nicht abgebildet. Es wird aber als Return Value zurückgegeben, da die Methode `draw_vehicle` damit die Position des Texts für die Geschwindigkeitsangabe berechnen kann.

2.10.3 Umrechnung der Koordinaten

Pygame hat ein eigenes Koordinatensystem, bei dem eine Einheit genau einem Pixel auf dem Bildschirm entspricht. Hingegen benutzt das Koordinatensystem der Simulation Meter als Einheit. Diese Koordinaten direkt zu übernehmen wäre daher nicht sinnvoll, stattdessen müssen sie skaliert und verschoben werden können.

```
def convert_pos(self, pos: tuple) -> tuple:
    'Konvertieren von Simulationskoordinaten in Bildschirmkoordinaten'
    posx = pos[0] * self.scaling + self.offsetx
    posy = pos[1] * self.scaling + self.offsety
    return (posx, posy)
```

Code 20: Umrechnung von Simulationskoordinaten in Bildschirmkoordinaten

Die Methode `convert_pos` konvertiert die Simulationskoordinaten in Bildschirmkoordinaten, indem sie diese mit der Skalierung `self.scaling` multipliziert und um den Faktor `self.offsetx` bzw. `self.offsety` verschiebt. Später wird es möglich sein, die Darstellung mit den Pfeiltasten zu verschieben und mit dem Scrollrad zu skalieren.

Ich möchte noch anmerken, dass das Koordinatensystem von Pygame an der oberen linken Ecke beginnt. Das heisst, die y-Achse ist im Vergleich zu einem normalen Koordinatensystem verkehrt. Statt die y-Werte bei der Konvertierung einfach umzukehren, basierte ich das gesamte Koordinatensystem der Simulation auf diesem Grundsatz. Das war vermutlich ein Fehler, da dieses ungewöhnliche Koordinatensystem bei der Erstellung des Strassennetzwerks berücksichtigt werden muss.

2.10.4 Darstellung von Strassen und Fahrspuren

Die Darstellung einer Strasse erfolgt in der Methode `draw_road`:

```
def draw_road(self, road):
    'Zeichnen einer Strasse auf dem Bildschirm'
    for lane in road.lanes_0+road.lanes_1:
        offset = lane.width/2
        point1 = self.convert_pos(road.offset(lane.pos_start, offset))
        point2 = self.convert_pos(road.offset(lane.pos_start, -offset))
        point3 = self.convert_pos(road.offset(lane.pos_end, -offset))
        point4 = self.convert_pos(road.offset(lane.pos_end, offset))
        pygame.draw.polygon(self.screen, self.grey, (point1, point2, point3, point4))
        if lane in road.lanes_0 and road.lanes_0.index(lane) > 0:
            pos_start = road.offset(lane.pos_start, -lane.width/2)
            self.draw_dashed_line(road, pos_start, angle=lane.angle, lenght=lane.lenght)
        elif lane in road.lanes_1 and road.lanes_1.index(lane) > 0:
            pos_start = road.offset(lane.pos_start, lane.width/2)
            self.draw_dashed_line(road, pos_start, angle=lane.angle, lenght=lane.lenght)
    if road.lanes_0 and road.lanes_1:
        self.draw_center_line(road)
```

Code 21: Darstellung einer Strasse

Ich entschied mich dazu, jede Fahrspur einzeln auf den Bildschirm zu zeichnen, da für die Fahrspuren bereits die genauen Start- und Endpositionen definiert waren. Zuerst erstellte ich die Fahrspuren ebenfalls als gedrehte Rechtecke mit der Methode `draw_rect`. Dann entdeckte ich jedoch die Methode `pygame.draw.polygon`, mit der eine beliebige Form anhand all ihrer Eckpunkte gezeichnet werden kann. Mit `road.offset` werden die Start- und Endpositionen einer Fahrspur um die halbe Spurbreite verschoben. Damit können alle vier Eckpunkte gebildet werden. Die Koordinaten dieser Punkte werden anschliessend in Bildschirmkoordinaten umgerechnet, um ein Polygon mit allen Eckpunkten auf den Bildschirm zeichnen zu können.

Damit die einzelnen Spuren erkennbar sind, müssen dazwischen Linien gezeichnet werden. Die Methode `draw_dashed_line` zeichnet eine gestrichelte Linie zwischen zwei Fahrspuren gleicher Richtung. Wenn mindestens eine Fahrspur in beide Richtungen vorhanden ist, wird mit der Methode `draw_center_line` zusätzlich eine durchgezogene Mittellinie erstellt. Da die Fahrzeuge in meiner Simulation nicht auf der Gegenfahrbahn überholen können, ist es durchaus sinnvoll, dass die Mittellinie immer durchgezogen ist. Die eben beschriebenen Methoden verwenden ebenfalls `pygame.draw.polygon` und `road.offset` zum Zeichnen der Spurbegrenzungslinien.

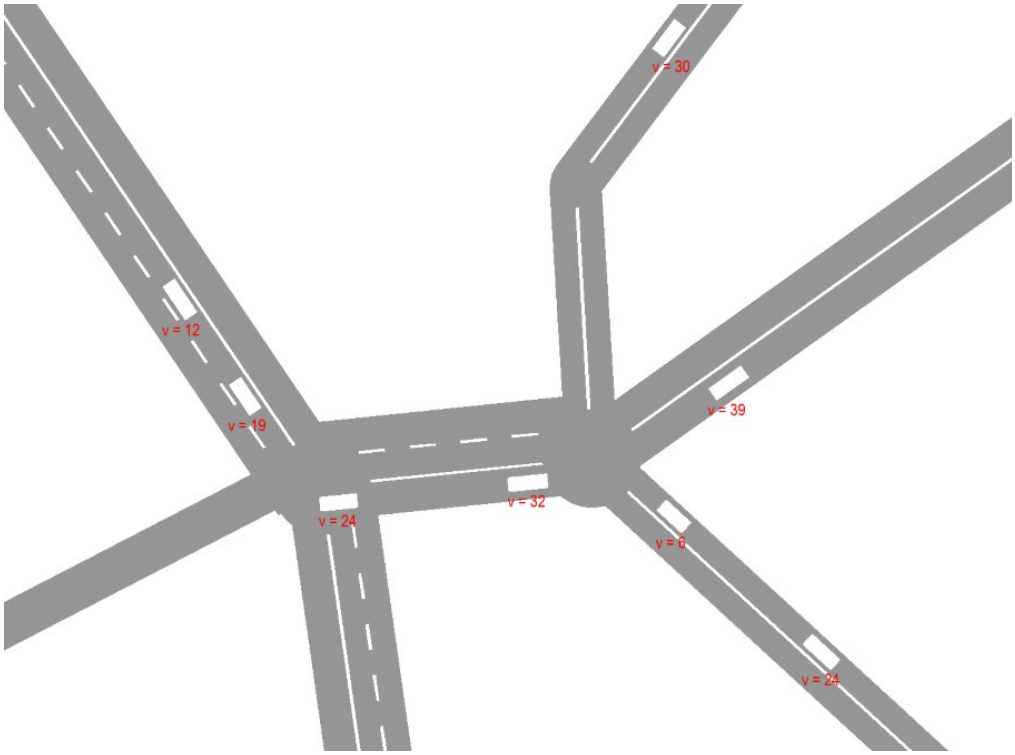


Abbildung 5: Grafische Darstellung mehrerer Strassen inkl. Spurbegrenzungen. Auf den Strassen sind mehrere Fahrzeuge zu sehen.

2.10.5 Darstellung von Kreuzungen

Kreuzungen müssen nicht zwingend dargestellt werden; aus Schönheitsgründen habe ich dies aber trotzdem gemacht. Die Methode `draw_intersection` zeichnet an der Position der Kreuzung einen Kreis mit deren Breite als Durchmesser. Dafür wird `pygame.draw.circle` verwendet. Zuvor stellte ich die Kreuzungen als Quadrate dar, das sah jedoch bei gewinkelten Strassen nicht schön aus.

2.10.6 Markierungen und Lichtsignale

Gewisse Eigenschaften der Fahrspuren sollten in der grafischen Darstellung ersichtlich sein. Dazu gehören:

- Einspurpfeile, wenn die erlaubten Abzweigungsrichtungen eingeschränkt sind.
- Das Symbol «Kein Vortritt», wenn die Fahrspur nicht vortrittsberechtigt ist.
- Lichtsignale, sofern die LSA aktiviert ist.

Die Symbole entnahm ich Wikipedia[16] und entfernte mit Photoshop den Hintergrund. Dann erstellte ich als Class Variable ein Dictionary, in welchem die Bilddateien bereits geladen werden. Das hat zum Vorteil, dass die Dateien nicht bei jedem Update-Intervall neu geladen werden.

[16] Wikipedia: *Bildtafel der Strassensignale in der Schweiz und in Liechtenstein seit 2023*. Auf: https://de.wikipedia.org/wiki/Bildtafel_der_Strassensignale_in_der_Schweiz_und_in_Liechtenstein_seit_2023 (abgerufen am 19.10.2024).

```
images: dict = {
    'Kein_Vortritt': pygame.image.load('Verkehrszeichen/Kein_Vortritt.png'),
    'Pfeil_links': pygame.image.load('Verkehrszeichen/Pfeil_links.png'),
    'Pfeil_geradeaus': pygame.image.load('Verkehrszeichen/Pfeil_geradeaus.png'),
    'Pfeil_rechts': pygame.image.load('Verkehrszeichen/Pfeil_rechts.png'),
    'Pfeil_links_geradeaus': pygame.image.load('Verkehrszeichen/Pfeil_links_geradeaus.png'),
    'Pfeil_rechts_geradeaus': pygame.image.load('Verkehrszeichen/Pfeil_rechts_geradeaus.png')
}
```

Code 22: Dictionary mit allen Symbolen

Die Dateien müssen sich im Ordner «Verkehrszeichen» befinden, der wiederum im gleichen Ordner wie die Programmdatei sein muss. Die Methode `draw_signs` überprüft, welche Symbole auf einer Fahrspur benötigt werden und fügt diese mit der Methode `draw_sign` an der richtigen Stelle ein. Die Bilder werden dabei auf die richtige Grösse skaliert und rotiert. Für Lichtsignale wird die Methode `draw_rect` benutzt, um einen farbigen Balken zu zeichnen.

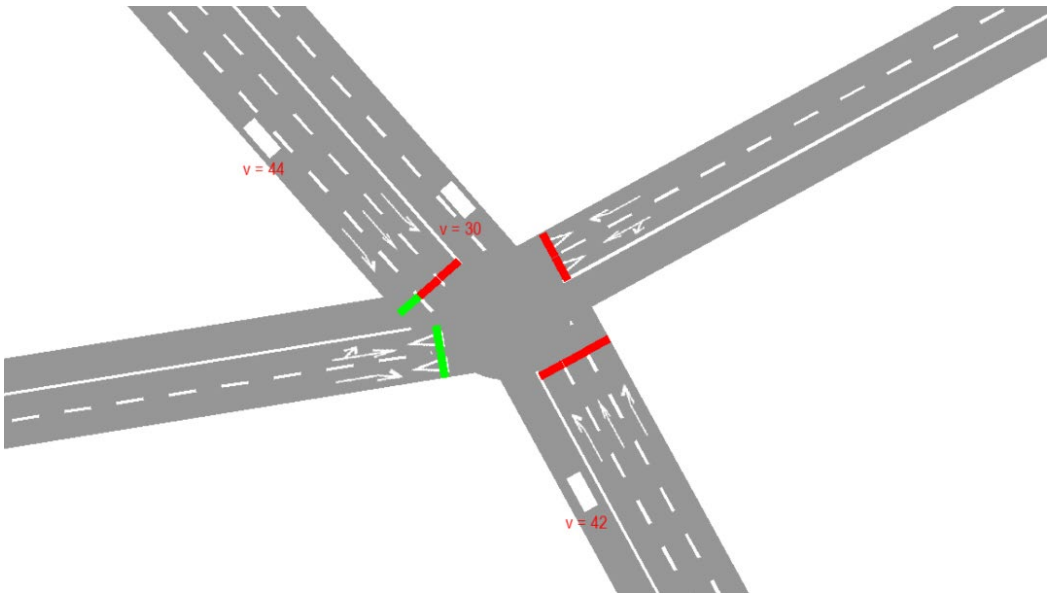


Abbildung 6: Eine Kreuzung mit Lichtsignalanlage. Ebenfalls erkennbar sind Einspurpfeile und Vortrittssignale.

2.10.7 Eingaben

Pygame unterstützt auch Tastatur- und Mauseingaben. Das machte ich mir zunutze, um diverse Steuerungsmöglichkeiten für die Simulation und die Darstellung einzubauen.

```
def check_events(self):
    'Einlesen von Maus- und Tastatureingaben'
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        elif event.type == pygame.KEYDOWN:
            # Erkennung aller Tastatureingaben
        elif event.type == pygame.MOUSEBUTTONDOWN:
            # Erkennung aller Mauseingaben
```

Code 23: Auswertung aller Tatstatur- und Mauseingaben

Die Auswertung der Eingaben geschieht in der Methode `check_events`. Mit einer for-Schleife werden alle Ereignisse (events) durchgegangen. Zuerst wird der Typ des Ereignisses ermittelt. Das Ereignis `pygame.QUIT` tritt auf, wenn das Fenster geschlossen wird. Dann muss dafür gesorgt werden, dass das Programm ordnungsgemäss beendet wird. Bei `pygame.KEYDOWN` und `pygame.MOUSEBUTTONDOWN` wird noch genauer unterschieden, welche Taste gedrückt wurde, anschliessend wird je nach Taste eine Aktion ausgeführt. Es folgt eine Liste mit allen Tasten und deren Funktionen:

Taste	**Funktion**
Pfeiltasten	Verschieben der Darstellung
Mausrad scrollen	Skalieren der Darstellung
Leertaste	Pausieren der Simulation
S	Beschleunigen der Simulationsgeschwindigkeit (bis 32x)
R	Zurücksetzen der Simulation (alle Fahrzeuge werden gelöscht)
V	Ein- und ausblenden der gesamten Visualisation
T	Ein- und ausblenden der Fahrzeugbeschriftungen
M	Ein- und ausblenden der Strassenmarkierungen
G	Ein- und ausschalten des Fahrzeuggenerators
Mausklick	Anzeige von Nummer und Position einer Kreuzung im Terminal

2.10.8 Update des Fensters

Die Methode `update` aktualisiert das Fenster und die Simulation:

```
def update(self):
    'Hauptschleife der Simulation, Update des Fensters'
    self.check_events()
    self.simulation.update(1 / self.FPS)
    self.screen.fill(self.white)
    if self.show_visualisation:
        self.font = pygame.font.SysFont('Arial', round(2 * self.scaling))
        # Darstellen aller Strassen
        for road in self.simulation.roads:
            self.draw_road(road)
        # Darstellen aller Kreuzungen
        for intersection in self.simulation.intersections:
            self.draw_intersection(intersection)
        # Darstellen aller Markierungen
        if self.show_signs:
            for road in self.simulation.roads:
                for lane in road.lanes_0 + road.lanes_1:
                    self.draw_signs(lane)
        # Darstellen aller Fahrzeuge
        for road in self.simulation.roads:
            for lane in road.lanes_0 + road.lanes_1:
                for vehicle in lane.vehicles:
                    self.draw_vehicle(vehicle)
    self.draw_stats()

    pygame.display.update()
    self.clock.tick(self.FPS * self.time_acceleration)
```

Code 24: Update des Fensters und der Simulation

Zuerst wertet die Methode alle Ereignisse aus. Dann wird die Simulation für den Zeitabstand `1/self.FPS` aktualisiert. Im nächsten Schritt wird das Fenster weiss gefüllt, sodass nacheinander alle Strassen, Kreuzungen, Markierungen und Fahrzeuge abgebildet werden können. Diese Reihenfolge ist wichtig, da sonst nicht alle Elemente sichtbar sind. Die Methode `self.draw_stats` zeigt einige Statistiken wie Zeit und Anzahl Fahrzeuge am oberen Bildschirmrand an. Schliesslich wird das Fenster aktualisiert, womit alle Änderungen sichtbar werden. Mit `self.clock.tick` wird sichergestellt, dass die Simulation nicht schneller als die eingestellte Geschwindigkeit läuft.

2.11 Datenspeicherung

Beim Starten der Simulation wird das Strassennetzwerk erstellt. Die Daten dazu müssen irgendwo gespeichert werden. Für meine Testläufe während der Entwicklung erstellte ich jede Strasse und jede Kreuzung manuell am Ende des Programms. Diese Methode wäre allerdings sehr ineffizient, wenn ein grosses Netzwerk aufgebaut werden soll. Stattdessen sollten die Daten in externen Dateien gespeichert werden, die vom Programm eingelesen werden können. Mir standen dabei zwei Dateitypen zur Auswahl, mit denen ich bereits vertraut war: CSV und JSON.

Zuerst dachte ich, dass CSV die bessere Lösung sei. CSV-Dateien sind durch Kommas abgetrennte Werte, die zusammen eine Tabelle bilden. Beispielsweise könnte eine CSV-Datei mit allen Strassen gebildet werden. Jede Zeile wäre dann eine Strasse und jede Spalte würde eine bestimmte Eigenschaft der Strasse beschreiben. Eigenschaften wären z. B. Startposition, Endposition, Höchstgeschwindigkeit und Anzahl Fahrspuren. Hier taucht aber schon das erste Problem auf: Wenn die einzelnen Fahrspuren selbst noch genauere Spezifikationen wie z. B. die erlaubten Abzweigungsrichtungen benötigen, ist das mit CSV nicht oder nur sehr umständlich möglich. CSV-Dateien erlauben nämlich keine Verschachtelungen. Man kann also nicht einfach ein Element zu einer Liste von mehreren Elementen machen. Natürlich kann man versuchen, mit eckigen Klammern eine Liste zu bilden, aber der CSV-Reader von Python würde diese nicht als Liste erkennen.

Als ich merkte, dass ich Verschachtelungen benötigte, wechselte ich zu JSON. Glücklicherweise war meine Methode zum Einlesen von CSV-Dateien noch nicht sehr umfangreich und konnte daher schnell ersetzt werden. JSON (JavaScript Object Notation) ist ein von Programmiersprachen unabhängiges Dateiformat, um strukturierte Daten in Textform zu speichern. Es gibt auch Erweiterungen, beispielsweise das Format GEOJSON, das ich später verwenden werde. JSON-Dateien haben eine sehr ähnliche Syntax wie Listen und Dictionaries in Python. Da sie aus Listen und Dictionaries bestehen, sind Verschachtelungen problemlos möglich. Die einzelnen Elemente werden zwar anders genannt, das ist jedoch unwichtig.

2.11.1 Struktur meiner JSON-Dateien

Für das vollständige Strassennetzwerk werden drei JSON-Dateien benötigt: Eine für Kreuzungen, eine für Strassen und eine für Lichtsignale. Letztere könnte zwar auch bei den Kreuzungen integriert werden, dies wäre aber zu unübersichtlich. Die Struktur der Dateien muss folgendermassen aussehen:

Kreuzungen

```
[
    {
        "index": 0,
        "posx": 10,
        "posy": 10,
        "turning_speed": 30,
        "vehicles_per_hour": 50
    },
    {
        "index": 1,
        "posx": 100,
        "posy": 10,
        "turning_speed": 30,
        "vehicles_per_hour": 50
    }
]
```

Code 25: Beispiel der Datei «Intersections.json»

Die Datei «Intersections.json» enthält alle Kreuzungen als Dictionaries. Die Kreuzungen werden nummeriert und haben eine Position. Zusätzlich haben sie eine maximale Abbiegegeschwindigkeit (in km/h) und eine Zahl, die angibt, wie viele Fahrzeuge dort pro Stunde generiert werden sollen. Im späteren Verlauf habe ich nämlich einen neuen Fahrzeuggenerator erstellt, der an den Kreuzungen zufällig Fahrzeuge generiert. Die Wahrscheinlichkeit, dass dies passiert, wird durch die Zahl `"vehicles_per_hour"` bestimmt. Auch die Destination der neuen Fahrzeuge wird so ausgewählt, dass am Schluss an jeder Kreuzung etwa gleich viele Fahrzeuge ankommen, wie dort generiert werden.

Strassen

```
[
    {
        "start_intersection": 0,
        "end_intersection": 1,
        "speed_limit": 50,
        "lanes_0": [
            {
                "priority": true,
                "width": 4,
                "turning_directions": [
                    -1,
                    0,
                    1
                ],
                "index": 0
            }
        ],
        "lanes_1": [
            {
                "priority": false,
```

```
                "width": 4,
                "turning_directions": [
                    -1,
                    0,
                    1
                ],
                "index": 1
            }
        ]
    }
]
```

Code 26: Beispiel der Datei «Roads.json»

In der Datei «Roads.json» sind alle Strassen gespeichert. Die Start- und Endpunkte einer Strasse werden nicht durch Positionen angegeben, sondern durch die Nummern der Kreuzungen, mit denen sie verbunden ist. Die Höchstgeschwindigkeit wird ebenfalls in km/h angegeben. Für die Fahrspuren gibt es wie im Python-Code zwei Listen für die beiden Richtungen. Die Fahrspuren sind jeweils eigene Dictionaries mit Angaben zum Vortritt, der Spurbreite und den erlaubten Abzweigungsrichtungen. Auch sie müssen nummeriert werden, da bei den Lichtsignalen darauf verwiesen wird.

Lichtsignale

```
[
    {
        "intersection": 1,
        "phases": [
            {
                "time": 20,
                "active_lanes": [
                    0
                ]
            },
            {
                "time": 3,
                "active_lanes": []
            },
            {
                "time": 20,
                "active_lanes": [
                    1
                ]
            },
            {
                "time": 3,
                "active_lanes": []
            },

        ]
    }
]
```

Code 27: Beispiel der Datei «Traffic_lights.json

Die Datei «Traffic_lights.json» enthält den Phasenplan für jede Lichtsignalanlage und die Kreuzung, die sie kontrolliert. Für jede Phase werden die Zeit und die Nummern der Fahrspuren, die grün haben, angegeben.

2.11.2 Einlesen der JSON-Dateien

Die Methode `json_reader` in der `class Simulation` habe ich erstellt, um aus den drei JSON-Dateien das Strassennetzwerk aufzubauen. Ihre Grundstruktur sieht folgendermassen aus:

```
def json_reader(
    self,
    intersections: str='Intersections.json',
    roads: str='Roads.json',
    traffic_lights: str='Traffic_lights.json'
):
    'Auslesen und erstellen des Strassennetzwerkes aus json-Dateien'
    # Erstellen aller Kreuzungen
    with open(intersections, 'r') as jsonfile:
        data = json.load(jsonfile)

        for intersection in data:
            # Erstellen der Kreuzung

    # Erstellen aller Strassen
    with open(roads, 'r') as jsonfile:
        data = json.load(jsonfile)

        for road in data:
            # Erstellen der Strasse

    # Verbinden aller Strassen und Kreuzungen
    self.connect_roads()

    # Erstellen aller Lichtsignalanlagen
    with open(traffic_lights, 'r') as jsonfile:
        data = json.load(jsonfile)
        for traffic_light in data:
            # Erstellen der Lichtsignalanlage
```

Code 28: Grundstruktur der Methode json_reader

Die Dateien werden mit dem Befehl `with open` geöffnet, sodass das json-Modul diese in Python-Syntax übersetzen kann. Das Modul muss am Anfang des Programms mit `import json` importiert werden. Zuerst werden alle Kreuzungen erstellt, dann die Strassen mit ihren Fahrspuren. Mit der Methode `self.connect_roads()` werden schliesslich alle Strassen und Kreuzungen miteinander verbunden. Zum Schluss werden noch alle Lichtsignalanlagen erstellt.

Die Methode zum Einlesen der Dateien muss nur einmal, nämlich beim Erstellen der Simulation, ausgeführt werden. Zusätzlich wird sie noch ausgeführt, wenn die Simulation mit der Taste «R» zurückgesetzt wird.

2.12 Verschönerung des Codes

Wie bereits erwähnt, sollte mein Code die PEP 8[17] einhalten. Bis zum Ende achtete ich jedoch kaum darauf; erst zuletzt installierte ich Pylint[18] aus dem «Extension Marketplace» von Visual Studio Code. Ich bekam viele Vorschläge zur Verbesserung des Codes. Einige davon waren jedoch falsch, da Pylint das Modul Pygame offenbar nicht kannte. Hier sind einige Fehler, die von Pygame entdeckt wurden:

- Ich verwendete Docstrings an Orten, an denen Kommentare verwendet werden sollten, beispielsweise bei Schleifen oder if-Bedingungen. Docstrings sind nur direkt nach der Definition einer Funktion, einer Klasse oder am Anfang des Programms vorgesehen.
- Bei einigen Methoden hatte ich hingegen den Docstring vergessen.
- Zwei Fehlertypen wurden nach `except` mit `or` verglichen.
- Die Systemvariable `type` wurde neudefiniert.
- Der Name `Traffic_light` entsprach nicht dem Namensstil für Klassen. Daher wurde die Class zu `class TrafficLight` umbenannt.
- Viele Leerzeilen hatten noch unnötige Leerschläge.
- Am Ende des Programms musste eine Leerzeile eingefügt werden.
- Einige Zeilen waren zu lang. Wenn ein Ausdruck in Klammern gesetzt wird, kann er aber auf mehrere Zeilen aufgeteilt werden. So konnte ich die Zeilenlänge verkürzen.
- Um mathematische Operationszeichen herum hatte ich nicht konsequent Leerschläge gesetzt.

Ebenfalls achtete ich darauf, möglichst konsequent Type Hints einzusetzen. Diese geben an, welchen Typ eine Variable haben sollte, erzwingen diesen aber nicht. In einigen Fällen, in denen ich Type Hints nicht sinnvoll fand, verzichtete ich darauf, beispielsweise wenn eine Funktion den Output `None` hat.

Mit der Verständlichkeit des Codes bin ich grösstenteils zufrieden. Bei komplizierten Funktionen habe ich Kommentare eingefügt, um sie leichter verständlich zu machen. Trotzdem bin ich noch ein Anfänger und mache sicherlich mehr Fehler als erfahrene Programmierer*innen.

[17] van Rossum, Guido; Warsaw, Barry; Coghlan, Alyssa: *PEP 8 – Style Guide for Python Code*. Auf: https://peps.python.org/pep-0008/ (abgerufen am 19.10.2024).
[18] Pylint 2024.

3 Simulation der Stadt Wetzikon

Wetzikon ist eine Gemeinde mit ca. 26’500 Einwohner*innen[19] im Zürcher Oberland. Ich wählte diese Stadt für meine Verkehrssimulation, da sie eine angemessene Grösse hat und ich sie bereits gut kannte. Ausserdem wusste ich, dass einige Strassen in Wetzikon stark belastet sind und dort oft Staus entstehen. Es gibt also Verbesserungspotential.

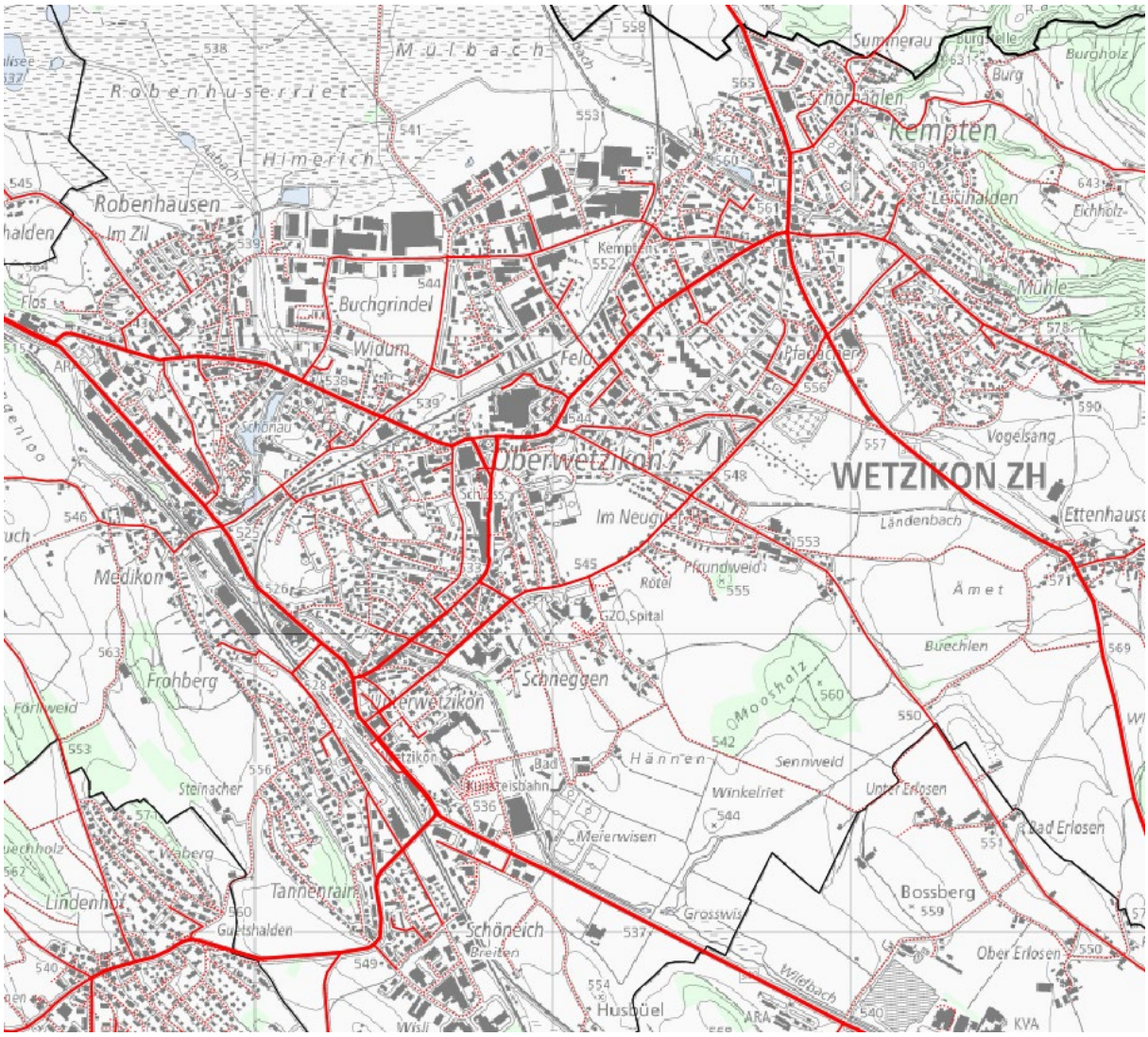


Abbildung 7: Verkehrsnetz der Stadt Wetzikon

Die Stadt wird von den beiden Verkehrsachsen Aathal-Hinwil im Südwesten und Pfäffikon-Hinwil im Nordosten durchzogen. Zwischen diesen Achsen gibt es viel Durchgangsverkehr, der die Strassen von Wetzikon zusätzlich belastet.

[19] Wikipedia: *Wetzikon*. Auf: https://de.wikipedia.org/wiki/Wetzikon (abgerufen am 20.10.2024).

3.1 Übernahme des Strassennetzes in die Simulation

Um den Verkehr von Wetzikon simulieren zu können, musste ich das Netzwerk von Strassen und Kreuzungen zuerst in meine Simulation übertragen. Dies war ein äusserst aufwendiger Prozess, da ich nur einen kleinen Teil davon automatisieren konnte. Als Datenquelle benutzte ich den GIS-Browser des Kantons Zürich[20], da dort die genauesten Daten zum Verkehrsnetz im Kanton Zürich verfügbar sind.

3.1.1 Arbeit mit dem GIS-Browser

Der GIS-Browser hat einen Kartenkatalog, aus dem man die benötigten Karten auswählt. Es können dabei auch mehrere Karten überlagert angezeigt werden. Die folgenden Karten wurden von mir verwendet:

- «Gesamtverkehrsmodell (GVM-ZH)» für das Strassennetz
- «Orthofoto SWISSIMAGE 2022» für genaue Informationen zu Fahrspuren, Strassenmarkierungen, etc.
- «Signalisierte Geschwindigkeit Kantonsstrassen» für Höchstgeschwindigkeiten auf den Kantonsstrassen
- «Tempo30- und Begegnungszonen (verfügt)» für alle Strassen mit Tempo 30.
- «Verkehrstechnik (BSA)» für Lichtsignalanlagen und Verkehrsmessstellen

Es werden Möglichkeiten zum Export von Daten angeboten. Diese sind jedoch nur für spezialisierte GIS-Programme geeignet, in die ich mich nicht einarbeiten wollte. Mir fiel aber auf, dass mit dem Tool «Zeichnen» Punkte gezeichnet und anschliessend als GEOJSON-Datei exportiert werden können. Diese Dateien können von Python wie normale JSON-Dateien eingelesen werden. Später konnte ich ein Programm schreiben, dass daraus die Datei «Intersections.json» erstellt. Ich hatte also eine Möglichkeit gefunden, um wenigstens die Kreuzungen aus dem GIS-Browser in meine Simulation zu exportieren.

Sogleich begann ich mit dem Markieren aller Kreuzungen im GIS-Browser. Ich konzentrierte mich dabei auf die Hauptverkehrsstrassen, und erfasste nicht jede einzelne Kreuzung in unwichtigen Quartierstrassen. Da in der Simulation nur gerade Strassen möglich sind, mussten auch Kurven als Kreuzungen markiert werden. Bei Kreiseln musste ich ausserdem jede Ausfahrt einzeln markieren, damit später ein Kreis gebildet werden konnte. Auch bei einer Änderung der Anzahl Fahrspuren musste ich einen Punkt setzen, beispielweise wenn es mehrere Einspurstreifen vor einer Kreuzung gibt.

[20] Kanton Zürich: *GIS-Browser*. Auf: https://geo.zh.ch/maps?x=2702895&y=1242852&scale=15896&basemap=arelkbackgroundzh (abgerufen am 20.10.2024).

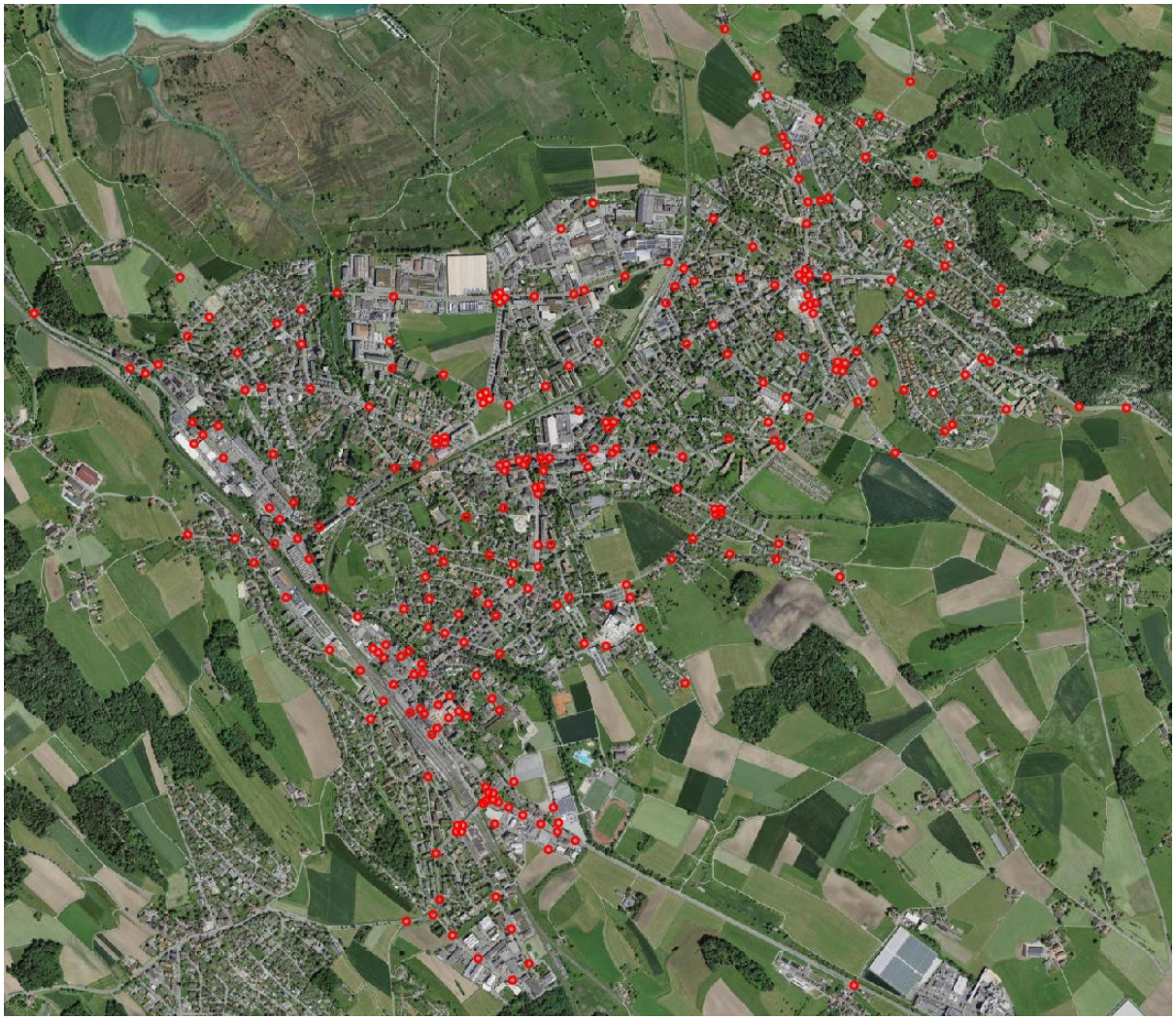


Abbildung 8: Luftbildkarte mit allen markierten Kreuzungen

Das Resultat waren über 270 Punkte, die ich später alle manuell verbinden musste. Die Koordinaten Punkte wurden vom GIS-Browser heruntergeladen. Dabei entstand die Datei «drawings.geojson», welche die Koordinaten aller Punkte enthält.

3.1.2 Konvertieren der Punkte

Die Koordinaten der Punkte werden vom GIS-Browser in Breiten- und Längengraden angegeben. Diese geografischen Koordinaten müssen für die Simulation in Meter umgerechnet werden. Dafür wird zuerst ein Punkt festgelegt, der die Position Null hat. Dann werden für jeden anderen Punkt die Koordinaten im Verhältnis zu diesem Referenzpunkt berechnet. In meinem Fall definierte ich den Referenzpunkt beim Bahnhof Wetzikon.

Die Erde ist eine Kugel, somit kann ihre Oberfläche nicht exakt auf einem zweidimensionalen Koordinatensystem abgebildet werden. Es müssen immer Kartenprojektionen angewendet werden, die einen Teil der Oberfläche verzerren. Bei einer kleinen Fläche wie der von Wetzikon sind die Verzerrungen jedoch völlig irrelevant.

Der Abstand zwischen zwei Breitenkreisen, die sich um ein Grad unterscheiden, beträgt immer konstant 111.3 km[21]. Bei den Längenkreisen ist dies nur am Äquator der Fall, da sie bis zu den Polen immer näher zusammenkommen. Der Abstand von 111.3 km lässt sich aber mit dem Cosinus der geografischen Breite multiplizieren, um den Abstand der Längenkreise an einem bestimmten Punkt zu berechnen. Somit wäre er am Nordpol 0 und am Äquator genau 111.3 km.

Wir berechnen also den Abstand der Längenkreise beim Referenzpunkt und gehen davon aus, dass dieser über die ganze Stadt konstant bleibt.

```
'Programm zur Umrechnung der Koordinaten aus dem GIS-Browser in unser Koordinatensystem'

import json
import math

with open('drawings.geojson', 'r') as jsonfile:
    data = json.load(jsonfile)

reference_point = (8.7912009, 47.3192026)
positions: list = []
for point in data['features']:
    geo_cords = point['geometry']['coordinates']
    posx = ((geo_cords[0] - reference_point[0]) * 111.3 * 1000
        * math.cos(math.radians(reference_point[1])))
    posy = (reference_point[1] - geo_cords[1]) * 111.3 * 1000
    position = (posx, posy)
    positions.append(position)

intersections: list = []
index: int = 0
for position in positions:
    intersection: dict = {
        'index': index,
        'posx': position[0],
        'posy': position[1],
        'turning_speed': 30
    }
    intersections.append(intersection)
    index += 1

with open('Intersections_new.json', 'x') as jsonfile:
    json.dump(intersections, jsonfile, indent=4)
```

Code 29: Umrechnung der geografischen Koordinaten in Simulationskoordinaten

Das Programm «coordinate_converter.py» (Code 29) schrieb ich, um diese Berechnung durchzuführen. Zuerst wird die Datei «drawings.geojson» eingelesen. Dann werden die eben beschriebenen Berechnungen für jeden Punkt durchgeführt. Die erhaltenen Koordinaten werden in der Liste `positions` gespeichert. Im nächsten Schritt wird für jeden Punkt ein Dictionary mit der entsprechenden Position und allen weiteren Parametern für eine Kreuzung erstellt. Schliesslich werden alle neu erstellten Kreuzungen in der Datei «intersections_new.json» gespeichert.

[21] Kompf, Martin: *Entfernungsberechnung*. Auf: https://www.kompf.de/gps/distcalc.html (abgerufen am 20.10.2024).

3.1.3 Erstellen der Strassen

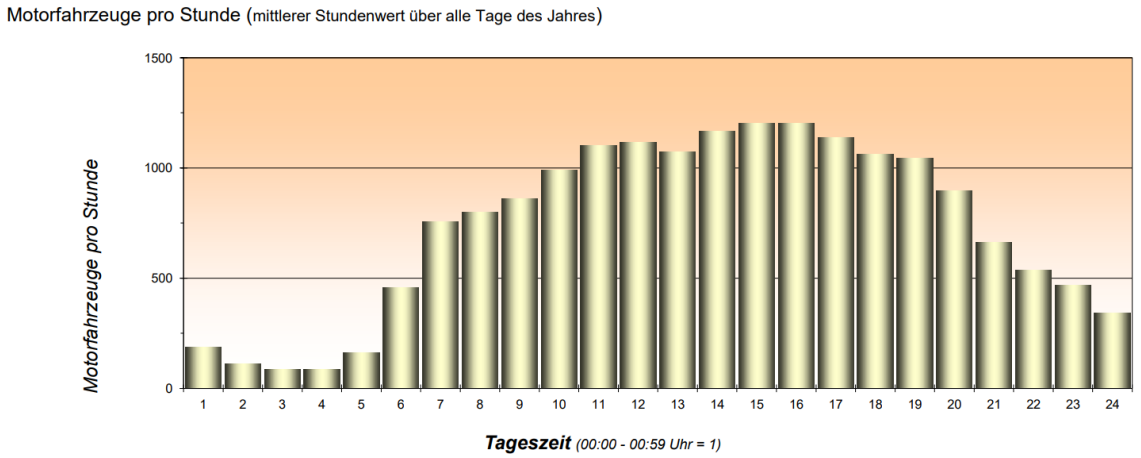
Als Nächstes musste die Datei «Roads.json» mit allen Strassen erstellt werden. Es blieb mir nichts anderes übrig, als für jede Strasse einzeln einen Eintrag in der JSON-Datei zu erstellen. Die Simulation konnte auch schon ohne Strassen ausgeführt werden, daher erstellte ich die Funktion, dass beim Mausklick auf eine Kreuzung deren Nummer herausgegeben wird. Damit konnte ich die Nummern der Kreuzungen herausfinden, die verbunden werden sollten. Mit dem zur Verfügung stehenden Kartenmaterial konnte ich dann die Höchstgeschwindigkeit und die Anzahl Fahrspuren einer Strasse ermitteln. Auf dem Satellitenbild konnte ich ablesen, welche Fahrspuren Vortritt haben und welche Abzweigungsrichtungen erlaubt sind, da dies normalerweise auf der Strasse markiert wird. Bei Unklarheiten zog ich Google Street View oder meine eigenen Erfahrungen zur Hilfe.

Abbildung 9: Das fertige Strassennetz von Wetzikon in der Simulation

3.1.4 Generation von neuen Fahrzeugen

Es wurde bereits erklärt, dass der Wert `"vehicles_per_hour"` festlegt, wie viele Fahrzeuge an einer Kreuzung pro Stunde generiert werden. Nun mussten dafür realistische Werte gefunden werden. Zuerst setzte ich den Wert aber für alle Kreuzungen auf 0. Schliesslich sollten nur dort Fahrzeuge generiert werden, wo auch wirklich Fahrzeuge herkommen. Das heisst vor allem an den Strassen, die in die Stadt hineinführen, aber auch innerhalb der Stadt in Wohn- oder Industriegebieten.

An den wichtigen Verkehrsachsen gibt es Messstellen für den motorisierten Individualverkehr, deren Daten im GIS-Browser frei zugänglich sind. Dort kann der durchschnittliche Stundenwert für alle Tageszeiten abgelesen werden. Als Beispiel ist hier die Verkehrsmessstelle Hinwil, Zürichstrasse dargestellt:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Querschnitt	188	111	87	87	161	456	758	800	861	988	1'101	1'116	1'074	1'168	1'202	1'203	1'138	1'063	1'044	897	661	536	468	340
Zürich	72	45	34	46	104	289	397	386	416	479	531	540	514	557	573	582	546	502	495	449	311	235	195	130
Hinwil	116	65	53	40	56	168	361	414	445	509	570	576	560	611	629	620	591	561	549	448	350	301	273	210

Abbildung 10: Daten der Verkehrsmessstelle Hinwil, Zürichstrasse. Für uns relevant ist die Richtung «Zürich», da diese Fahrzeuge durch Wetzikon fahren.

Ich entschied mich für die Nutzung der Spitzenwerte, da das Verkehrsnetz so auf seine maximale Auslastung untersucht werden konnte. Insgesamt benutzte ich sechs verschiedene Verkehrsmessstellen. Damit konnten bereits alle wichtigen Zuläufe abgedeckt werden. Für die restlichen Zulaufstrassen nahm ich selbst sinnvolle Werte an, die aber auf jeden Fall kleiner waren.

Auch für die Wohn- und Industriequartiere innerhalb der Stadt musste ich die Werte abschätzen. In Wohnquartieren gab ich den Kreuzungen Werte zwischen 5 und 10 Fahrzeugen pro Stunde, in Industriequartieren zwischen 10 und 40 Fahrzeugen pro Stunde. Manchmal wählte ich auch höhere Werte, vor allem wenn ich nicht alle Strassen in einem Quartier erschlossen hatte, oder bei grossen Anlagen wie Einkaufszentren.

3.1.5 Erstellen der Lichtsignalzeitpläne

Erst beim Erstellen der Lichtsignalzeitpläne fiel mir auf, dass die einzelnen Fahrspuren in der JSON-Datei noch nicht nummeriert waren, ich aber eindeutige Nummern benötigte. Da ich bereits sehr viele Strassen mit ihren Fahrspuren erstellt hatte, schrieb ich kurzerhand ein einfaches Programm, um diesen Prozess zu automatisieren und Zeit zu sparen.

Die Lichtsignalanlagen sind zwar in der Karte «Verkehrstechnik (BSA)» vermerkt, jedoch gibt es keine Angaben dazu, mit welchem Zeitplan diese gesteuert werden. Wahrscheinlich aus dem Grund, dass die Zeitpläne in der Realität dynamisch geregelt werden.

Ich musste also meine eigenen Zeitpläne entwickeln, um einen möglichst guten Verkehrsfluss zu gewährleisten.

Jede Lichtsignalanlage hat eine unterschiedliche Konfiguration, deshalb kann kein allgemeiner Phasenplan erstellt werden. Grundsätzlich sollte es aber möglichst wenig Phasen geben, während denen jeweils möglichst viele Fahrspuren grün haben. Dabei gehe ich davon aus, dass die Fahrzeuge bei grün freie Fahrt haben und keine Konflikte entstehen können. Mir ist aufgefallen, dass gerade so viele Phasen benötigt werden, wie Strassen in die Kreuzung einmünden. Die folgende Darstellung zeigt alle Phasen der Kreuzung Zürcher-/West-/Bertschikerstrasse in Wetzikon:

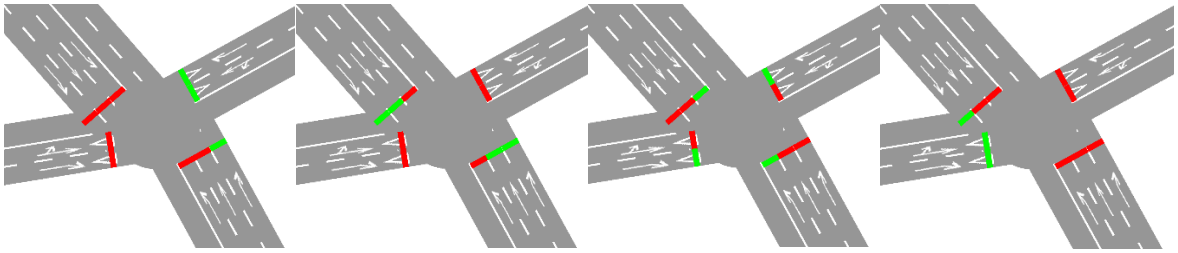


Abbildung 11: Alle Phasen einer Lichtsignalanlage mit vier Einmündungen. Es haben zwar mehrere Fahrspuren gleichzeitig grün, die Wege der Fahrzeuge kreuzen sich aber nicht.

Zwischen jeder Phase baute ich noch eine kurze Rotphase ein, damit alle gefährlichen Fahrzeuge die Kreuzung vor der nächsten Grünphase verlassen können. Ebenfalls achtete ich darauf, dass die Zeitpläne mehrerer nahe beieinander gelegener Lichtsignalanlagen synchronisiert sind, sodass der Verkehr auf der Hauptverkehrsstrasse gleichzeitig durch beide Kreuzungen fahren kann.

3.2 Ergebnisse der Simulation

Meine Simulation zeigte, dass das Verkehrsnetz der Stadt Wetzikon durch den Spitzenverkehr an einigen Stellen überlastet wurde. Bei manchen Durchgängen trat nach wenigen Stunden Simulationszeit ein Verkehrsinfarkt auf. Ein Verkehrsinfarkt ist ein totaler Stillstand des Verkehrs, der sich nicht von selbst wieder auflösen kann.

Ein Teil des Problems lag sicherlich in der Grösse der Simulation. Schliesslich wurde nur das Strassennetzwerk der Stadt Wetzikon simuliert, alle Autos mussten also durch die Stadt fahren. In der Realität würden die Fahrer*innen versuchen, die Stadt bei Stau zu umfahren. Ausserdem lief die Simulation mehrere Stunden bei Spitzenverkehr, der in der Realität nach einer Stunde wieder abnehmen würde.

Einige der Problemstellen, an denen besonders viel Stau entstanden ist, werden folglich etwas genauer betrachtet. Für jede Problemstelle werde ich auch eine mögliche Lösung vorschlagen.

3.2.1 Kreisel Rapperswiler-/Grüningerstrasse

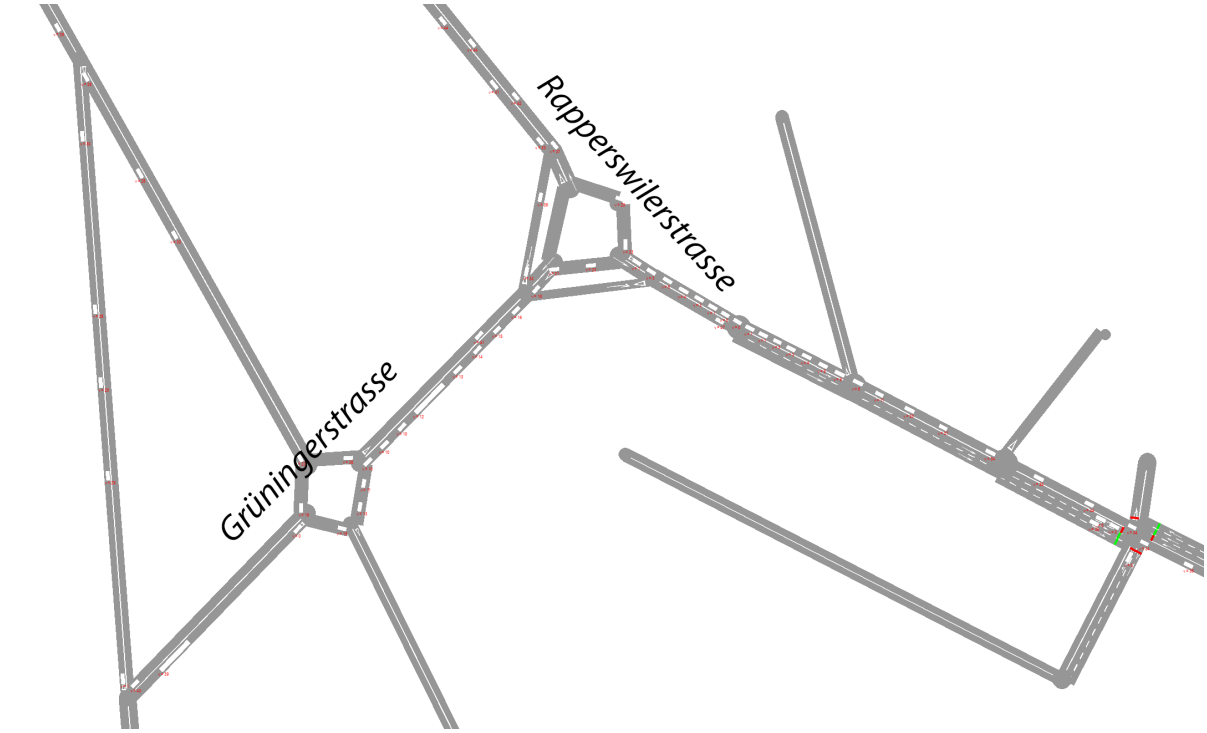


Abbildung 12: Rückstau am Kreisel Rapperswiler-/Grüningerstrasse

Der Kreisel an der Rapperswilerstrasse hat vor einigen Jahren eine Kreuzung mit Lichtsignalanlage ersetzt. Es zeigt sich, dass dieser mit dem Spitzenverkehr an seine Kapazitätsgrenzen stösst, aber noch damit zurechtkommt. Aus der Richtung Hinwil bildet sich ein kleiner Rückstau, da die Fahrzeuge von dort fast ungehindert bis zum Kreisel durchgelassen werden. Im Gegensatz dazu gibt es keinen Stau aus der Gegenrichtung. Der Grund dafür könnte sein, dass die Lichtsignalanlagen an der Bahnhofstrasse weniger effizient sind und den Verkehr zurückhalten. An der Grüningerstrasse bildet sich ebenfalls ein Rückstau, der teilweise den nächsten Kreisel blockiert.

Der Kreisel an der Rapperswilerstrasse bietet keine weiteren Optimierungsmöglichkeiten. Stattdessen sollte das Problem der Verkehrsüberlastung an anderen Stellen angegangen werden.

3.2.2 Bahnhof Wetzikon

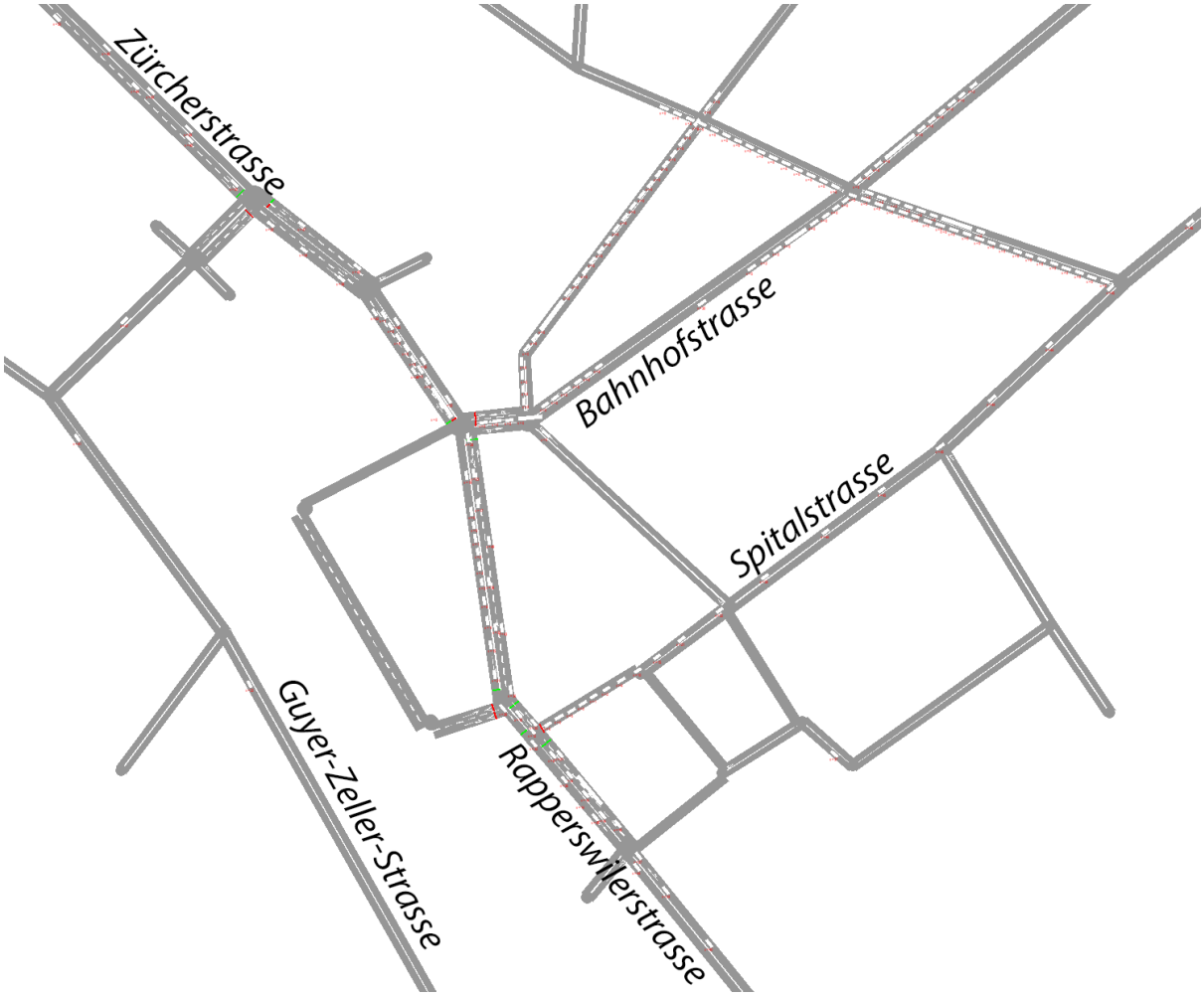


Abbildung 13: Stau an der Spital- und Bahnhofstrasse

In diesem Gebiet tritt das grösste Problem auf: Viele Fahrzeuge kommen via Spital- oder Bahnhofstrasse aus dem Nordosten der Stadt und möchten in die bereits überlastete Achse Aathal-Hinwil einbiegen. Dabei entstehen Staus an der Spital- und Bahnhofstrasse. Wenn die Wartezeiten zu lang werden, versuchen die Fahrzeuge, auf die jeweils andere Strasse auszuweichen. Dafür benutzen sie die kleinen Verbindungsstrassen, die jedoch nicht vortrittsberechtigt sind und daher verstopft werden. Ein weiteres Problem tritt auf, wenn zu viele Fahrzeuge aus der Zürcherstrasse links in die Bahnhofstrasse abbiegen möchten. Dann überfüllen sie den Einspurstreifen und blockieren den Weg für Fahrzeuge, die geradeaus fahren möchten.

Zur Lösung des Problems würde ich einen Kreisel an der Kreuzung Zürcher-/Bahnhofstrasse vorschlagen, da Kreisel wesentlich effizienter sind und obendrein noch die Verkehrssicherheit erhöhen. Dafür müsste jedoch geprüft werden, ob die Platzverhältnisse ausreichen, da das Gebiet dicht bebaut ist. Zusätzlich könnte eine Tempo-30-Zone auf der Spitalstrasse dazu beitragen, den Verkehr auf die Bahnhofstrasse umzuleiten und damit die Spitalstrasse zu entlasten.

3.2.3 Kreuzung Zürcher-/West-/Bertschikerstrasse

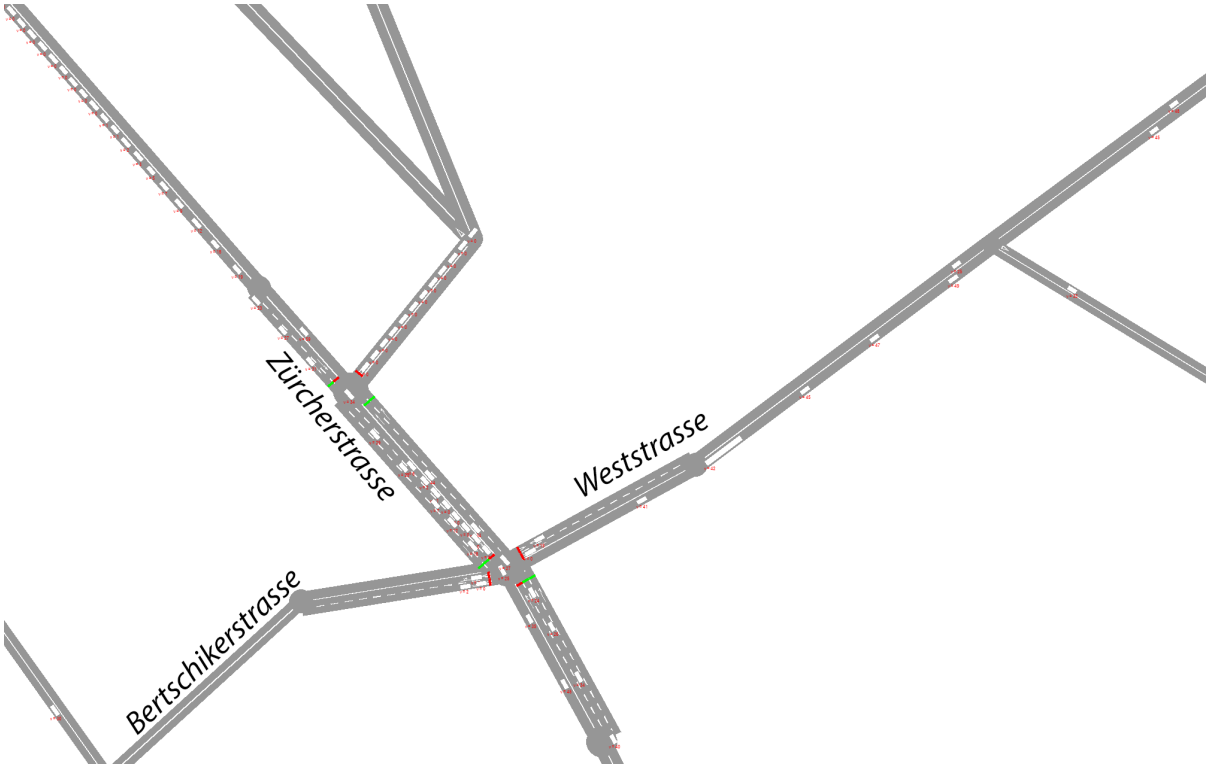


Abbildung 14: Stau an der Zürcherstrasse

An der Zürcherstrasse bildet sich ein langer Stau vor der Kreuzung Zürcher-/West-/Bertschikerstrasse. Der Grund dafür ist das sehr hohe Verkehrsaufkommen aus dem Aathal. Dort endet nämlich die Oberlandautobahn, die beim Betzholzkreisel in Hinwil fortgesetzt wird. Auch an der Weststrasse bildet sich manchmal ein Stau, da diese sehr kurze Grünzeiten erhält.

Eine Lückenschliessung der Autobahn ist in Planung, wird aber noch viele Jahre dauern. Wetzikon soll dabei einen eigenen Autobahnanschluss erhalten. Damit würde die gesamte Verkehrsachse Aathal-Hinwil vom Durchgangsverkehr entlastet.

3.2.4 Kreisel in Kempten

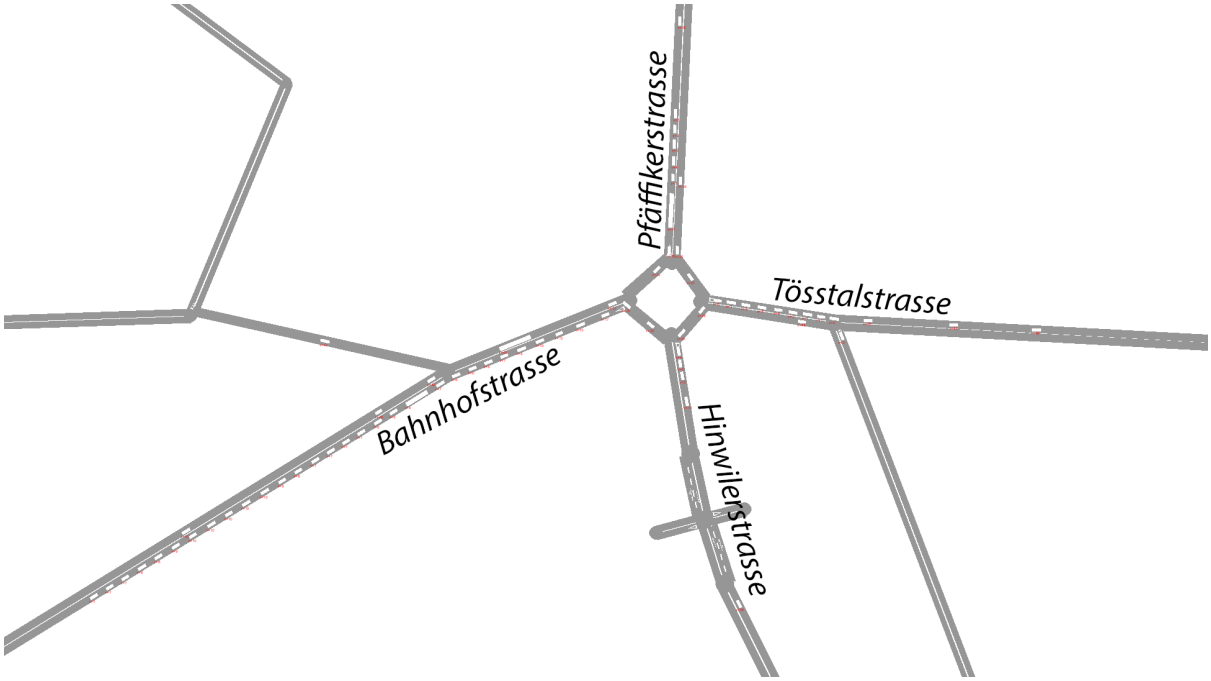


Abbildung 15: Stau an einem Kreisel in Kempten

Dieser Kreisel liegt auf der Achse Pfäffikon-Hinwil. Diese hat an sich schon ein hohes Verkehrsaufkommen. Gleichzeitig kommen aber auch viele Fahrzeuge aus der Bahnhofstrasse, die in diese Achse einbiegen oder in die Tösstalstrasse weiterfahren möchten. Da die Fahrzeuge aus der Pfäffikerstrasse fast durchgehend den Kreisel besetzen, entsteht ein Stau an der Bahnhofstrasse.

Die Lösung hierfür wäre eine Umfahrung von der Pfäffikerstrasse direkt bis an die Zürcherstrasse, ohne dabei das Stadtzentrum von Wetzikon durchfahren zu müssen. Tatsächlich ist eine solche Umfahrung bereits in Planung, sie soll an die Weststrasse anschliessen.

4 Schlusswort

Nun habe ich meine Maturitätsarbeit abgeschlossen. Mit meiner Leistung bin ich grösstenteils zufrieden, vor allem, was das Programmieren angeht: Ich habe erfolgreich eine mikroskopische Verkehrssimulation entwickelt, die den motorisierten Individualverkehr einer Stadt einigermassen realitätsgetreu abbilden kann. Die Simulation habe ich erfolgreich am Beispiel der der Stadt Wetzikon getestet, dabei konnte ich einige Problemstellen identifizieren und Lösungsansätze vorschlagen.

Leider blieb mir am Schluss keine Zeit, um die Lösungsansätze mit der Simulation auf ihre Wirksamkeit zu untersuchen. Damit komme ich auch gleich zum Zeitmanagement: Ich hatte zwar zu Beginn einen Zeitplan erstellt, diesen aber danach nicht mehr berücksichtigt. Die meiste Zeit verbrachte ich mit dem Programmieren, da mir dies am meisten Spass machte. Mit dem Schreiben der Arbeit begann ich erst spät. Zu diesem Zeitpunkt hatte ich das Programm fertig geschrieben und das Verkehrsnetz der Stadt Wetzikon in die Simulation eingebaut, aber die Ergebnisse waren noch nicht ausgewertet. Da das Schreiben länger dauerte als erwartet, musste die Auswertung der Ergebnisse abgekürzt werden. Daraus habe ich gelernt, möglichst früh mit dem Schreiben einer Arbeit anzufangen und immer genug Pufferzeiten einzuplanen.

Für den professionellen Anwendungsbereich gibt es bereits viele Programme zur Verkehrssimulation, mein Programm könnte aber z. B. im Bildungsbereich eingesetzt werden. Dafür könnten auch noch weitere Funktionen hinzugefügt werden.

5 Danksagung

Ein herzlicher Dank geht an meine betreuende Lehrperson, Albert Kern. Er hat mir geholfen, meine Idee auf einen machbaren Rahmen einzugrenzen. Beim Schreiben der Arbeit hat er mir grosszügige Freiheiten gelassen, sodass ich weitestgehend selbstständig arbeiten konnte. Trotzdem war er immer bereit für ein Treffen mit mir und beantwortete meine Fragen innert kürzester Zeit.

Ebenfalls bedanken möchte ich mich bei meiner ehemaligen AM-Lehrperson Beat Jäckle. Er hat mir die Programmiersprache Python mit grösster Begeisterung beigebracht und hat mich motiviert, meine Programmierkenntnisse für diese Maturitätsarbeit wieder zu verwenden.

6 Verzeichnisse

6.1 Literaturverzeichnis

Floss, Theresa: Die Mathematik hinter einem Navigationssystem – *Wie Routenplaner ihre Wege berechnen*. Innsbruck, 2023. Auf: https://www.uibk.ac.at/mathematik/algebra/media/teaching/bachelorarbeit_floss_mathematik_navigation.pdf (abgerufen am 16.10.2024).

Kanton Zürich: *GIS-Browser*. Auf: https://geo.zh.ch/maps?x=2702895&y=1242852&scale=15896&basemap=arelkbackgroundzh (abgerufen am 20.10.2024).

Kompf, Martin: *Entfernungsberechnung*. Auf: https://www.kompf.de/gps/distcalc.html (abgerufen am 20.10.2024).

Microsoft: *Visual Studio Code*. Auf: https://code.visualstudio.com/ (abgerufen am 19.10.2024).

Pygame: *Pygame Front Page*. Auf: https://www.pygame.org/docs/ (abgerufen am 19.10.2024).

Pylint: *Star your Python code*. Auf: https://www.pylint.org/ (abgerufen am 19.10.2024).

Python Software Foundation: *Classes*. Auf: https://docs.python.org/3/tutorial/classes.html (abgerufen am 19.10.2024).

Python Software Foundation: *The Python Tutorial*. Auf: https://docs.python.org/3/tutorial/ (abgerufen am 19.10.2024).

The Geany contributors: *About Geany*. Auf: https://www.geany.org/about/geany/ (abgerufen am 19.10.2024).

Touring Club Schweiz: *Der Vortritt und ich*. Auf: https://www.tcs.ch/mam/Verkehrssicherheit/PDF/Booklets/der-vortritt-und-ich.pdf (abgerufen am 17.10.2024).

Touring Club Schweiz: *Fahren auf der Autobahn – das sollten Sie wissen*. Auf: https://www.tcs.ch/de/testberichte-ratgeber/ratgeber/auto/autobahn-fahren.php (abgerufen am 15.11.2024)

Treiber, Martin; Kesting, Arne; Helbing, Dirk: *MOBIL: General Lane-Changing Model for Car-Following Models*. Dresden, 2006. Auf: https://mtreiber.de/publications/MOBIL_TRB.pdf (abgerufen am 14.10.2024).

Treiber, Martin; Kesting, Arne: *Verkehrsdynamik und -simulation. Daten, Modelle und Anwendungen der Verkehrsflussdynamik*. Springer, Dresden, 2010.

Treiber, Martin: *The Intelligent-Driver Model and its Variants*. Auf: https://traffic-simulation.de/info/info_IDM.html (abgerufen am 14.10.2024).

Treiber, Martin: *The Lane-change Model MOBIL*. Auf: https://traffic-simulation.de/info/info_MOBIL.html (abgerufen am 14.10.2024).

van Rossum, Guido; Warsaw, Barry; Coghlan, Alyssa: *PEP 8 – Style Guide for Python Code*. Auf: https://peps.python.org/pep-0008/ (abgerufen am 19.10.2024).

Weibel, Benedikt: *Wir Mobilitätsmenschen. Wege und Irrwege zu einem nachhaltigen Verkehr*. NZZ Libro, Basel, 1. Auflage, 2021.

Wikipedia: *Bildtafel der Strassensignale in der Schweiz und in Liechtenstein seit 2023*. Auf: https://de.wikipedia.org/wiki/Bildtafel_der_Strassensignale_in_der_Schweiz_und_in_Liechtenstein_seit_2023 (abgerufen am 19.10.2024).

Wikipedia: *Wetzikon*. Auf: https://de.wikipedia.org/wiki/Wetzikon (abgerufen am 20.10.2024).

6.2 Abbildungsverzeichnis

6.3 Codeausschnitte

7 Anhang

7.1 Zugriff auf das Programm

Der gesamte Programmcode inkl. der benötigten Dateien kann unter folgendem Link heruntergeladen werden:

https://drive.google.com/file/d/17s9_IFgu9ONz-jVPUWX5m4U_RLWwca6gf/view?usp=sharing

7.2 Versionsprotokoll

Version	Datum	Änderungen
1.0	22.04	- Erstellen einer Fahrzeug-Class - Berechnung der Bewegung eines Fahrzeugs (auf gerader Strasse) mit dem IDM - Anzeigen der Distanz zu stehendem Fahrzeug im Terminal
1.1	23.04	- Simulation einer Fahrzeugkolonne (mehrere Fahrzeuge) - Darstellung der Simulation mit pygame
1.2	24.04	- Verbesserung der grafischen Darstellung (Fenstergrösse anpassbar) - Sortierung der Fahrzeuge in einer Liste (effizienter) - Automatische Generation von neuen Fahrzeugen
2.0	25.04	- Anzeige der Geschwindigkeit und Distanz für jedes Fahrzeug - Einführung von verschiedenen Fahrzeugtypen (Autos und Lastwagen) - Zufällige Bestimmung der Parameter abhängig vom Fahrzeugtyp - Zufällige Generation von Autos oder Lastwagen - Umstrukturierung des Programmcodes (Verwendung von mehreren Klassen) - Experiment mit stehendem Fahrzeug (Blockade)
2.1	06.05	- Darstellung von mehreren Strassen mit Start- und Endpositionen - Fahrzeugsimulation auf jeder Strasse einzeln - Umrechnung der Fahrzeugpositionen, damit sie auf den Strassen korrekt angezeigt werden - Skalierung und Verschiebung der Darstellung mit Pfeiltasten & Mausrad - Pausieren der Simulation mit der Leertaste - Ein- und Ausblenden der Geschwindigkeitsanzeige - Korrektur von inkorrekter Verwendung von Class Variables - Viele kleine Verbesserungen im Programmcode
2.2	12.05	- Implementierung eines Algorithmus zur Routenplanung für Fahrzeuge - Neue Class für Kreuzungen - Erstellen eines Verkehrsnetzwerkes aus Strassen und Kreuzungen - Verbindung von Kreuzungen/Strassen, damit die Fahrzeuge an einer Kreuzung die nächste Strasse auswählen können
2.3	19.05	- Implementierung von mehrspurigen Strassen - Darstellung von jeder Spur einzeln - Momentan kann von Fahrzeugen nur eine Spur benutzt werden (provisorisch) - Darstellung von Markierungslinien zwischen Spuren
2.4	30.05	- Verbesserte Version von 2.3
2.5	13.06	- Fahrzeuge können Strassen in beide Richtungen benutzen - Es wird automatisch die rechte Spur benutzt - Implementierung einer Methode zum Spurwechsel (noch unbenutzt)

		- Verbesserung der Methode zur Verbindung von Strassen und Kreuzungen - Alle Methoden mit Kommentaren versehen zur besseren Übersicht
2.6	16.06	- Erstellen eines Strassennetzwerkes als Beispiel mit einer einfachen Funktion
2.7	27.06	- Fahrzeuge berücksichtigen auch vorausfahrende Fahrzeuge nach der nächsten Kreuzung für die Beschleunigung - Ausbau der Spurwechselmethode
3.0	28.06	- Ermittlung, ob ein Spurwechsel nötig ist mit dem Modell MOBIL - Automatischer Wechsel der Spur, wenn die Kriterien erfüllt sind - Einsetzen des Fahrzeuges an der korrekten Position auf die neue Spur - Weiterfahren auf der gleichen Spur nach der Kreuzung
3.1	30.06	- Verbesserung des Spurwechsels - Neues Strassennetzwerk (Kreis) zum Testen des Spurwechsels - Anpassung der Parameter für realistische Verhältnisse - Fehlerbehebung, um Zusammenstösse zu verhindern (nicht erfolgreich)
3.2	09.07	- Implementierung einer Mindestzeit zwischen Spurwechseln - Ermittlung des hinterherfahrenden Fahrzeuges auch nach einer Kreuzung
3.3	30.07	- Anzeigen von Statistiken (Zeit und Fahrzeuganzahl) auf dem Bildschirm - Fehlerbehebung, um Zusammenstösse zu verhindern (erfolgreich)
4.0	03.08	- Zurück zum Strassennetzwerk mit Kreuzungen - Berücksichtigung von anderen Fahrzeugen beim Abbiegen (mit Rechtsvortritt), noch fehlerhaft
4.1	04.08	- Breite der Kreuzung wird automatisch an die Strassenbreite angepasst - Abbremsen an der Kreuzung mit dem IDM - Viele kleine Verbesserungen beim Vortritt - Zeit kann beschleunigt werden
4.2	05.08	- Neue Methode sort_lanes, um Strassen an der Kreuzung zu sortieren - Korrekte Implementierung des Vortritts (noch nicht vollständig)
4.3	10.08	- Methode get_crossing_lanes für korrekten Rechtsvortritt - Methode, um vortrittsbedingte Blockaden aufzulösen - Verbesserungen am Dijkstra-Algorithmus
4.4	18.08	- Verbesserung der Blockaden-Methode (noch nicht erfgolgreich)
4.5	21.08	- Methode get_next_lane() wird weniger oft ausgeführt, um das Programm effizienter zu machen - Verbesserte Vortrittsfunktionen
4.6	22.08	- Strassen werden sortiert, sodass auch Einbahnstrassen berücksichtigt werden
4.7	24.08	- Korrekte Erkennung und Auflösung von Blockaden - Visualisierung von Verkehrszeichen (Vortrittszeichen) - Diverse kleine Anpassungen am Programmcode

5.0	26.08	- Wechsel auf die richtige Spur vor dem Abbiegen bei mehrspurigen Strassen - Markierung von Richtungspfeilen auf der Strasse
5.1	28.08	- Spurwechsel funktioniert jetzt korrekt - Verlangsamen des Fahrzeuges vor dem Abbiegen
5.2	05.09	- Class für Lichtsignale - Ziele: Visualisierung von roten/grünen Strichen, Testen der Lichtsignale bei einer Kreuzung
5.3	06.09	- Visualisierung der Lichtsignale - Anpassungen bei den Vortrittsregeln für Lichtsignale - Erste funktionsfähige Lichtsignalanlage (als Test)
5.4	07.09	- Änderung des Lichtsignalzyklus
6.0	16.09	- Erstellen einer Methode zum Erstellen von Kreuzungen und Strassen aus einer CSV-Datei
6.1	19.09	- Methode reload() zum Zurücksetzen der Simulation - Erstellen eines Programms, um Koordinaten von Kreuzungen umzurechnen
6.2	22.09	- Umstellung auf JSON zur Datenspeicherung
6.3	25.09	- Fahrzeuge können in der Kreuzung gespeichert werden, wenn auf der Strasse kein Platz ist (Fehlervermeidung) - Berücksichtigung von Abzweigungsrichtungen beim Dijkstra-Algorithmus
6.4	06.10	- Fahrzeuggenerator an Kreuzungen mit Wahrscheinlichkeiten
6.5	07.10	- Wechsel auf die richtige Spur zum Abbiegen direkt nach der Kreuzung - Fehlerbehebung bei der Sortierung der Strassen für die Vortrittsregeln - Berücksichtigung der übernächsten Kreuzung für den Vortritt - Möglichkeit zum Ausblenden der Visualisierung für schnellere Simulation
6.6	08.10	- Speicherung der Lichtsignalzyklen in einer JSON-Datei - Fehlerbehebungen beim Spurwechsel und den Vortrittsregeln
6.7	09.10	- Berücksichtigung des Vortritts bei Spurwechseln am Anfang der Strasse - Verschönerungen am Code (Pylint)
6.8	12.10	- Verbesserung der Fahrzeitberechnung (kein Verkehrsinfarkt mehr aufgrund besserer Routenplanung) - Letzte Verbesserungen bei den Vortrittsregeln - Verschönerungen am Code (Pylint)