Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander Döring

Simulation eines Fahrrads und Erstellung eines Karteneditors im Stadtkontext

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Döring

Simulation eines Fahrrads und Erstellung eines Karteneditors im Stadtkontext

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Dr. Florian Vogt

Eingereicht am: 6. April 2016

Alexander Döring

Thema der Arbeit
Simulation eines Fahrrads und Erstellung eines Karteneditors im Stadtkontext

Stichworte
Prozedurale Generierung, Straßengenerierung im Stadtkontext, L-System, Game Engine, Unreal Engine 4, Fahrradsimulation, EmoBike

Kurzzusammenfassung
Diese Arbeit beschäftigt sich mit der Konzeption und prototypische Umsetzung einer Fahrradsimulation für das EmoBike Projekt, in Unreal Engine 4, sowie eines Karteneditors, der Straßennetze im Stadtkontext generieren kann.

Alexander Döring

Title of the paper
Simulation of a bicycle and creation of map editor in a city context

Keywords
procedural generation, urban street generation, L-system, game engine, Unreal Engine 4, bycicle simulation, EmoBike

Abstract
This work deals with the conception and prototypical implementation of a bike simulation for the EmoBike project, in Unreal Engine 4 and a map editor, which can generate street networks in a city context.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Im EmoBike Projekt, welches eine Fahrradsimulation ist, die zum Provozieren und Aufzeichnen von Emotionen genutzt wird, wird ein Karteneditor benötigt mit dessen Hilfe effizient neue Testszenarien im Stadtkontext generiert und angepasst werden können.

Des Weiteren werden die Welten in modernen 3D-Simulationen und Spielen immer größer, was einen Generator der große Stadtgebiete erzeugen kann auch hier anwendbar macht. Da eine Erzeugung per Hand oft aus Kosten- oder Zeitgründen nicht mehr möglich ist.

1.2. Zielsetzung

Das Ziel dieser Bachelorarbeit ist erstens die Entwicklung einer Fahrradsimulation und prototypische Umsetzung in der Unreal Engine 4[1] für das EmoBike Projekt, welche in das bereits existierende System integriert werden soll und zweitens die Entwicklung eines Karteneditors zur Erstellung von Straßennetzen für die Fahrradsimulation.

Der Editor soll als Basis für die Erstellung von neuen Testszenarien im Stadtkontext für das EmoBike Projekt genutzt werden. Dazu muss der Editor in der Lage sein, verschiedenartige Straßennetze zu erzeugen und spezielle Aufgaben für unterschiedliche Szenarien erstellen zu können.

Dabei liegt der Schwerpunkt der Arbeit auf der prozeduralen Generierung von stadttypischen Straßennetzen für die Testszenarien.

1.3. Gliederung

Im zweiten Kapitel wird der Aufbau des EmoBike System erklärt, sowie zwei grundlegende Algorithmen für die Straßengenerierung im Stadtkontext.

[1] vgl. Epic Games (2016).

Im dritten Kapitel werden die Anforderungen an die Fahrradsimulation und dem Karteneditor aufgestellt.

Im vierten Kapitel wird im ersten Teil das Konzept für die Integration und Umsetzung der Fahrradsimulation erläutert. Im zweiten Teil wird das Konzept der Straßengenerierung und Grundstückaufteilung dargelegt.

Im fünften Kapitel wird die prototypische Umsetzung der beiden Softwarekomponenten in Unreal Engine 4 behandelt.

Im sechsten Kapitel wird der Prototyp auf die Erfüllung der Anforderung geprüft. Des Weiteren wird die Performance des Straßengenerators getestet.

Im letzten Kapitel wird die Arbeit zusammengefasst und ein Ausblick auf mögliche Erweiterungen dar

gestellt.

2. Grundlagen

2.1. EmoBike System

2.1.1. Übersicht

Das EmoBike Projekt ist ein Versuchsaufbau zur Messung von Emotionen, die gezielt in einer Fahrradsimulation provoziert werden. Die Simulation wird visuell durch ein simuliertes Fahrrad im EmoBike Game umgesetzt, welches ein Proband physisch durch ein Fahrradergometer steuert(siehe Abbildung 2.1). Der Haltegriff des Ergometers wurde durch einen beweglichen Fahrradlenker ersetzt, um eine realistischere Simulation zu ermöglichen und Testszenarien interaktiver zu gestalten. Ein bereits vorhandenes Testszenario ist ein kurzer Slalomparkur in dem ein Proband Münzen einsammeln muss. Bei dem Testszenario kann zum Beispiel Freude beim Probanden gemessen werden, wenn er eine Münze einsammelt oder den Parkur erfolgreich beendet.

Das Projekt ist in unabhängige Module aufgeteilt, die über einen zentralen Message Broker[1] kommunizieren.

Kontrollmodul

Das Kontrollmodul besteht aus mehreren Softwarekomponenten, die die Sensordaten vom Ergometer verarbeiten und den Tretwiderstand des Ergometers an die simulierte Situation anpassen. Die verarbeiteten Sensordaten werden über den Message Broker an alle registrierten Module verteilt.

Gamemodule

Das EmoBike Game simuliert anhand der vom Kontrollmodul gelieferten Daten die Bewegung des Fahrrads in der virtuellen Welt und liefert dem Kontrollmodul Information über Steigung und eventuelle Kollisionen.

[1] Nachrichtenverwalter.

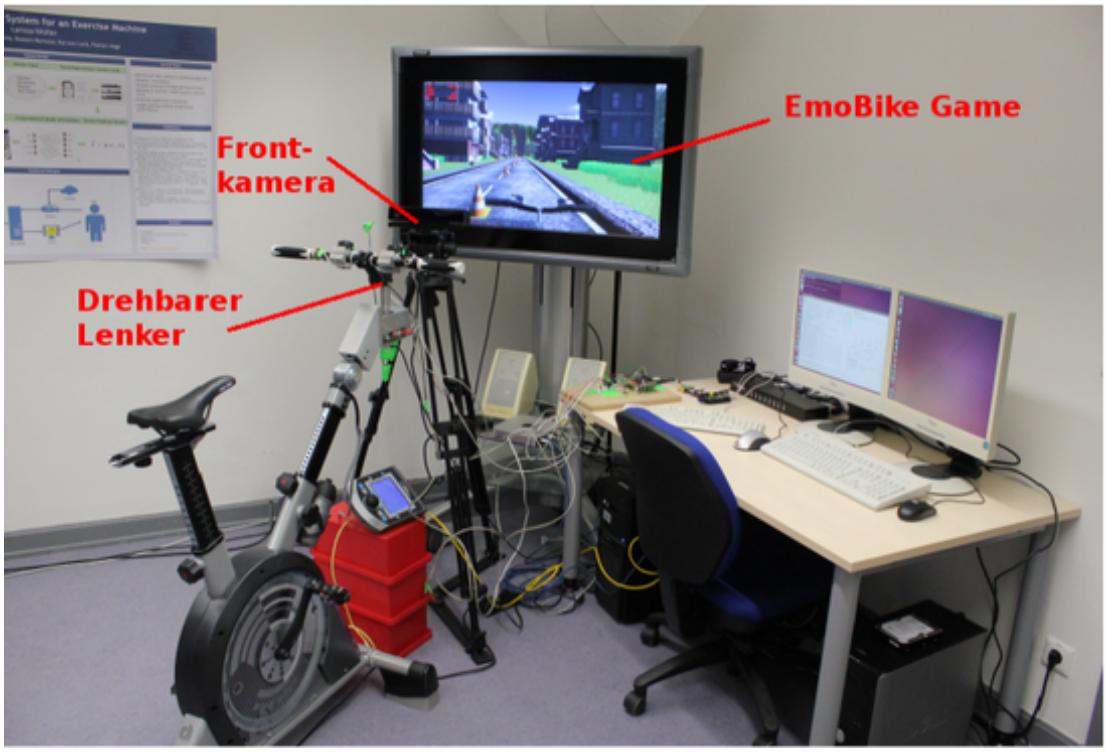


Abbildung 2.1.: EmoBike Aufbau aus Hornschuh (2015)

Zusätzlich ist es die Aufgabe des EmoBike Games Ereignisse auszulösen, die beim Probanden bestimmte emotionale Reaktionen hervorzurufen. Die Game Ergeignisse werden über den Message Broker dem System mitgeteilt.

2.1.2. Kommunikation des Systems

Das EmoBike Projekt nutzt ActiveMQ[2] als Message Broker für die Kommunikation zwischen den Hauptkomponenten.

Der Message Broker hat mehrere Nachrichtenkanäle, die in ActiveMQ Topic genannt werden. Jede empfangene Nachricht zu einem Topic wird an alle Klienten, die sich für das Topic als Empfänger registriert haben, weitergeleitet.

Jeder Klient kann sich als Empfänger bei einem Topic registrieren oder als Publisher eine Nachricht an ein Topic senden.

[2] Apache Software Foundation (a).

EmoBike Nachrichtenprotokoll

Damit alle Komponenten des EmoBike Systems die Nachrichten verstehen können, ist ein Protokoll in JSON definiert.

Jede Nachricht besteht aus einem JSON Objekt mit mehreren Key-Value-Paaren. In allen Nachrichten sind die Key-Value-Paare *sender*, *timestamp* und *type* enthalten. Die weiteren Key-Value-Paare sind je nach *type* der Nachricht unterschiedlich.

```
{
  "sender": "id",
  "timestamp": "32432324",
  "type":   "direction",
  "datatype": "KEYVALUE",
  "data":  "0.44"
}
```

Nachricht	Funktion	Sender	Type	Datatype	Data	Format
Richtung	Richtungsausschlag	raspberry	direction	KEYVALUE	-1.00 bis 1.00	%1.2f
Referenz	Referenz Signal	raspberry	reference	NONE	-	-
Geschwindigkeit	Geschw. in Meter pro Sekunde	bikemodule	speed	KEYVALUE	-28.0 bis 28.0	%2.1f
Steigung	Steigung in Grad	game	slope	KEYVALUE	0.00 bis 360.00	%3.2f
Gangschaltung	Gänge schalten	raspberry 2	gear	KEYVALUE	1 bis 28	%d
Bremse	Geschw. und Widerstand anpassen	raspberry 2	break	KEYVALUE	0 bis x	%d
Reset	Level neu starten	beliebig	reset	NONE	-	-
Load	Level laden	controlcenter	level	KEYVALUE	LevelName	String
Level geladen	Level wurde geladen	game	levelLoaded	MAP		
Level gestartet	Startlinie wurde überschritten	game	levelStarted	MAP		
Level gestoppt	Ziellinie wurde überschritten	game	levelFinished	MAP		
Event in Game	Event Trigger wurde ausgelöst	game	TriggerName	Event abhängig		

Abbildung 2.2.: EmoBike Nachrichtenprotokollformat

2.1.3. Physiksimulation in der Unreal Engine 4

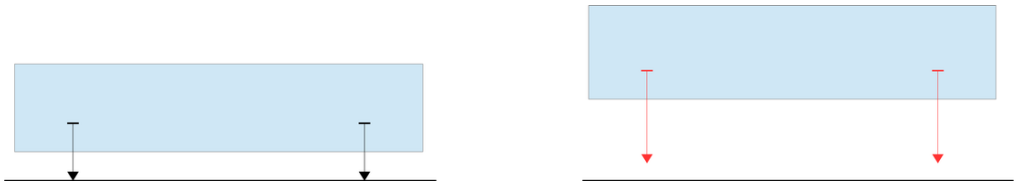
Unreal Engine 4[3] nutzt Nvidia PhysX[4] für alle physikalische Simulationen. Die Physiksimulation läuft in einer parallelen Umgebung, die alle Objekt mit physikalischer Interaktion abbildet und vor jedem Rendering eines Frames die Simulation aktualisiert. In jedem Simulationsschritt werden die Kräfte berechnet, die auf jedes bewegliche Objekt wirken. Daraus wird für jeden Körper die Beschleunigung berechnet und die neue Geschwindigkeit des Objekts ermittelt. Im nächsten Schritt wird die neue Postion des Objekts berechnet und auf Kollisionen geprüft. Die dadurch entstandene Position wird anschließend an das Unreal Objekt übergeben.

[3] vgl. Epic Games (Physics Simulation).
[4] vgl. NVIDIA (PhysX SDK 3.3.1).

Für die Simulation von Fahrzeugen hat PhysX ein zusätzliches Modul(PhysX Vehicle[5]) das auch in der Unreal Engine nutzbar ist. Ein Fahrzeug besteht aus einem starren Körper, auf den die normale physikalische Simulation angewandt wird. Um ein Fahrzeug zu erstellen, muss der starre Körper, die Radaufhängungen und die Variablen der Räder und Federungen definiert werden. In der Simulation muss dann der Lenkeinschlag und Drehmoment der einzelnen Räder gesetzt werden. Die restliche Berechnung ist nicht weiter beeinflussbar.

Zur Berechnung der Kräfte, die von den Rädern und Federungen an der jeweiligen Aufhängung auf den Körper ausgeübt werden, wird eine vereinfachte Formel angewandt. Dafür müssen bei der Erstellung des Fahrzeugs für jede Feder maximal Länge(ohne Last), minimal Länge(maximale Komprimierung) und die Federkonstante bestimmt werden. Sowie für jedes Rad, der Radius, maximaler Lenkeinschlag und Reifentyp festgelegt werden. Des Weiteren hat jedes Rad dynamische Variablen(Rotationsgeschwindigkeit, Drehmoment, Lenkeinschlag), die in jedem Simulationsschritt angepasst werden.

Im ersten Schritt wird die Komprimierung jeder Feder berechnet. Dafür wird ein Raycast von der Radaufhängung bis zur Unterseite des Rades(max. Federlänge + Radius des Rades) ausgeführt. Wenn der Raycast einen Untergrund trifft(siehe Abbildung 2.3a), erhält man Daten über den Untergrund(Kontaktpunkt und Material) und den Distanzquotient. Ansonsten ist das Rad in der Luft(siehe Abbildung 2.3b) und es werden keine weiteren Berechnungen für das Rad ausgeführt.



(a) Beide Raytraces treffen den Boden und die Federn sind halb komprimiert, dementsprechend haben die Räder guten Bodenkontakt.

(b) Beide Raytraces enden in der Luft, deshalb kann keine Kraft von den Rädern erzeugt werden und die Berechnung wird beendet.

Abbildung 2.3.: Raytrace für ein Fahrzeug mit zwei Rädern

Mit dem Distanzquotient wird die Komprimierung der Feder bestimmt und daraus wird die Federkraft mit Hilfe der Federkonstanten berechnet.

[5] vgl. NVIDIA (Vehicles).

$$Federkraft = Federkonstante * Komprimierung.$$

Die Federkraft wirkt über die Radaufhängung auf den Körper und wird genutzt, um zu berechnen, wie gut der Kontakt des Rades mit dem Untergrund ist. Als Nächstes wird der Rollwiederstand zwischen Reifen und Untergrund aus einer Reifentyp/Material Tabelle ermittelt.

Im letzten Schritt wird die Kraft berechnet, die das Rad erzeugt. Die so berechneten Kräfte werden an der jeweiligen Radaufhängung auf den Fahrzeugkörper ausgeübt

2.2. Karteneditor

2.2.1. Erweitertes L-System zur Straßengenerierung

Ein L-System[6] ist ein Textersetzungssystem, bei dem mit Hilfe von Produktionsregeln die variablen Symbole in einem String ersetzt werden. Der Hauptunterschied zu anderen Ersetzungssystemen ist das in jedem Durchlauf alle Symbole, auf die eine Regel anwendbar ist, ersetzt werden.

L-Systeme werden auch zur Darstellung von organischen Wachstumsprozessen genutzt. Um eine Zeichnung zu bekommen, werden die Symbole des finalen Strings als Zeichnungsoperationen interpretiert, die von der Position und Richtung der letzten Zeichnungsoperation fortgesetzt werden.

In einem einfachen L-System gibt es meist zwei Symbole, eins zum Zeichnen eines Striches und eins um die Richtung zu ändern. Da es in dieser Umsetzung immer nur die aktuelle Postion und Richtung gibt, wurde das System zum Zeichnen von komplexeren Zweigstrukturen um einen LIFO[7]-Speicher erweitert. Auf dem Speicher wird die aktuelle Postion/Richtung mit dem terminal Symbol [gespeichert und die letzte gespeicherte Position/Richtung mit dem terminal Symbol] wiederhergestellt.

Parametrische L-Systeme[8] sind eine Erweiterung der Symbole um Parameter. Die erweiterten Symbole werden Module genannt. Parameter eines Moduls können von Produktionsregeln als Kondition für die Anwendbarkeit einer Regel genutzt und verändert werden.

[6] vgl. Lindenmayer (1968).
[7] Last In, First Out.
[8] vgl. Prusinkiewicz und Lindenmayer (2012), S. 40f.

$$Id : linkerKontext < Vorgänger > rechterKontext : Kondition \rightarrow Nachfolger$$

Abbildung 2.4.: Syntax von Produktionsregeln

Es können Parameter vom linken/rechten Kontext sowie aus dem Vorgänger in der Kondition und dem Nachfolger verwendet werden. Eine Regel besteht mindestens aus Id, Vorgänger und Nachfolger.

Ein einfaches Beispiel eines L-System zur Erstellung eines Blockmusters bei dem Modul $s(a)$ als Strich mit der Richtung a interpretiert wird.

Beispiel L-System

```
Axiom: r(0,0)

p1: r(x, a) : x == 0 → s(a)[r(1,a+90)][r(1,a-90)]r(0,a)
p2: r(x,a) : x > 0 → r(x-1,a)

Ersten drei Ersetzungen:
0: r(0,0)
1: s(0)[r(1,90)][r(1,-90)]r(0,0)
2: s(0)[r(0,90)][r(0,-90)]s(0)[r(1,90)][r(1,-90)]r(0,0)
3: s(0)[s(90)[r(1,180)][r(1,0)]][s(-90)[r(1,0)][r(1,180)]]s(0)[r(0,90)][r(0,-90)]s(0)[r(1,90)][r(1,-90)]r(0,0)
```

Bereits im sechsten Schritt kommt es zur Zeichnung übereinanderliegender Linien(siehe Abbildung 2.5), um das zu vermeiden, müsste das L-System um weitere Parameter und kontextsensitive Produktionsregel erweitert werden.

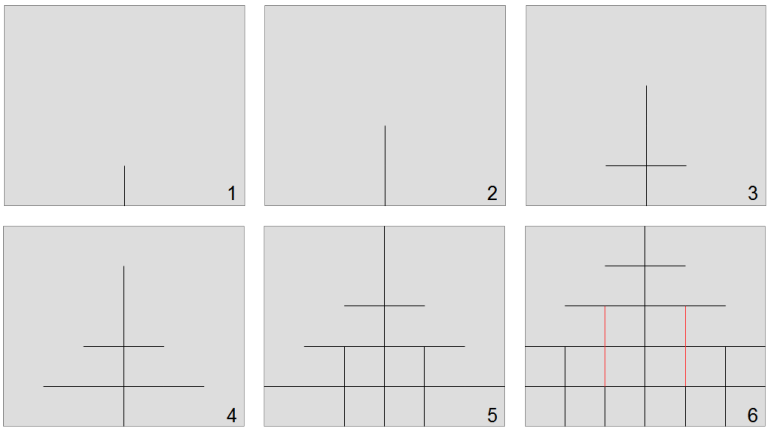


Abbildung 2.5.: Darstellung des Beispiel L-Systems für sechs Schritte (übereinanderliegende Linien sind rot dargestellt)

Um L-Systeme zur Straßengenerierung zu nutzen, kann jedes Symbole, das zuvor einen Strich erzeugt hat, als Straßensegment interpretiert werden und die Übergänge zwischen den Strichen als Kreuzungen.

Da zur Straßengenerierung mit verschiedenen Straßenmustern zu viele Parameter und spezielle Produktions Regeln benötigt gewesen wären, haben sich Parish und Müller dazu entschieden ein erweitertes L-System[9] zu erstellen, in dem die meisten Parameter von externen Funktionen ermittelt werden, die in Produktionsregeln aufgerufen werden.

Dadurch werden vielen kontextsensitive Produktionsregel vermieden, die alle bei Änderung an einem Straßenmuster einzeln angepasst werden müssten. Im Fall von externen Funktionen muss nur die betroffene Funktion angepasst werden und nichts am L-System geändert werden.

Das erweiterte L-System erstellt in der ersten Produktionsregel nur einen idealen Nachfolger(siehe Abbildung 2.6), bei dem alle Parameter unbestimmt sind.

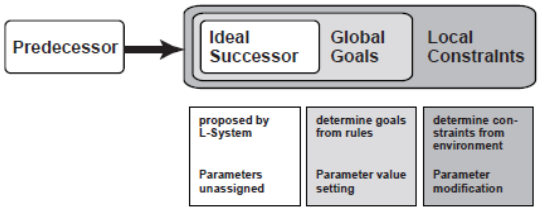


Abbildung 2.6.: Funktionen, die auf den idealen Nachfolger angewandt werden[10]

In der nächsten Produktionsregel wird die Funktion $GlobalGoals$ ausgeführt. In der Funktion wird mit Hilfe der Bevölkerungsdichte und des gewählten Straßenmusters eine Richtung und Länge der neuen Straße bestimmt. Dieser Straßenvorschlag wird in den Parametern des idealen Nachfolgers gespeichert.

Im letzten Schritt wird die Funktion $LocalConstraints$ ausgeführt. Diese prüft, ob die durch $GlobalGoals$ vorgeschlagene Straße im gegebenen Terrain zulässig ist und sich nicht mit anderen Straßen schneidet. Wenn der Vorschlag unzulässig ist, versucht die Funktion den Vorschlag anzupassen, sollte das nicht möglich sein, wird der Vorschlag abgelehnt. Vom Ende einer erfolgreich erstellten Straße werden neue ideale Nachfolger gebildet.

Regeln des erweiterten L-Systems(für bessere Lesbarkeit leicht angepasst)[11]

```
Abk.: rv = RuleVariable, sv = StraßeVariable, x = counter
```

[9] Parish und Müller (2001)
[10] Abbildung 3 aus Parish und Müller (2001)

```
axiom: road(0, rv), query(sv, UNASSIGNED)

R1: road(x, rv) : x < 0 → ε
R2: road(x, rv) > query(sv, state) : state==SUCCEED,
        (nx[3], nsv[3], nrv[3]) = globalGoals(sv,rv)
      → segment(roadVar angle, roadVar.length)
          branch(nx[1], nrv[1], nsv[1])
          branch(nx[2], nrv[2], nsv[2])
          road(nx[0], nrv[0]) query(nsv[0], UNASSIGNED)
R3: road(x, rv) > query(sv, state) : state==FAILED → ε
R4: branch(x, sv, rv) : x > 0 → branch(x−1, sv, rv)
R5: branch(x, sv, rv) : x == 0 → [road(x, rv),query(sv, UNASSIGNED)]
R6: branch(x, sv, rv) : x < 0 → ε
R7: road(x, rv) < query(sv, state) : x < 0 → ε
R8: query(sv, state) : state== UNASSIGNED,
        (nsv,ns)= localConstraints(sv) → query(nsv, ns)
R9: query(sv, state) : state != UNASSIGNED → ε
```

Die Regeln können durch Vereinfachung in zwei Regeln zusammengefasst werden.[12]

Da $road$ und $query$ immer als Paar vorkommen und unter den gleichen Bedingungen ersetzt werden, können die beiden Module zu einem Modul $roadq(x, rv, sv, state)$ vereint werden. Dadurch werden $R1$ und $R7$ zur gleichen Regel und die Regel R_9 wird nicht mehr benötigt, weil $roadq$ entweder in Regel R_2 ersetzt oder in Regel R_3 gelöscht wird und kein $query$ mehr übrig bleibt, der in R_9 gelöscht werden muss.

Gleichermaßen kann $branch$ durch $roadq$ ersetzt werden, da es nur notwendig war, weil $query$ sonst von Regel R_8 zu früh ersetzt worden wäre. Dadurch wird Regel R_5 nicht mehr benötigt, da $branch$ nicht mehr durch $roadq$ ersetzt werden muss und Regel R_6 ist identisch zu Regel R_7.

Regeln nach der ersten Anpassung

```
R2: roadq(x, rv, sv, state) : state==SUCCEED,
        (nx[3], nsv[3], nrv[3]) = globalGoals(sv,rv)
          → segment(roadVar.angle, roadVar.length)
            [roadq(nx[1], nrv[1], nsv[1], UNASSIGNED)]
            [roadq(nx[2], nrv[2], nsv[2], UNASSIGNED)]
             roadq(nx[0], nrv[0], nsv[0], UNASSIGNED)
```

[11] vgl. Parish und Müller (2001).
[12] vgl. Barrett (2007–2009).

```
R3: roadq(x, rv, sv, state) : state==FAILED → ε
R4: roadq(x, rv, sv, state) : x > 0 → roadq(x-1, rv, sv, state)
R7: roadq(x, rv, sv, state) : x < 0 → ε
R8: roadq(x, rv, sv,state) :state== UNASSIGNED,
        (nsv,s)= localConstraints(sv) → roadq(x,rv, nsv,s)
```

In einem normale L-System sind keine Regeln möglich, die bedingte Teilersetzungen haben. Für eine weitere Vereinfachung und bessere Verständlichkeit wird das L-System hier um diese Möglichkeit erweitert.

Die Regeln R_2 und R_3 werden nur angewandt, wenn $state$ in Regel R_8 durch die Funktion $LocalConstraints$ gesetzt wurde. Im Fall von $SUCCEED$ wird $roadq$ durch ein Straßensegment und neue Straßenvorschläge ersetzt und im Fall von $FAILED$ durch ε ersetzt. Also können die Regeln R_2 und R_3 in Regel R_8 integriert werden, so dass im $SUCCEED$ Fall $roadq$ durch den Nachfolger aus Regel R_2 ersetzt wird und im $FAILED$ Fall durch ε ersetzt wird. Da $state$ nur noch in Regel R_8 benötigt wird, kann der Parameter aus $roadq$ entfernt werden.

Regel R_7 löscht $roadq$, wenn x einen negative Wert hat. Dieser Fall tritt nur ein, wenn die Funktion $GlobalGoals$ x einen negativen Wert zugewiesen hat. Deswegen kann Regel $R8$ so verändert werden, das $roadq$ nur erzeugt wird, wenn x positive ist.

Regeln nach der zweiten Anpassung

```
axiom: roadq(0, rv, sv)

R4: roadq(x, rv, sv) : x > 0 → roadq(x-1, rv, sv)
R8: roadq(x, rv, sv) : x==0, (nsv,state)= localConstraints(sv);
        if (state==SUCCEED) :
           (nx[3], nsv[3], nrv[3]) = globalGoals(nsv,rv)
               → segment(sv.angle, sv.length)
                 if(nx[1]>=0) [roadq(nx[1], nrv[1], nsv[1])]
                 if(nx[2]>=0) [roadq(nx[2], nrv[2], nsv[2])]
                 if(nx[0]>=0) roadq(nx[0], nrv[0], nsv[0])
```

Das nun entstandene L-System hat eine Regel zur Verwaltung, wann die einzelnen Straßenvorschläge ersetzt werden, sowie eine Regel, die für jeden Straßenvorschlag mit $LocalConstraints$ prüft, ob dieser gültig ist. Ist das der Fall, wird die Straße erstellt und mit der $GlobalGoals$ Funktion weiter Vorschläge erzeugt.

Wenn man nun den Startpunkt jedes Straßenvorschlags in $roadq$ speichert, kann man die Speicherung der Postion von jeder Abzweigung [und] entfernen und das Ganze in eine Programmiersprache umsetzen.

Pseudocode des L-Systems

```
Liste RoadQ
Liste Segment

RoadQ.Add(roadq(0, rv, sv))

until RoadQ.Empty()
   for each roadq in RoadQ
      if (roadq.x > 0)
         roadq.x—
      else
         RoadQ.Delete(roadq)
         (nsv, state) = LocalConstraints(roadq.sv)
         if(state == SUCCEED)
            Segment.Add ( segment(nsv) )
            (nx[3], nsv[3], nrv[3]) = globalGoals(nsv,roadq.rv)
            if (nx[1] >= 0) RoadQ.Add( roadq(nx[1], nrv[1], nsv[1]) )
            if (nx[2] >= 0) RoadQ.Add( roadq(nx[2], nrv[2], nsv[2]) )
            if (nx[0] >= 0) RoadQ.Add( roadq(nx[0], nrv[0], nsv[0]) )
```

Da ein Großteil der Schleifendurchläufe damit verbracht wird x in der *For-Schleife* zu testen und zu verringern, kann das Ganze weiter optimiert werden, indem man die List RoadQ durch ein Priority Queue[13] ersetzt, bei dem die Straßenvorschläge nach dem kleinsten x sortiert werden. Da im Priority Queue der x Wert der Straßenvorschläge nicht mehr verändert wird, muss der Wert des aktuellen Vorschlages zu dem neuen Wert des neuen Vorschlages hinzugefügt werden und die *For-Schleife* kann entfernt werden, da immer der richtige Straßenvorschlag ausgewählt wird.

Angepaster Pseudocode des L-Systems

```
PriorityQueue RoadQ
Liste Segment

RoadQ.Add(roadq(0, rv, sv))
```

[13] Prioritätswarteschlange.

```
until RoadQ.Empty()
   roadq = RoadQ.Pop()
   (nsv, state) = LocalConstraints(roadq.sv)
   if(state == SUCCEED)
      Segment.Add(segment(nsv))
      (nx[3], nsv[3], nrv[3]) = globalGoals(nsv,roadq.rv)
      if (nx[1] >= 0) RoadQ.Add(roadq(roadq.x+nx[1], nrv[1], nsv[1]))
      if (nx[2] >= 0) RoadQ.Add(roadq(roadq.x+nx[2], nrv[2], nsv[2]))
      if (nx[0] >= 0) RoadQ.Add(roadq(roadq.x+nx[0], nrv[0], nsv[0]))
```

2.2.2. Kleinster Kreis im planaren Graphen

Für die Straßengenerierung muss entschieden werden, ob sich ein Straßenblock gebildet hat. Dies kann durch die Ermittlung des kleinsten Kreises im planaren Graphen bestimmt werden.

Der kleinste Kreis in einem planaren Graphen ist ein Kreis(Weg mit gleichem Start- und Endknoten), bei dem kein anderer Kreis existiert, der sich eine Kante mit dem Kreis teilt und im Inneren des Kreises liegt.

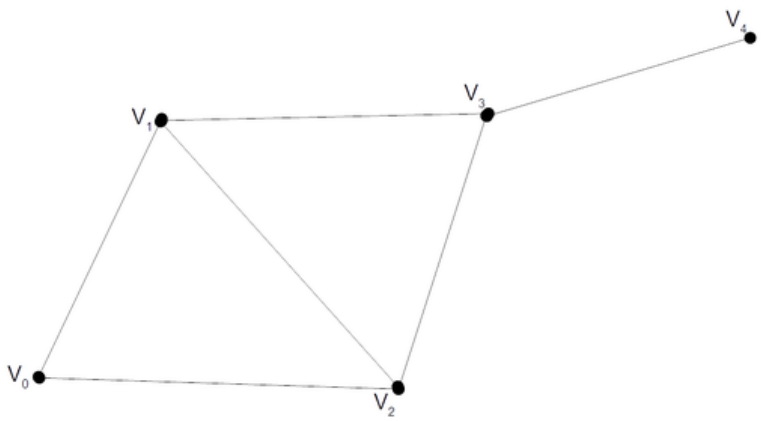


Abbildung 2.7.: Planarer Graph mit drei Kreisen und einem Filament

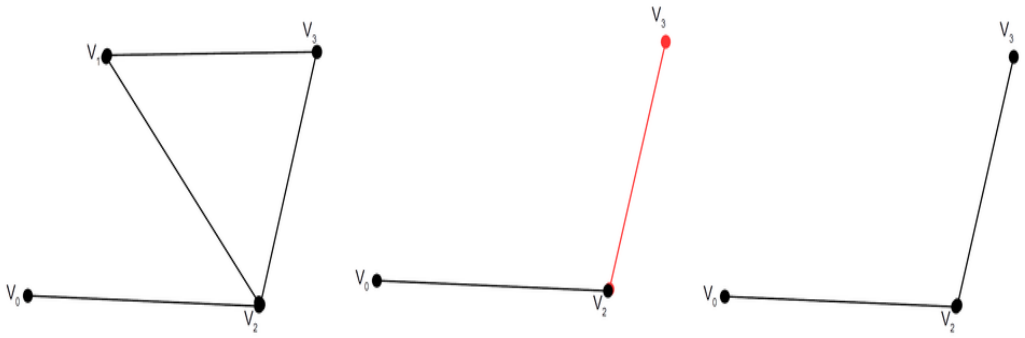
Der Graph in Abbildung 2.7 enthält drei Kreise $K_1\{V_0, V_1, V_3, V_2\}$, $K_2\{V_0, V_1, V_2\}$ und $K_3\{V_1, V_3, V_2\}$. Der Kreis K_1 ist kein kleinster Kreis, da die zwei Kreise K_2 und K_3 jeweils eine Kante mit K_3 teilen und im Inneren von K_1 liegen. K_2 und K_3 sind kleinste Kreise, da sie keine weiteren Kreise enthalten.

Wege wie der Weg $\{V_3, V_4\}$, die nicht Teil eines kleinsten Kreises sind, werden als Filament bezeichnet.

Das Finden eines kleinsten Kreises in einem planaren Graphen kann mit Hilfe des von David Eberly entwickelten Algorithmus[14] umgesetzt werden.

Der Algorithmus teilt den Graphen in drei Listen auf. Eine Liste mit allen kleinsten Kreisen, eine Liste mit allen Filamenten und eine Liste mit freistehenden Knoten. Um diese zu finden werden im ersten Schritt alle Knoten in einer Liste gespeichert und nach Ihren x/y Koordinate sortiert, so dass der Knoten mit dem kleinsten x als erstes in der Liste steht. Wenn zwei Knoten den gleichen x Wert haben, werden sie nach dem kleinsten y sortiert.

In der Hauptschleife des Algorithmus wird immer der erste Knoten aus der Liste genommen und von dort aus geprüft, ob der Knoten teil eines Kreises ist. Wenn der Knoten keine Kanten hat, wird er der Liste der freien Knoten hinzugefügt und aus dem Graphen und der sortierten Liste entfernt.



(a) Filament endet bei V_2, da V_2 ein Zweigknoten ist.

(b) Filament endet bei V_2, da die nächste Kanten von V_2 markiert ist.

(c) Filament endet bei V_3, da V_3 keine weiteren Kanten hat.

Abbildung 2.8.: Die drei Fälle von Filamenten

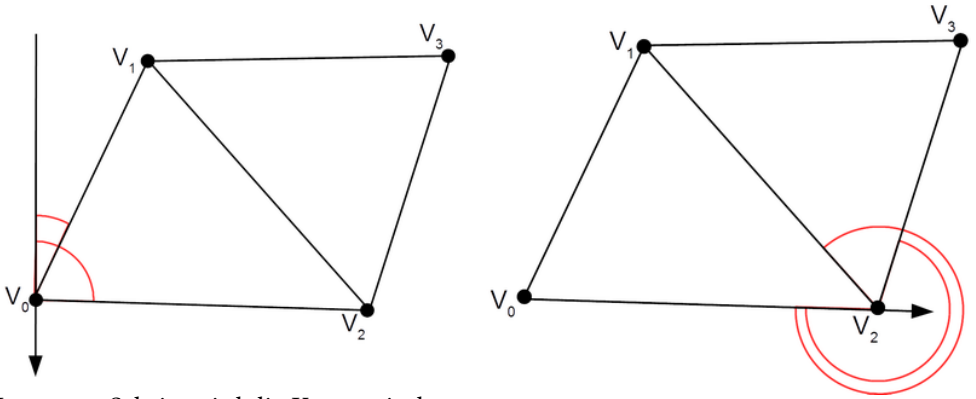
Wenn der Knoten eine Kante hat und diese nicht markiert ist, ist der Knoten Teil eines Filaments. Es wird solange dem Weg über die Kanten gefolgt, bis ein Knoten keine weiteren Kanten mehr hat(2.8a), die nächste Kante eines Knoten markiert ist(2.8b) oder ein Zweigknoten[15] erreicht wird(2.8c).

Der so gefundene Weg wird in der Liste der Filamente gespeichert, des Weiteren werden alle Knoten und Kanten des Weges, mit Ausnahme des letzten Knotens, aus dem Graphen und der sortierten Liste entfernt. Wenn der letzte Knoten nur eine Kante hat, wird dieser auch entfernt. Wenn der Knoten eine Kante hat und diese markiert ist, muss nur der Knoten und die Kante aus dem Graphen entfernt werden, da eine markierte Kante Teil eines bereits entfernten

[14] vgl. Eberly (2014).
[15] Knoten der drei oder mehr Kanten hat.

Kreises ist.

Ansonsten wird geprüft, ob der Knoten Teil eines Kreises ist. Dazu wird von dem ersten Knoten aus die Kante mit dem größten Winkel im Uhrzeigersinn zu einer senkrechten Linie ausgewählt(siehe Abbildung 2.9a). Von dem nächste Knoten wird immer die nächste Kante gesucht, die den größten Winkel gegen den Uhrzeigersinn zu der letzten Kante hat(siehe Abbildung 2.9b). Dieses wird wiederholt bis entweder der erste Knoten oder ein Endknoten erreicht wird.



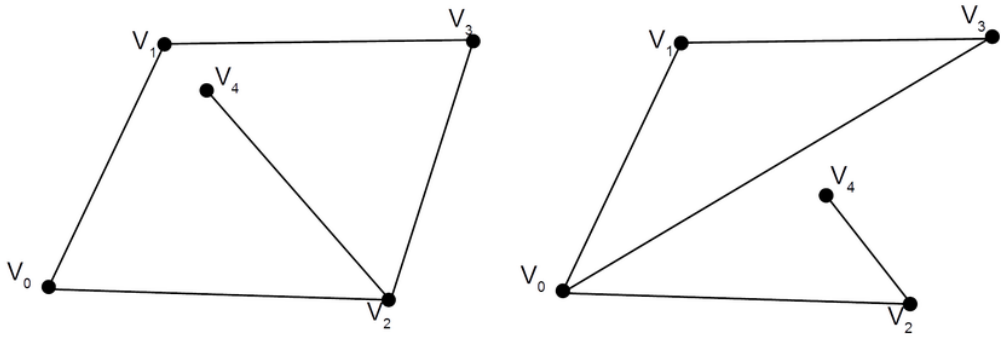
(a) Im ersten Schritt wird die Kante mit dem größten Winkel im Uhrzeigersinn gesucht. In diesem Fall (V_0,V_2).

(b) Im ersten Schritt wird die Kante mit dem größten Winkel gegen den Uhrzeigersinn gesucht. In diesem Fall (V_2,V_1).

Abbildung 2.9.: Graphen zur Darstellung der Kantenfindung

Im ersten Fall wurde ein kleinster Kreis gefunden und wird in der Liste der kleinsten Kreise gespeichert. Außerdem wird die erste Kante des Kreises aus dem Graphen entfernt und alle anderen Kanten werden markiert, damit sie nicht mit Filamenten verwechselt werden.

Im zweiten Fall wurde ein Filament gefunden und es wird mit Hilfe von Backtracking des bereits gefundenen Weges versucht einen Zweigknoten zu finden. Wenn ein Zweigkonten gefunden wurde(siehe Abbildung 2.10a), wird der Teil des gefundenen Weges vom Start des Backtracking bis zum Zweigknoten aus dem Graphen entfernt und der Liste der Filamente hinzugefügt. Die Suche des kleinsten Kreises wird vom Zweigknoten fortgesetzt. Sollte beim Backtracking der erste Knoten im Weg erreicht werden(2.10a), wird der ganze Weg aus dem Graphen entfernt und der Liste der Filamente als Filament hinzugefügt. Die Suche wird in der Hauptschleife mit dem selben Knoten neu gestartet.



(a) In V_4 wird das Backtracking gestartet und endet am Zweigknoten V_2, $\{V_2, V_4\}$ wird als Filament entfernt und die Suche des Kreis wird von V_2 vorgesetzt.

(b) In V_4 wird das Backtracking gestartet und endet am Anfangsknoten V_0, demzufolge wird der gesamte Weg als Filament entfernt und die Suche von V_0 aus neu gestartet.

Abbildung 2.10.: Beispiele zum Backtracking

3. Anforderungsanalyse

3.1. Simulation eines Fahrrades

Das simulierte Fahrrad muss sich realistisch verhalten und verzögerungsarm die Steuereingaben des Kontrollmoduls umsetzen. Dazu muss der EmoBike Message Broker in die Unreal Engine 4 eingebunden werden.

Außerdem muss das simulierte Fahrrad die Steigung und andere Simulationsereignisse wie das Erreichen des Zieles an das Kontrollmodul melden.

3.2. Karteneditor im Stadtkontext

Der Karteneditor muss es ermöglichen das Benutzer nach einer kurzen Einführung in die Bedienung des Editors in der Lage sind größere Straßennetze im Stadtkontext zu erstellen. Dazu wird ein Straßengenerator benötigt, der leicht konfigurierbar ist und verschiedene Straßennetze auf einem gegeben Terrain erzeugen kann. Des Weiteren muss der Generator die dabei entstandenen Gebiete zwischen den Straßen in Grundstücke aufteilen und Gebäude darauf platzieren können.

Sowie ein manueller Editor der es ermöglicht die generierten Straßen zu verändern und neue Straßen per Hand hinzuzufügen, so dass sich alle Teile der Stadt an die Änderung anpassen und eine weitere Generierung von Straßen möglich bleibt, ohne das der Benutzer das unterliegende System kennen muss.

4. Konzept

4.1. Fahrrad

Die Fahrradsimulation setzt die Fortbewegung des Fahrrades in der Spielwelt um. Dazu werden die von der Kontrollkomponente berechnete Geschwindigkeit und der Lenkeinschlag auf ein physikalisch simuliertes Fahrrad angewandt.

Um den Datenaustausch zwischen Kontrollkomponente und Simulation zu ermöglichen, muss die Programmbibliothek[1] des Message Brokers in die Unreal Engine 4 eingefügt werden. Da Unreal Engine 4 JSON native unterstützt, kann das EmoBike Nachrichtenprotokoll ohne größeren Aufwand implementiert werden.

Zur Umsetzung des physikalisch simulierten Fahrrades wird ein Modell eines Dreirades verwendet. Dies ist notwendig, da ein zweirädriges Fahrrad bei niedrigen Geschwindigkeiten oder Unebenheiten keine eigene Balance hat und es zu aufwändig wäre die Balancierung zu berechnen.

Da das physikalische Fahrzeugmodell von Unreal Engine 4 parallel zum visuellen Modell existiert, aber nicht dargestellt wird, kann das ohne großen Aufwand umgesetzt werden, indem statt einem Hinterrades zwei leicht versetzte Räder definiert werden(siehe Abbildung 4.1). Während das visuelle Fahrradmodell nur ein Hinterrad hat.

Für die physikalische Simulation des Fahrrades muss die vom Kontrollmodul gegebene Geschwindigkeit in das notwendige Drehmoment zum Beschleunigen des Fahrrads umgerechnet werden. Dies erfolgt indem die notwendige Beschleunigung ($a = V_a - V_g$) aus der Differenz zwischen aktueller und gegebener Geschwindigkeit bestimmt wird. Mit Hilfe der Beschleunigung wird die Kraft ($F = a * m$) berechnet, die notwendige ist, um die Masse des Fahrrades auf die gegebene Geschwindigkeit zu bringen. Die Kraft wird in das Drehmoment ($\tau = F * r$) am Hinterrad umgerechnet.

Der Lenkeinschlag kann ohne weitere Berechnungen auf das Vorderrad angewandt werden.

[1] vgl. Apache Software Foundation (b).

Abbildung 4.1.: Fahrrad mit den zwei zusätzlichen Hinterrädern(in rot) für die Physiksimulation

Das simulierte Fahrrad muss der Kontrollkomponente Daten über die Steigung liefern. Dazu kann der Richtungsvektor des Fahrrades genommen und in die Steigung umgerechnet werden. Der Richtungsvektor wird von Unreal Engine 4 zur Verfügung gestellt.

4.2. Event- und Aufgabensystem

Für die Testszenarien muss das Fahrrad durch Events und Aufgaben mit der Simulation interagieren können.

Da Unreal Engine 4 durch das eigene Blueprint Visual Scripting System[2] schon ein komplexes dennoch einfach zu nutzendes System zum Erstellen von triggerbaren Events und Aufgaben hat (siehe Abbildung 4.2), muss nur ein Interface zum EmoBike Message Broker erstellt werden.

[2] vgl. Epic Games (Blueprints).

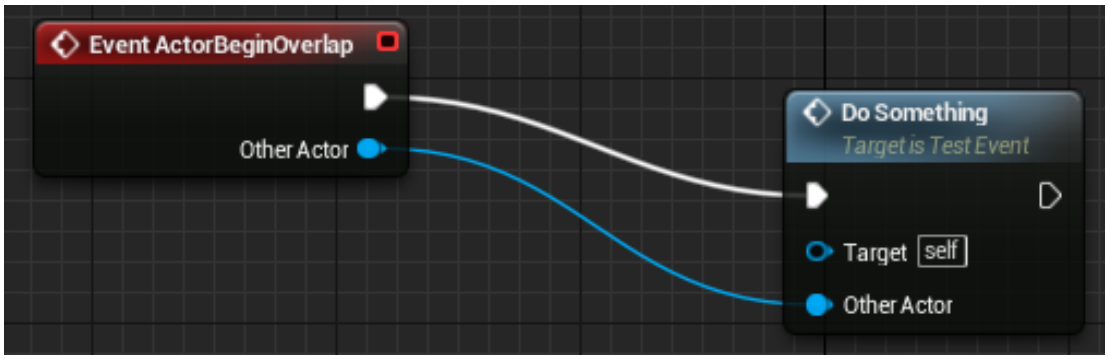


Abbildung 4.2.: Einfacher Blueprint Event, der ausgelöst wird, wenn das Objekt mit einem anderen kollidiert und die Funktion *DoSomething* ausführt.

Da die Grundlage dafür schon für die Steuerung des Fahrrades erstellt wurde, muss nur noch eine Funktion erstellt werden, die in jedem Blueprint aufrufbar ist und eine Nachricht mit beliebiger Anzahl an Datenpaaren senden kann.

4.3. Generierung

4.3.1. Straßendatenstruktur

Die Datenstruktur der Straße ist an die Winged Edge Datenstruktur[3] angelehnt, da eine der intensivsten Operationen das Finden von Straßenblöcken(Faces) ist, welche in der Winged Edge Datenstruktur über eine einfache Iteration der Kanten gefunden werden können. Des Weiteren erlaubt die Datenstruktur nur planare Graphen, was auch in einem Straßennetz der Fall ist, da Straßen, die sich schneiden würden, immer eine Kreuzung bilden.

Jede Straße speichert die Start- und Endkreuzung, den linken und rechten Block und je die linke und rechte Straße an jeder Kreuzung. Die Position einer Straße wird durch die Kreuzungen ermittelt. Wenn eine Kreuzung keine weiteren Straße hat, verweisen die linke und rechte Straße auf null. Wenn eine Kreuzung nur eine weitere Straße hat, verweisen die linke und die rechte Straße auf diese Straße. Eine der wichtigsten Operationen ist das Finden der nächsten Straße links, welche von KreuzungA die Straße BLinks ist und von KreuzungB ARechts ist. Um mögliche Fehler zu vermeiden, hat die Straße die Funktion NächsteStraßeLinks(Startkreuzung), die je nach Startkreuzung die richtige Straße liefert. Eine weitere Funktion ist VorherigeStra-

[3] vgl. Baumgart (1975), S. 15f.

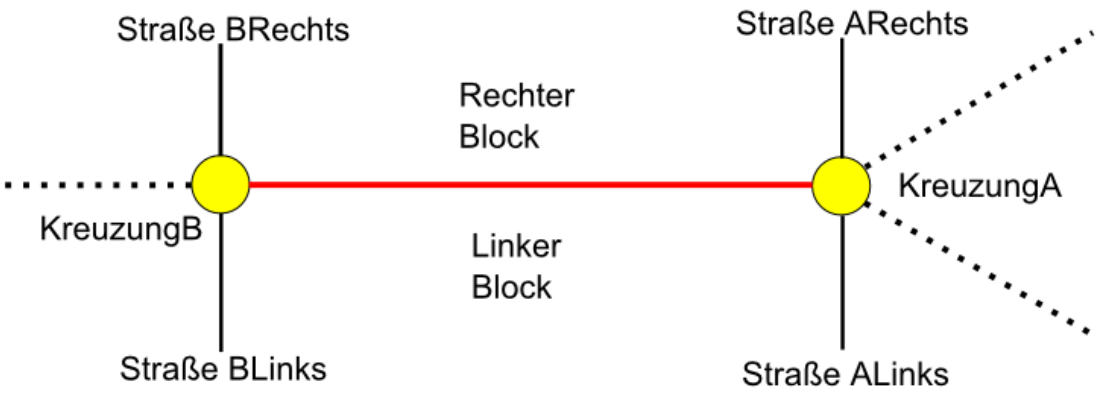


Abbildung 4.3.: Datenstruktur der Straße

ßeRechts(Startkreuzung), welche für KreuzungA die Straße ARechts und für KreuzungB die Straße BLinks zurückgibt.

Kreuzungen speichern ihre Position in der Welt und eine Liste der Straßen, die in der Kreuzung enden. Blöcke haben eine Liste aller äußeren und inneren Straßen, sowie aller Kreuzungen. Die äußeren Straßen eines Blocks bilden einen geschlossenen Weg.

4.3.2. Straßengenerierung

Der Generierungsalgorithmus basiert auf dem in Kapitel 2.2.1 vorgestellten L-System zur Erstellung eines Straßennetzes.

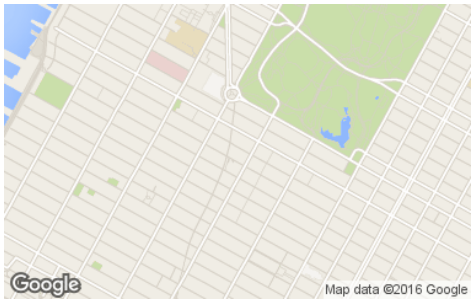
Die Straßengenerierung arbeitet Straßenvorschläge nach ihrer bei der Erstellung durch das jeweilige Straßenmuster gesetzten Priorität ab. Dabei wird immer der Straßenvorschlag mit der höchsten Priorität gewählt und auf Umsetzbarkeit getestet. Wenn die Straße erfolgreich erstellt wurde, werden von dem Ende der erstellten Straße weitere Straßenvorschläge durch das aktive Straßenmuster erstellt.

Straßenmuster

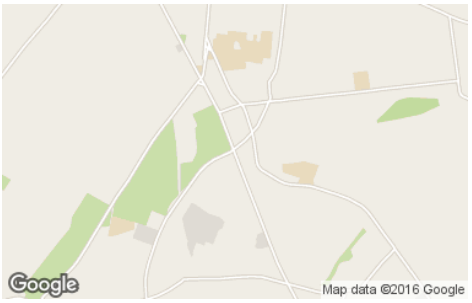
Ein Straßenmuster ist eine Vorlage zur Generierung spezifischer Straßenlayouts. Im Rahmen dieser Arbeit wurden zwei Muster erstellt. Es können weitere Straßenmuster hinzugefügt werden.

Das Blockstraßenmuster ist ein Muster das parallel verlaufende Straßen hat, die in zwei primäre Richtungen zueinander liegen. Ein gutes Beispiel dafür ist Manhattan(siehe Abbildung 4.4a). Dies wird dadurch umgesetzt, das von der letzten erstellten Straße immer drei Straßenvorschläge erzeugt werden, einer der die Straße gerade weiterführt und zwei die die Straße in einem immer gleichen Winkel nach links und rechts fortsetzen.

Das minimale Höhenunterschiedsmuster[5] ist ein Muster das Straßen so erzeugt, dass sie



(a) Auszug des Straßennetz von Manhattan



(b) Hauptstraßen einer Stadt in New Jersey

Abbildung 4.4.: Kartenausschnitte von GoogleMaps[4]

zwischen zwei Orten den besten Weg wählen. Es repräsentiert die Verbindungsstraßen der einzelnen Stadtzentren, die meist die ersten Straßen bei der Entstehung einer Stadt waren und mehr der Gegebenheit der Landschaft folgen als einem Bebauungsplan(siehe Abbildung 4.4b).

Dies wird dadurch umgesetzt, dass in der Richtung der letzten erstellen Straße nach einem Stadtzentrum gesucht wird. Wenn ein Stadtzentrum gefunden wurde, wird ein Straßenvorschlag erstellt, der den Weg mit dem minimalsten Höhenunterschied zwischen dem Ende der Straße und dem Stadtzentrum hat. Um mehr als eine Straße, die zwei Standzentren verbindet, zu erstellen, wird nach einigen Straßenstücken eine Abzweigung zu andren Stadtzentren gesucht. Stadtzentren können entweder vorher bestimmt oder zufällig erzeugt werden.

Straßenvorschlagstest

Der Test prüft, ob ein Straßenvorschlag in das Straßennetz passt, ohne andere Straßen zuschneiden oder ihnen zu nahe zukommen. Der Test basiert auf dem von Kelly und McCabe vorgestellten Snap Algorithmus[6].

[4] Quelle: Google (2016).
[5] vgl. Kelly und McCabe (2007), S 3f.
[6] vgl. Kelly und McCabe (2007), S 6.

Da ein Block immer ein geschlossenes Straßennetz bildet und ein Straßenvorschlag maximal an einer der Außenstraßen des Blocks endet, müssen nur die Straßen des Straßennetzes getestet werden, die Teil des Blockes sind in dem der Straßenvorschlag liegt.

Die vorgeschlagene Straße startet immer von einer existierenden Kreuzung(Startkreuzung) und endet an einem vorgeschlagenen Endpunkt(P_E). Wenn die Startkreuzung bereits Straßen hat, wird ermittelt welche Straße links und rechts von der vorgeschlagenen Straße liegt und ob der Winkel auf beiden Seiten zu den Straßen groß genug ist(siehe Abbildung 4.5). Sollte das nicht der Fall sein, wird der Straßenvorschlag abgelehnt.

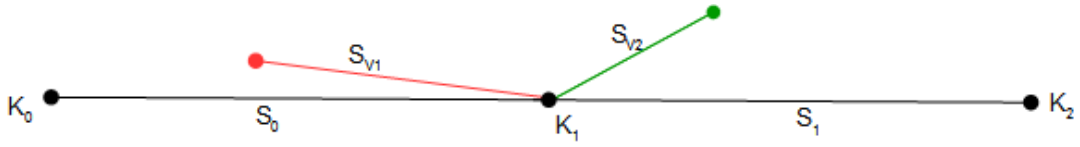
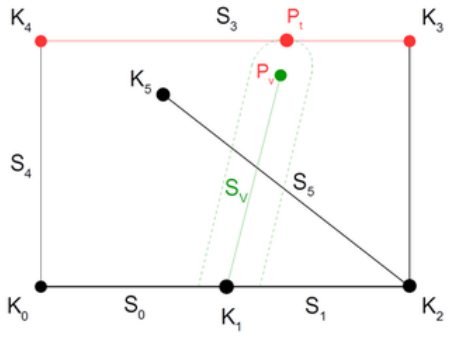


Abbildung 4.5.: Winkeltest des Straßenvorschlages: Der Straßenvorschlag S_{V1} hat einen zu kleinen Winkel zur linken Straße S_0 und wird abgelehnt; der Straßenvorschlag S_{V2} hingegen hat zur linken Straße S_0 und zur rechten Straße S_1 einen ausreichend große Winkel und wird angenommen.

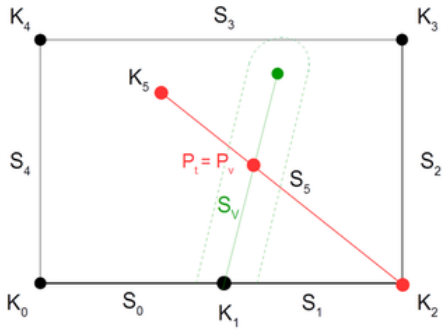
Für jede Straße im Testblock, mit Ausnahme aller Straßen die mit der Startkreuzung verbunden sind, wird geprüft, ob sich die vorgeschlagene Straße mit der jeweiligen Straße schneidet oder zu nahe beieinander liegen. Dies wird in drei Schritten getestet(siehe Abbildung 4.6).

Im ersten Schritt werden die beiden Punkte Pt und Pv berechnet, wo die Teststraße und die vorgeschlagene Straße am dichtesten zueinander sind. Im zweiten Schritt wird geprüft, ob P_V näher am Startpunkt der Straße liegt als ein vorherig gefundener Schnittpunkt(R_{max}). Dies wird mit Hilfe der prozentual zurückgelegten Distanz(R) von P_V auf der vorgeschlagen Straße berechnet. Ist R größer als R_{max}, kann die nächste Straße getestet werden, da die vorgeschlagene Straße sich schon vorher mit einer anderen Straße schneidet und maximal dort enden wird. Im dritten Schritt wird die Distanz zwischen den beiden Punkten berechnet. Wenn die berechnete Distanz kleiner als der Mindestabstand(D_m) zwischen zwei Straßen ist, wird die Teststraße als potentielle Schnittstraße mit Schnittpunkt P_T zwischengespeichert und R_{max} wird auf den Wert von R gesetzt. Ist die Straße von P_S zu P_T kürzer als die Mindestlänge einer Straße, wird der Straßenvorschlag abgelehnt.

Wenn nach dem Test aller Straße im Testblock keine Schnittstraße gefunden wurde, wird für den Endpunkt eine neue Kreuzung erstellt und der Straßenvorschlag zu einer Straße zwischen den beiden Kreuzungen umgesetzt.



(a) Als erstes wird S_V mit S_3 getestet. Da R_{max} noch nicht gesetzt ist, wird direkt getestet, ob sich die Straßen schneiden.Da die Distanz$|P_V - P_T|$ kleiner als D_m ist, wird S_3 als potenzielle Schnittstraße gespeichert und R_{max} gesetzt.



(b) Als nächstes wird S_V mit S_2 getestet. Da R kleiner als der neue R_{max} ist, wird getestet, ob die Straßen sich schneiden. Da $P_V = P_T$ schneiden die Straßen sich und S_2 wird als potenzielle Schnittstraße gespeichert.

Abbildung 4.6.: Beispiel für den Schnitttest

Wenn eine Schnittstraße gefunden wurde, wird geprüft, ob der Schnittpunkt P_T den Mindestabstand zu den Kreuzungen der Schnittstraße einhält. Ist das der Fall, wird in die Schnittstraße am Schnittpunkt P_T eine neue Kreuzung eingefügt und die vorgeschlagen Straße an die neue Kreuzung angeschlossen. Ist das nicht der Fall, wird keine neue Kreuzung erstellt, stattdessen wird die vorgeschlagene Straße an die Kreuzung der Schnittstraße angeschlossen, die zu nah an P_T war. In beiden Fällen muss getestet werden, ob sich ein neuer Block gebildet hat.

4.3.3. Blockfindung

Um zu testen, ob sich durch das Hinzufügen einer Straße ein neuer Block(kleinster Kreis) gebildet hat, wird der in Kapitel 2.2.2 beschriebene Algorithmus zur Findung aller kleinsten Kreise im planaren Graphen verwendet. Im Gegensatz zum beschriebenen Algorithmus muss hier nur geprüft werden, ob die neu hinzugefügte Straße einen neuen kleinsten Kreis erzeugt oder nicht.

Der relevanteste Teil des beschriebenen Algorithmus ist, das zur Findung des kleinsten Kreises von der letzten Kante ausgehend, immer die Kante mit dem größten Winkel gegen den Uhrzeigersinn genommen wird. Was in der gegebenen Datenstruktur der Straße, durch die Funktion NächsteStraßeLinks(Startkreuzung)[7] ohne großen Rechenaufwand bestimmt werden

[7] In Kapitel 4.3.1 vorgestellt

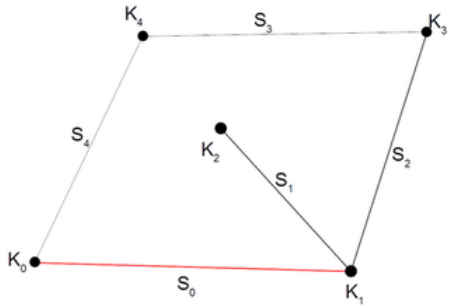
kann. Das erlaubt es die Suche von jeder Straße für einen einzelnen, potentiellen, kleinsten Kreis durchzuführen, ohne alle Knoten zu sortieren und über den ganzen Graph iterieren zu müssen.

Wenn die Startkreuzung der neuen Straße keine weiteren Straßen hat, kann die Suche direkt beendet werden, da sich kein Block gebildet haben kann.

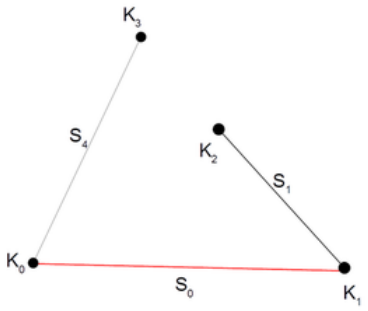
Startend von der neuen Straße wird solange die nächste Straße links genommen, bis entweder die Startstraße gefunden wurde oder keine nächste Straße links existiert.

Wird die Startstraße gefunden, wurde ein kleinster Kreis gefunden und es ist ein neuer Block durch die Straße entstanden.

Wenn keine nächste Straße existiert, wurde ein Filament gefunden. In dem Fall wird wie im ursprünglichen Algorithmus durch Backtracking des gefunden Weges, eine Straße gesucht, die eine rechts vom Weg abzweigende Straße hat. Wenn eine Abzweigung gefunden wurde, wird die Suche des Kreises von der abzweigenden Straße fortgesetzt. Wird die Startstraße beim Backtracking erreicht, ist die neue Straße Teil eines Filaments, folglich ist kein neuer Block entstanden.



(a) Die nächste links(NL) von der Startstraße S_0 ist S_1, S_1 hat keine NL, somit startet das Backtracking. Vorherige rechts von S_1 ist S_2, also wurde eine Abzweigung gefunden, NL von S_2 ist S_3, NL von S_3 ist S_4 und NL von S_4 ist S_0, was die Startstraße ist. Demnach wurde ein neuer Block gefunden.



(b) Die ersten beide Schritte sind gleich, allerdings wird beim Backtracking kein Abzweig gefunden und S_0 erreicht, somit ist Startstraße Teil eines Filaments. Also ist kein neuer Block entstanden.

Abbildung 4.7.: Beispiel des Blockfindungsalgorithmus

Wurde ein neuer Block gefunden, wird ein Block erstellt und die Straßen des gefundenen Weges als äußere Straßen hinzugefügt. Die Straßen der gefundenen Filamente werden als innere Straßen zum Block hinzugefügt und vom alten Block entfernt. Kreuzungen der Filamente, die

nicht Teil des gefunden Weges sind, werden vom alten Block entfernt, da sie innerhalb des neuen Blockes liegen. Alle Straßen, deren Block sich geändert hat, werden angepasst.

4.3.4. Grundstückgenerierung

Die Grundstückgenerierung basiert auf einem Subdivisionsalgorithmus der in Vanegas u. a. (2012) vorgestellt wird. Der Algorithmus teilt ein Grundstück solange bis die entstehenden Grundstücke der gewünschten Größe entsprechen(siehe Abbildung 4.8). Dazu wird das minimal umgebende Rechteck[8] des Grundstückes berechnet, mit dessen Hilfe wird eine Trennlinie erstellt, die das Grundstück in zwei Grundstücke teilt. Die Trennlinie verläuft mittig parallel zur kürzeren Seite des Rechtecks. Sollte eines der beiden entstehenden Grundstücke kein Straßenzugang haben, wird die Trennlinie um 90° gedreht.

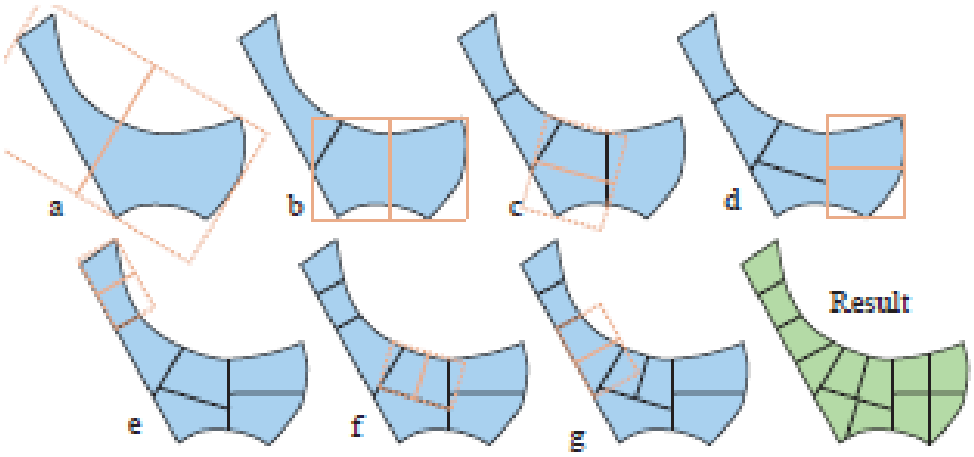


Abbildung 4.8.: Minimal umgebende Rechtecksubdivision[9]

Der Algorithmus erzeugt fehlerhafte Grundstücke, wenn die Trennlinie mehr als zwei Straßen schneidet(siehe Abbildung 4.9), was durch innere Straßen oft vorkommen kann. Der von mir angepasste Algorithmus erkennt, wenn mehr als zwei Straßen die Trennlinie schneiden und erzeugt die richtige Anzahl an Grundstücken.

Der Algorithmus verarbeitet alle Punkte des Grundstückes im Uhrzeigersinn, dabei wird jeweils getestet, ob die Kante zwischen dem aktuellen Punkt und dem nächsten Punkt sich mit der Trennlinie schneidet. Wenn die Kante sich nicht schneidet, wird der nächste Punkt

[8] Minimum Bounding Box
[9] Quelle: Abbildung 6 aus Vanegas u. a. (2012)

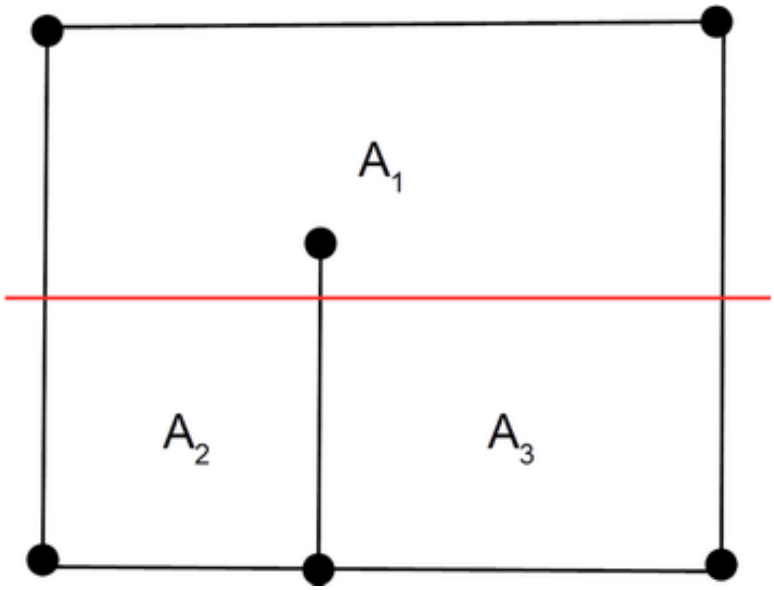


Abbildung 4.9.: Fehler in der Grundstücksteilung: Die Trennlinie(rot) schneidet drei Straßen, der Algorithmus teilt das Grundstück in zwei auf, was dazu führt das A2 und A3 ein Grundstück bilden, was aber durch eine Straße getrennt ist.

der Liste des Teilgrundstückes hinzugefügt. Wenn die Kante sich mit der Trennlinie schneidet, wird der Schnittpunkt am Ende der Liste eingefügt und es wird eine neue Liste für das nächste Teilgrundstück erstellt. Der neuen Liste werden der Schnittpunkt und der nächste Punkt hinzugefügt.

Nach der Aufteilung müssen einige Listen zusammengefügt werden, weil sie zum selben Grundstück gehören. Die erste und letzte Liste gehören immer zusammen und können direkt vereint werden.

Wenn danach mehr als zwei Listen existieren, gib es mindestens eine Liste, die zu einer anderen Liste hinzugefügt werden muss. Es können nur Listen, die auf der gleichen Seite der Trennlinie liegen, verbunden werden. Praktischerweise ist jede zweite Liste auf der gleichen Seite.

Um die Listen zu finden die zum selben Grundstück gehören, wird für alle Listen der ersten Seite getestet, ob der erste Punkt der letzten Liste zwischen den ersten und letzten Punkt einer der anderen Listen liegt. Wenn eine Liste gefunden wird, ist die letzte Liste Teil des Grundstückes. Aus diesem Grund müssen die Punkte am Anfang der gefundenen Liste hinzugefügt werden und der Vorgang wird mit der nächsten Liste wiederholt. Wurde keine Liste gefunden, sind die restlichen Listen vollständige Grundstücke(siehe Abbildung 4.10).

Der Test für die andere Seite ist identisch, nur wird jetzt immer die erste Liste mit allen anderen verglichen.

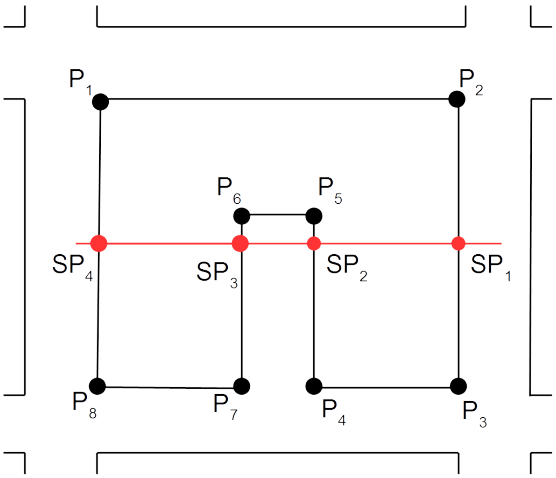


Abbildung 4.10.: Fehlerbehebung nach der Grundstückteilung

- Der Algorithmus teilt die Punkt in fünf Listen auf $L_1\{P_2, S_{P1}\}$, $L_2\{S_{P1}, P_3, P_4, S_{P2}\}$, $L_3\{S_{P2}, P_5, P_6, S_{P3}\}$, $L_4\{S_{P3}, P_7, P_8, S_{P4}\}$ und $L_5\{S_{P4}, P_1\}$.
- Die erste und die letzte Liste(L_1, L_5) werden zu $L_{5,1}\{S_{P4}, P_1, P_2, S_{P1}\}$ vereint.
- Es wird getestet, ob S_{P2} von L_3 zwischen S_{P4} und S_{P1} von $L_{5,1}$ liegt, was der Fall ist, also wird L_3 zu $L_{5,1}$ hinzugefügt $L_{3,5,1}\{S_{P2}, P_5, P_6, S_{P3}, S_{P4}, P_1, P_2, S_{P1}\}$.
- Es wir getestet ob S_{P1} von L_2 zwischen S_{P3} und S_{P4} von L_4 liegt, was nicht der Fall ist, also sind $L_2 und L_4$ vollständige Grundstücke.

Gebäudeplatzierung

Die Gebäudeplatzierung ist eine einfache Umsetzung zur Visualisierung des Ergebnisses der Grundstücksaufteilung und stellt nur sicher, dass sich das zu platzierende Gebäude innerhalb der Grenzen des Grundstückes befindet. Für eine richtige Umsetzung von Stadtgebäuden wäre eine Gebäude- oder Fassadengenerierung notwendig, was nicht Teil dieser Arbeit ist. Durch eine erweiterbar gehaltene Umsetzung der Gebäudeplatzierung ist dies einfach integrierbar.

4.3.5. Straßenvisualisierung

Die Straßenvisualisierung erstellt mit Hilfe einzelner Module wie zum Beispiel 45°-Kurve, T-Kreuzung oder Kreisel die Visualisierung für die Kreuzungen und bildet anschließend mit geraden Straßenstücken die Straßen zwischen den Kreuzungen.

Für jede Kreuzung wird die passende Kreuzungsart ausgewählt. Dies geschieht hauptsächlich nach Anzahl der Straßen, die zu einer Kreuzung gehören.

Wendekreis(1 Straße)

Wenn die Kreuzung nur eine Straße hat, wird ein Wendekreis als Kreuzungsart gewählt und zur Straße ausgerichtet.

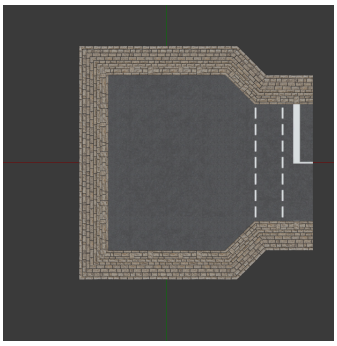


Abbildung 4.11.: Wendekreis Straßenstück[10]

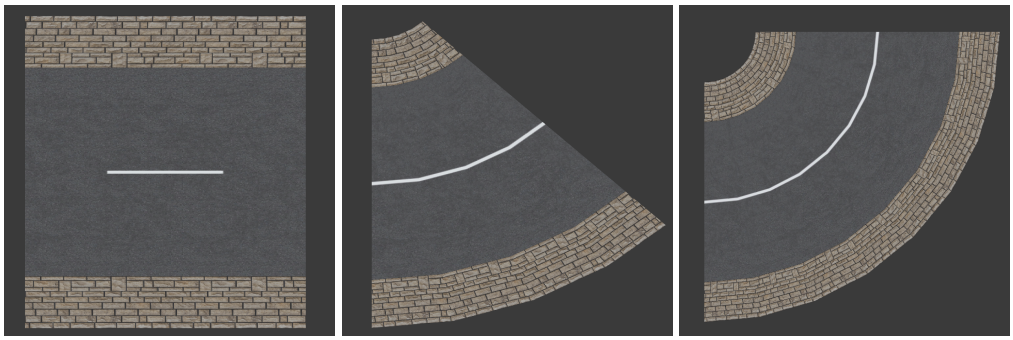
Kurve(2 Straßen)

Wenn die Kreuzung zwei Straßen hat, wird die Kreuzung so ausgerichtet, das die Winkel beider Straßen zu dem jeweiligen gegenüberliegenden Anschlusspunkt gleich sind. Wenn die Winkel kleiner als 11° sind wird ein gerades Straßenstück als Kreuzungsart gewählt, sonst wird ein Kurvenstück (22°/45°/90°) gewählt, so dass der Restwinkel beider Anschlusspunkte am kleinsten ist.

Kreuzung mit mehr als zwei Straßen

Wenn die Kreuzung 3-6 Straßen hat, wird die Kreuzung an den zwei Straßen ausgerichtet, die am geradesten zueinander sind. Im nächsten Schritt werden alle Straßen im Uhrzeigersinn,

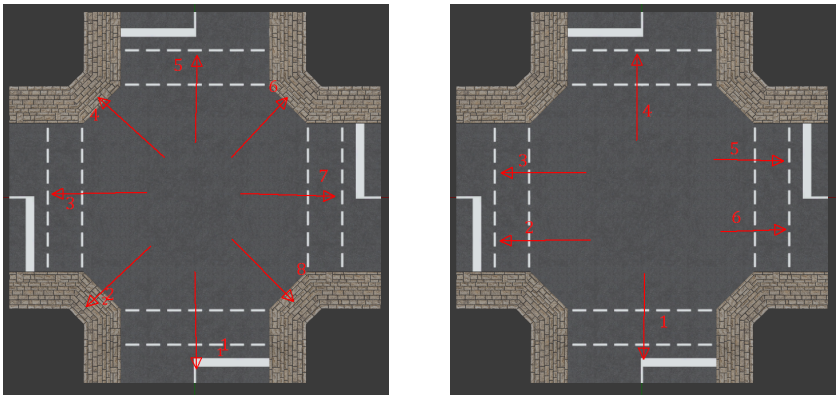
[10] Alle Straßenstücke wurden selbst modelliert und mit Texturen von Textures.com versehen.



(a) Gerades Straßenstück (b) 45°-Straßenstück (c) 90°-Straßenstück

Abbildung 4.12.: Straßenstücke, die bei Kreuzungen mit zwei Straßen verwendet werden

ausgehend von der Straße nach dem die Kreuzung ausgerichtet wurde, auf acht Referenzrichtungen aufgeteilt, um die optimale Kreuzungsart zu bestimmen und sicherzustellen, dass alle Straßen überschneidungsfrei an die Kreuzung angeschlossen werden können.



(a) Kreuzung mit allen acht Referenzrichtungen.

(b) Kreuzung nach dem zwei Referenzrichtungen entfernt wurden.

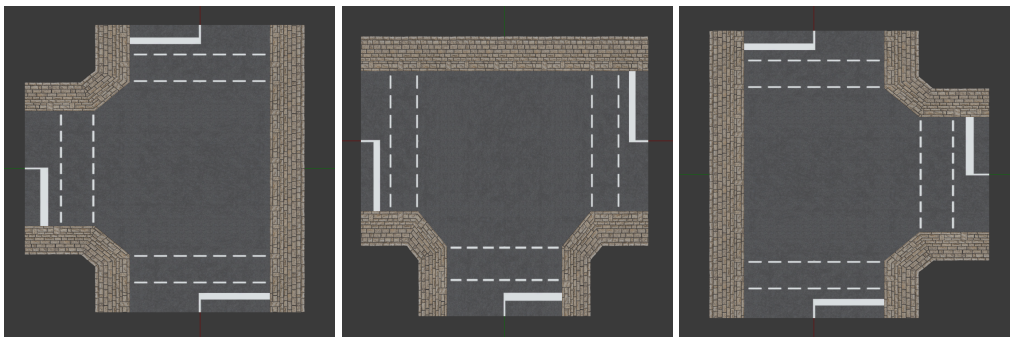
Abbildung 4.13.: Darstellung der Referenzrichtungen

Der vorherige Algorithmus hat die maximalen sechs Straßen auf acht Referenzrichtungen aufgeteilt. Um die weitere Verarbeitung zu vereinfachen, werden zwei leere Referenzrichtungen entfernt.

Sollte Referenzrichtung 5 nicht belegt sein und die linke oder rechte Seite mehr Straßen haben, wird die Referenzrichtung 5 und eine leerer Referenzrichtung auf der Seite mit weniger Straßen gelöscht. Damit wird die vorherige linke oder rechte oberste Referenzrichtung auf der Seite mit mehr Straßen zur neuen Referenzrichtung 4. Wenn Referenzrichtung 5 belegt ist, wird jeweils links und rechts ein leerer Referenzrichtung entfernt.

T-Kreuzung(3 Straßen)

Im Fall von drei Straßen wird ein T-Stück als Kreuzungsart gewählt und es muss entschieden werden, wie das T-Stück zur ersten Straße ausgerichtet wird. Dies kann leicht anhand der belegten Referenzrichtungen entschieden werden. Wenn die gegenüberliegende Referenzrichtung 4 unbelegt ist, ist nur Fall(b) (siehe Abbildung 4.14) möglich. In den beiden anderen Fällen ist entweder die Referenzrichtung 2/3 belegt, was Fall(a) bedeutet oder die Referenzrichtung 5/6, was Fall(c) bedeutet. Je nach Fall wird jede Straße der jeweilige belegten Referenzrichtung einem Anschlusspunkt der T-Kreuzung zugeordnet.



(a) links und gegenüber (b) links und rechts (c) rechts und gegenüber

Abbildung 4.14.: T-Kreuzungsstück

X-Kreuzung(4 Straßen)

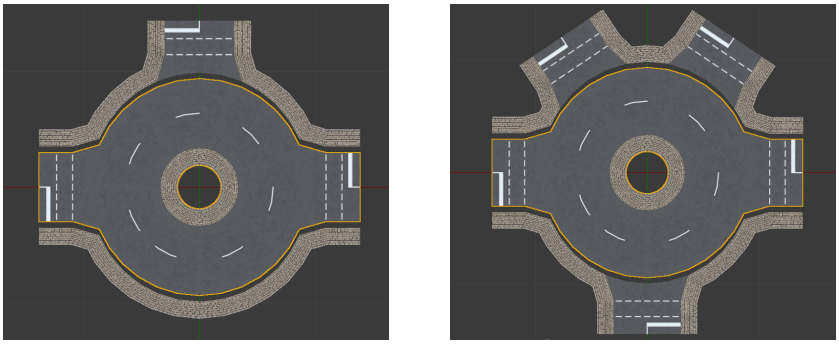
Im Fall von vier Straßen kann es vorkommen, dass beide Referenzrichtung 2/3 auf der linken Seite oder beide Referenzrichtung 5/6 auf der rechten Seite belegt sind. Wenn das der Fall ist, wird als Kreuzungsart der Kreisel gewählt. Ansonsten wird ein X-Stück als Kreuzungsart gewählt und jede Straße einem der vier Anschlusspunkte zugeordnet.

Kreisel(4-6 Straßen)

Im Fall von 4-6 Straßen werden als erstes die Straßen der Referenzrichtung 1/4, den Anschlusspunkten 1/3 des Kreisels zugeordnet(siehe Abbildung 4.15: der linke und der rechte Anschlusspunkt). Da diese immer vorhanden sind.

Anschließend wird für jede Seite geprüft wie viele Straßen die Referenzrichtung 2/3 und 5/6 belegen(oben und unten in der Abbildung). Wenn eine Seite von keiner Straße belegt ist, wird ein reines Gehwegstück auf der Seite des Kreisels angesetzt. Im Fall von einer Straße wird das Abfahrtstück mit einem Anschlusspunkt in der Mitte gewählt und bei zwei Straßen wird das

Abfahrtstück mit zwei Anschlusspunkten gewählt. Die Straßen werden als nächstes an die neu entstandenen Anschlusspunkte angeschlossen.



(a) Kreisel mit oben dem Stück für eine Straße und unten das reine Gehwegstück

(b) Kreisel mit oben dem Stück für zwei Straßen und unten für eine Straße

Abbildung 4.15.: Kreiselstücke

Anpassung der Anschlusspunkte

Nach der Wahl der Kreuzungsart wird für jeden Anschlusspunkt der Kreuzungsart der Winkel zwischen angeschlossener Straße und der Richtung des Anschlusspunktes berechnet. Ist der Winkel größer als 11° , werden solange Kurvenstücke angeschlossen bis der Winkel zwischen 0° bis 11° liegt. Wenn der Winkel kleiner als 11° ist, wird die Postion und Richtung des Anschlusspunktes der Straße mitgeteilt.

Straßen zwischen den Kreuzungen

Nachdem alle Kreuzungen erstellt wurden, müssen noch die geraden Strecken zwischen den Kreuzungen durch gerade Straßenstücke verbunden werden. Da immer noch bis zu 11° Richtungsunterschied zwischen dem Anschlusspunkten und der Straße bestehen kann und die Straßenstücke eine feste Länge haben, wird das erste und letzte Stück der Straße mit einer Spline[11] so gebogen und gestreckt, dass ein nahtloser Übergang zwischen Straße und Kreuzung entsteht. Dazu kann das von Unreal Engine 4 gegebene Spline Mesh[12] benutzt werden.

[11] vgl. Shikin und Plis (1995), S 96.
[12] vgl. Epic Games (Splines).

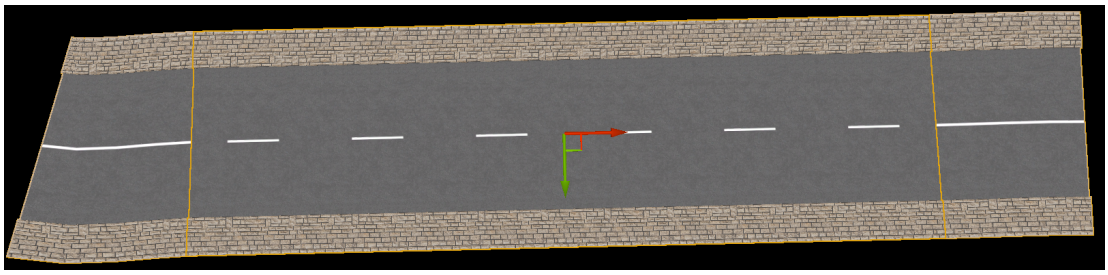


Abbildung 4.16.: Straßenstück mit hervorgehobenen Splines an beiden Enden: An der linken Seite kann man die Verformung besonders gut sehen.

4.4. Manueller Editor

Das Verändern des generierten Straßennetzes per Hand ist im Editor generell möglich. Allerdings wird beim Verändern einer Straße oder Kreuzung das restliche Straßennetz nicht automatisch informiert, was zu Fehlern in der Visualisierung oder dem unterliegenden Graphen führen würde. Unreal Engine 4 bietet zwei Ansätze, um mit solchen Problemen umzugehen.

Der erste Ansatz sind eine Reihe von Funktionen(*PostMove*, *PostDelete*, *PostUndo*), die nach jeder Veränderung eines Objektes aufgerufen werden. Allerdings wird jedem Objekt nur die Änderung an sich selbst mitgeteilt, was es sehr aufwändig macht, sicherzustellen das der unterliegende Graph fehlerfrei bleibt, wenn mehr als ein Objekt verschoben wird.

Der zweite Ansatz sind verschiedene Editor Modi mit dem sich die gesamte Interaktion mit dem Editor verändern lässt. Der Editor hat bereits mehrere Modi zum Beispiel dem Landscape Editor Modus[13], der speziell für die Bearbeitung von Terrains zuständig ist. Der Editor ist leicht um zusätzliche Modi erweiterbar, da alle Editor Modi unter einem Interface[14] implementiert wurden. Bei dem nur die benötigten Funktionen implementiert und anschließend der angepasste Editor Modus beim Start des Unreal Editors registriert werden muss.

Der zweite Ansatz wurde gewählt, da er leichter umzusetzen ist und einen eigenen Toolbarbereich hat, wo das Benutzerinterface für den Generator implementiert werden kann. Des Weiteren wird hierdurch die Straßengenerierung und -bearbeitung klar von den andren Elementen des Editors getrennt.

[13] vgl. Epic Games (Landscape Editor).
[14] vgl. Epic Games (Editor Mode).

Straßen/Kreuzungen bewegen

Der Editor Modus hat eine Funktion die beim Verschieben oder Drehen von Objekten aufgerufen wird. Diese Funktion bekommt als Input die ausgewählten Objekte und die relative Verschiebung/Drehung des Transformationswidget. Im unverändert Editor Modus würde diese Transformation auf alle ausgewählten Objekte angewandt. Was allerdings zu Problemen führt, wenn Straßen und die dazugehörigen Kreuzungen gleichzeitig verschoben werden. Da die Postionen von Straßen nur durch die Start- und Endkreuzung definiert sind, was dazu führt das die selbe Kreuzung mehrfach verändert wird. Um das Problem zu beheben, wird im angepassten Editor Modus, eine Liste aller zu verändernden Kreuzungen erstellt und nur diese werden angepasst. Nach dem alle Kreuzungen verändert wurden, wird die Visualisierung für alle dazugehörigen Straßen angepasst.

Straßen/Kreuzungen löschen

Beim Löschen von Straßen und Kreuzungen kommt es zu ähnlichen Problemen wie beim Bewegen, da beim Löschen einer Kreuzung auch die Straße gelöscht wird. Wenn eine Kreuzung und eine dazugehörige Straße ausgewählt sind, würde der unveränderte Editor Modus versuchen dieselbe Straße zweimal zu löschen. Im angepassten Editor Modus wird eine Liste aller Straßen, die ausgewählt sind oder durch eine ausgewählte Kreuzung gelöscht würden, erstellt und nur diese Straßen werden gelöscht. Kreuzungen werden automatisch gelöscht, wenn die letzte Straße entfernt wurde.

Undo

Um die Undo-Funktion des Editors nutzen zu können, müssen alle Interaktionen mit Hilfe des eingebauten Transaktionssystem durchgeführt werden. Dazu muss eine Transaktion erstellt werden und jedes Objekt, das verändert werden soll, vor der Veränderung durch Aufrufen der Modify-Funktion markiert werden. Danach kann jede Action im Editor rückgängig gemacht werden.

Straßen/Kreuzungen hinzufügen

Da das Hinzufügen von Straßen und Kreuzungen in das Straßennetz per Hand sehr aufwändig und fehleranfällig wäre, wurde die Mausklickverarbeitung des Editor Modus angepasst. Mit der angepassten Funktion wird bei einem Mausklick mit gedrückter Alttaste auf das Terrain automatisch ein Straßenvorschlag erzeugt, der von der ausgewählten Kreuzung zu dem an-

geklickten Punkt im Terrain geht. Der Straßenvorschlag wird vom selben System wie bei der Generierung getestet.

Auf ähnliche Weise kann eine Kreuzung in eine Straße eingefügt werden. In dem mit gedrückter Alttaste auf eine Stelle der ausgewählten Straße geklickt wird, wo die neue Kreuzung entstehen soll. Die Straße wird dann an der Stelle geteilt und eine neue Kreuzung eingefügt.

5. Umsetzung

5.1. Fahrrad

5.1.1. Message Broker Implementierung

Um die C++ Bibliothek des Message Brokers in Unreal Engine 4 zu benutzten, muss beim Start der Simulation eine Connection zum Message Broker hergestellt werden und einzelne *MessageHandler* und *-Producer* für die jeweiligen Topics registriert werden.

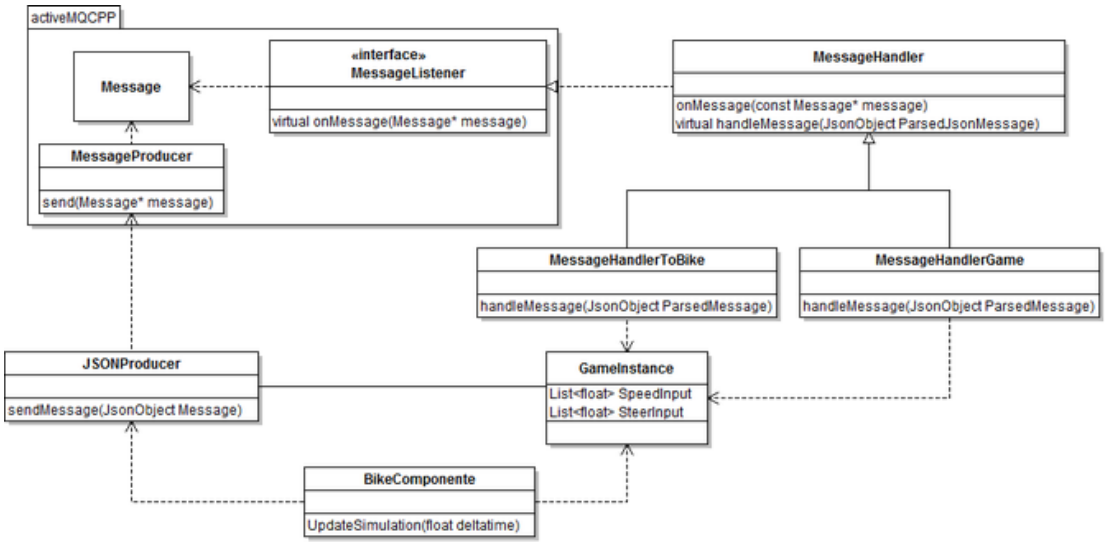


Abbildung 5.1.: Klassendiagramm der Kommunikationsschnittstelle

Um beim Start der Simulation die Connection zu erstellen, wird die *Init*-Funktion der *GameInstance*-Klasse der Unreal Engine 4 erweitert. Die *GameInstance*-Klasse wird benutzt, um levelübergreifende Prozesse und Daten zu verwalten. Sie wird beim Start der Simulation initialisiert und bleibt beim Wechsle zwischen einzelnen Levels erhalten, bis die Simulation beendet wird.

Für jedes Topic wurde eine *Message Handler*-Unterklasse erstellt, in der die *handleMessage*-Funktion so angepasst wurde, dass die zugehörigen JSON Nachrichten verarbeitet werden können.

Da die *Message Handler* in einem eignen Thread laufen und alle von Unreal Engine 4 verwalteten Objekte keine direkte Änderung von anderen Threads zulassen, musste eine zentrale Schnittstelle implementiert werden. Hierfür wurde die *GameInstance*-Klasse genutzt, weil hierdurch die gesamten Abhängigkeiten in einer einzigen Klasse verwaltet werden und die Fahrradimplementierung leicht ausgetauscht werden kann.

Die vom *Message Handler* empfangenen Inputdaten werden in threadsicheren Listen gespeichert. Diese Listen werden vor dem Berechnen des Physikschrittes von der Update Funktion des Fahrrads ausgelesen und geleert.

Sollte sich die Steigung geändert haben, wird eine Nachricht mit Hilfe des *Message Producer* gesendet. Hierfür muss die Steigung in das JSON Format umgewandelt werden und als nächstes mit Hilfe der *SendMessage*-Funktion des *Message Producer* gesendet werden.

5.1.2. Implementierung des Fahrrades

Da die Unreal Engine 4 eine eigene Fahrzeugimplementierung mitbringt(siehe Grundlagen 2.1.3), musste in der *Fahrzeug*-Klasse nur die *Update*-Funktion um die Berechnung des Drehmoments und der Rückgabe der Steigung erweitert werden. Zur Ermittlung der Steigung wird der von Unreal Engine 4 zur Verfügung gestellte *Forward*-Vektor des Fahrrads genutzt, der immer parallel zur Horizontalen des Fahrrades liegt und durch den Vergleich mit dem *WorldUp*-Vektor die Steigung liefert.

Danach musste ein Fahrrad Mesh als Visualisierung ausgewählt werden. Des Weiteren mussten die Werte der Räder(Anzahl/Radius/Position), sowie Gewicht und Windwiderstand passend zu dem Fahrrad Mesh eingestellt werden. Als nächstes musste eine Kamera relativ zum simulierten Fahrrad platziert und ausgerichtet werden(siehe Abbildung 5.2).

Am Schluss wurde das simulierte Fahrrad durch mehrere Tests am Ergometer kalibriert. Dabei wurden besonders die Kameraposition sowie der Lenkeinschlag zur Stellung des Lenkers des Ergometers angepasst.

5.1.3. Event Nachricht

Die Grundlage für das Senden von Nachrichten wurde bereits für die Steuerung des Fahrrads geschaffen. Deswegen musste nur das Senden von Nachrichten im EmoBike Nachrichtenformat durch *Blueprints* ermöglicht werden.

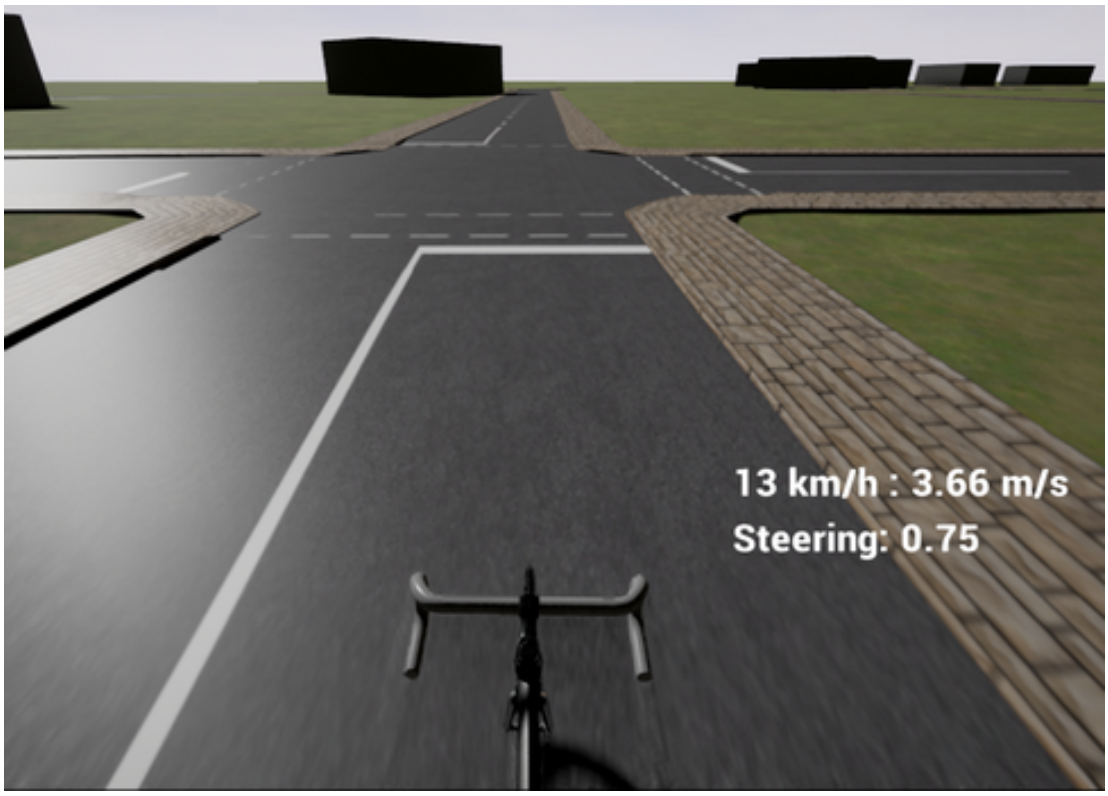


Abbildung 5.2.: Sicht aus der Kameraperspektive des Fahrrades

Dazu wurde eine Funktion erstellt, die eine Type-Variable und einen Liste mit KeyValue-Paaren in das richtige Nachrichtformat umwandelt.

Damit die Funktion in Blueprints benutzbar ist, muss nur das *UFunction*-Macro mit *Spezifizier BlueprintCallable* benutzt werden.

Blueprint Funktion

```
UFUNCTION(BlueprintCallable, Category = "Message")
static void SendMessageEvent(FString type, TArray<FDataPair> data);
```

Zum Erzeugen der richtigen Nachricht muss die Funktion zwischen drei Datentypen unterscheiden können. Wenn die Liste leer ist, wird eine Nachricht mit $DataType = None$ erzeugt. Wenn die Liste nur ein Element enthält, wird eine Nachricht mit $DataType = KeyValue$ und dem $Data$-Feld mit dem Wert des KeyValue-Paares erzeugt. Ansonsten wird eine Nachricht mit $DataType = Map$ und der KeyValue-Liste als JSON-Array im $Data$-Feld erzeugt.

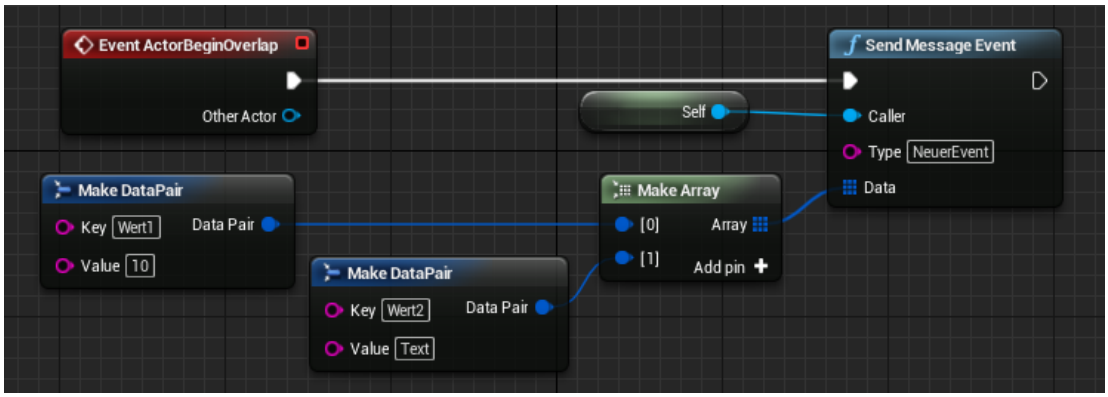


Abbildung 5.3.: Beispiel zur Versendung einer Eventnachricht an den Message Broker

5.2. Generierung

5.2.1. Datenstrukturen

Die in Kapitel 4.3.1 vorgestellten Strukturen für Straße, Kreuzung und Block wurden als Unterklassen zu der Unreal Engine Klasse *AActor* erstellt.

Die *AActor*-Klasse ist die Basisklasse für alle Objekte, die in einem Level erstellt werden können. Die Klasse ist modular aufgebaut und kann um einzelne Komponenten erweitert werden, wie zum Beispiele *UStaticMeshComponente*, die es ermöglicht ein Mesh zu dem Actor hinzuzufügen.

Um eine Unterklasse zu *AActor* zu erstellen, muss einer Klasse zwei Makros hinzugefügt werden. Das *UCLASS()*-Makro muss vor der Deklaration und das *GENERATED_BODY* im Inneren der Deklaration hinzugefügt werden. Die beiden Makros zusammen sorgen dafür, dass die Engine die Klasse kennt und um generierte Funktionen erweiterten kann. Um Komponenten hinzuzufügen, müssen diese im Konstruktor mit Hilfe des *FObjectInitializer* erstellt werden.

Beispiel anhand der Straßenimplementierung

```
1 UCLASS()
2 class CITYGEN_API AStreet : public AActor
3 {
4         GENERATED_BODY()
5 public:
6         UMeshComponent* Mesh;
7         AStreet(const FObjectInitializer& O)
8         {
9                 Mesh = O.CreateDefaultSubobject<UMeshComponent>(
```

```
10                  this , TEXT("Mesh"));
11          }
```

Im Beispiel wird die Straßenklasse erstellt und eine Meshkomponente hinzugefügt.

Alle Funktionen des Generierungsalgorithmus wurden entweder als statische Funktion in der Klasse StreetGenerator erstellt oder sind Teil von Straße-, Kreuzung- oder Blockklasse. Für die Straßenmuster wurde ein Klasse *Muster* mit der Funktion *ApplyPattern* erstellt, von der jedes andere Muster erbt.

Die Generierung wird in einem eigenen Thread ausgeführt. Dazu wird die von Unreal Engine zur Verfügungen gestellte *FRunable* benutzt. Diese erstellt einen gewöhnlichen Thread. Einziger Vorteil ist das die Implementierung nicht vom Betriebssystem abhängt.

5.2.2. Straßen

Da die meisten Straßentests nur die Position der Kreuzungen nutzen, sind so gut wie alle Tests mit Hilfe von Funktionen umgesetzt, die in den von Unreal Engine 4 implementierten *FMath*-, *FVector*- und *FVector2D*-Klasse gegeben sind. Eine der wichtigsten Funktionen ist *FMath::SegmentDistToSegment*($A, B, C, D, OutP_{AB}, OutP_{CD}$), die für die zwei Geraden(AB, CD) die Punkte($OutP_{AB}$/$OutP_{CD}$) auf der jeweiligen Geraden findet, an dem die Geraden am dichtesten zu anderen sind. Wenn die Punkte gleich sind, schneiden sich die Straßen an der Stelle. Mit dem Ergebnis wird der Wert R berechnet, um zu bestimmen, ob die Straße weiter entfernt ist als eine vorherige gefunden Straße, sowie die Distanz zwischen den zwei Straßen.

Da die von der Unreal Engine 4 verwalteten Objekte nur im Hauptthread erzeugt werden können, musste die Straßengenerierung, so verändert werden, dass ein *FTask* zum Erzeugen eines neuen Objektes erstellt wird und nicht direkt ein neues Objekt. Der Thread muss dann auf die Erstellung des Objektes im Hauptthread warten, was jedoch meistens zeitnah geschieht.

FTask sind ein Multi-Threading Tool von Unreal Engine 4 zum Verwalten von Aufgaben, die in einem bestimmten Thread durchgeführt werden müssen. Im Hauptthread des Editors werden bei jedem Durchlauf alle anstehenden Tasks abgearbeitet.

5.2.3. Grundstücke

Um Grundstücke aufzuteilen, müssen als erstes die Innenpunkte des Blockes bestimmt werden. Dazu wird startend von einer Außenstraße des Blockes über alle Straßen iteriert und von jedem Start-/Endpunkt der Straße, der Innenpunkt des Blockes berechnet.

Für die Liste der Innenpunkte wird mit Hilfe des Rotating Caliper Algorithmus[1] das kleinste umschließende Rechteck bestimmt.

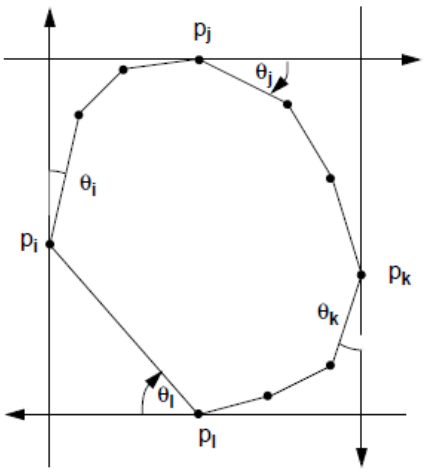


Abbildung 5.4.: Start des Rotating Caliper Algorithmus

In Abbildung 5.4 ist ein Polygon mit zwei um 90° verschobenen Messschieber zu sehen. Diese werden immer zu der Kante ausgerichtet, die von dem Punkt an dem sie anliegen, den kleinsten Winkel Θ hat. Bei jeder Drehung wird die Fläche zwischen den Messschiebern berechnet und die Seite des Messschieber von dem gewählten Winkel zum nächsten Punkt geschoben. Nachdem alle Kanten abgearbeitet sind, ist das kleinste umschließende Rechteck gefunden, welches die kleinste Fläche hat.

Mit diesem Rechteck kann wie im Konzept 4.3.4 gezeigt, die Mittellinie berechnet und das Grundstück geteilt werden. Um zu testen, ob die entstehenden Teilgrundstücke noch Straßenzugang haben, wird getestet, ob mindesten eine Kante auf einer der Straßen des Blockes liegt.

[1] vgl. Toussaint (1983); Caliper - Messschieber.

5.3. Benutzerinterface des Straßengenerators

Das Interface nutzt Slate UI Framework[2], welches zum Erstellen des Unreal Editor Interface verwended wird. Das Framework ist leicht erweiterbar und der Editor hat viele Einstiegspunkte, an den Veränderungen registriert werden können.

Jede dem Editor bekannte Klasse hat eine Detailansicht die alle Variablen, die mit dem *UProperty*-Makro deklariert wurden, automatisch anzeigt.

```
UPROPERTY(EditAnywhere, Category = "StrGen")
float StreetLength
```

Die Detailansicht kann für jede Klasse durch die Implementierung des Interface *IDetailCustomization* angepasst werden. Dabei können in der Funktion $CustomizeDetails$ des Interfaces weitere Slate-Elemente hinzugefügt oder bestehende angepasst werden.

Das Interface wurde genutzt, indem eine Konfigurationsklasse erstellt wurde, in der alle Optionsfelder des Generators als Variable enthalten sind. Die Konfigurationsklasse wurde um Buttons zum Starten der einzelnen Generierungsschritte erweitert. Des Weiteren wurden alle Felder und Buttons so angepasst, dass sie nur sichtbar sind, wenn sie in dem jeweiligen Generierungsschritt anwendbar sind.

Slate Beispiel

```
1  CustomizeDetails(IDetailLayoutBuilder& Layout)
2  {
3      IDetailCategoryBuilder& Category = Layout.EditCategory("StrGen");
4      .Visibility(&Gen::CanStartGen)
5      Category.AddCustomRow("GenButton")
6      [
7          SNew(SButton)
8              .OnClicked(Gen.ToSharedRef(),&Gen::StartGen )
9              .Content()
10             [
11                 SNew(STextBlock)
12                 .Text("Start Generation")
13             ]
14     ]
15 }
```

[2] vgl. Epic Games (Slate).

In dem Beispiel wird in Zeile 7-14 ein Button zu der Kategorie *StrGen*(Zeile 3) hinzugefügt. Beim Drücken des Buttons wird die Funktion *StartGen* aufgerufen(Zeile 8). Der Button enthält ein Textelement(Zeile 11) mit dem Text "*Start Generation*". Die gesamte Kategorie ist nur sichtbar, wenn die Funktion *CanStartGen*(Zeile 4) *true* zurück liefert.

Abschließend muss die Detailmodifikation beim Start des Editors registriert werden und wird automatisch benutzt, wenn eine Detailansicht zu der Klasse geöffnet wird. Um das manuelle Öffnen zu vermeiden, wurde die Detailansicht dem Straßeneditormodus hinzugefügt.

6. Evaluation

Für die Fahrradsimulation wurde erfolgreich ein Prototyp erstellt, der nahtlos in das bestehende System eingefügt werden konnte. Die Simulation läuft selbst bei großen Straßennetzen ohne merkbare Verzögerung. Einzige Ausnahme ist die Verzögerung bei Änderungen in der Steigung, was allerdings an der langsamen Anpassung des Tretwiederstandes beim Ergometer liegt.

Der für den Straßeneditor erstellte Prototyp kann mit dem Grundwissen über die Bedienung des Unreal Editors, welches durch ein kurzes Turoial erlangt werden kann, ohne große Fachkenntnisse über Leveldesign benutzt werden, um in wenigen Schritten ein Straßennetz zu generieren.

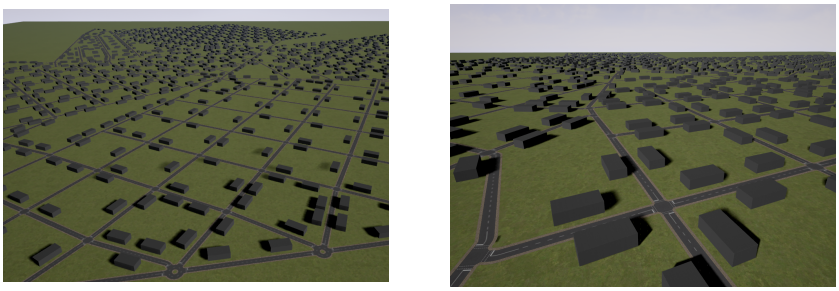


Abbildung 6.1.: Screenshots eines generierten Straßennetzes mit Gebäuden

Die Straßenvisualisierung erstellt alle Kreuzungsarten fehlerfrei und ohne sichtbare Übergänge. Zudem wird die Visualisierung beim Verändern von Straßen und Kreuzungen in Echtzeit ohne merkliche Verzögerungen angepasst.

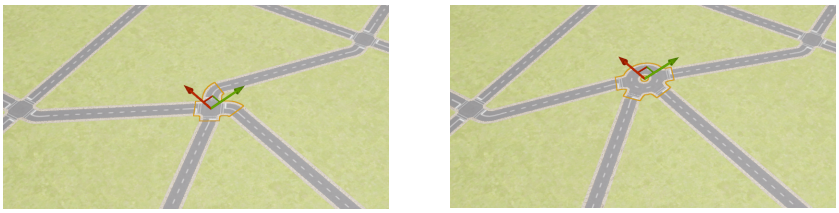


Abbildung 6.2.: Screenshot einer Kreuzung vor und nach dem Verschieben

Das angepasste Editorinterface für den manuellen Editor erlaubt es einzelne Straßen und Kreuzungen mit den normalen Tools des Editors anzupassen, allerdings ist die Auswahl von Straßen aus größerer Entfernung nicht immer einfach, da die Straßen direkt auf dem Terrain liegen und der Editor nicht genau erkennen kann, ob die Straße oder das Terrain angeklickt wurde.

In einem Test wurde geprüft, wie sich der Generator bei verschiedener Anzahl zu generierenden Straßen verhält. Hierfür wurde das Blockmuster verwendet. Da es kein Zufall enthält und beim Anwenden auf drei quadratische Straßenblöcke, die jeweils doppelt so groß sind wie der Vorgänger, jeweils doppelt so viele Straßen erzeugt. Zur Messung wurde die Zeit vom Start bis zum Ende der Straßengenerierung gemessen, sowie wie viel der gemessenen Zeit für die Erstellung von Objekten durch den Gamethread verbraucht wurde.

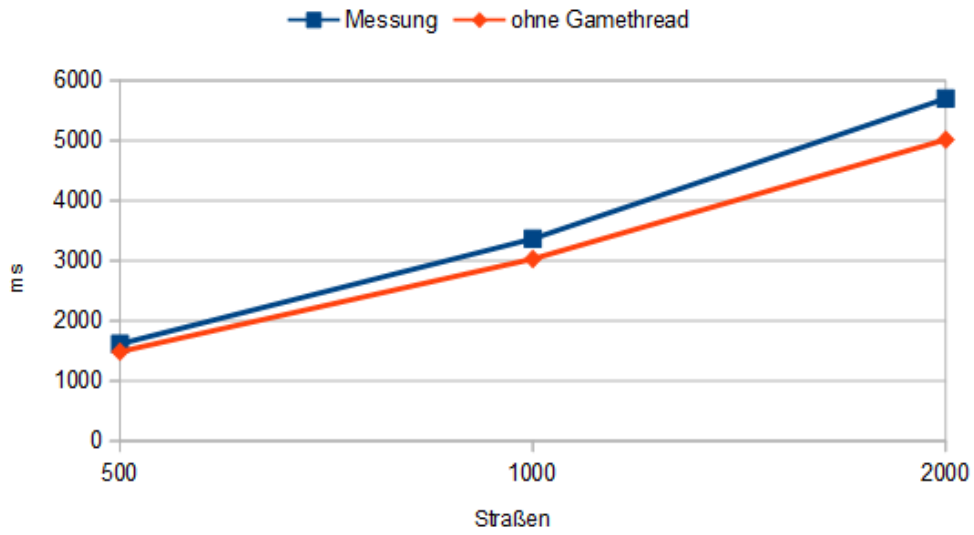


Abbildung 6.3.: Graph mit Messergebnissen des Straßengeneratortests

Die Messung zeigt, das der Generierungsalgorithmus recht konstant bleibt, aber der Editor mit dem Erstellen der Straßenobjekte das Ganze leicht verzögert.

7. Fazit

7.1. Zusammenfassung

In dieser Arbeit wurde eine Fahrradsimulation für das EmoBike Projekt entwickelt, sowie ein Karteneditor, in dem Straßennetze für die Fahrradsimulation generiert und bearbeitet werden können.

Die Fahrradsimulation nutzt die Fahrzeugimplementierung der Unreal Engine 4, die zur Steuerung durch ein Fahrradergometer an das EmoBike System angepasst wurde.

Der Karteneditor wurde als neuer Editormodus in den Unreal Editor eingefügt und kann genutzt werden, um Straßennetze zu bearbeiten und mit dem für diese Arbeit entwickelten Generator zu erzeugen.

Der Generator basiert auf einem erweitertem L-System, was genutzt wird, um Straßenvorschläge anhand von Mustern zu erzeugen. Diese Straßenvorschläge werden auf Umsetzbarkeit in dem bereits erzeugten Straßennetz getestet.

Der Editor ist für einen Anwender, der keine Fachkenntnisse besitzt, nach kurzer Einweisung benutzbar und in der Lage Straßennetze für eine Kleinstadt zeitnah zu generieren, was in einem Versuch nachgewiesen wurde.

7.2. Ausblick

Fahrradsimulation

Die Fahrradsimulation könnte verbessert werden, indem der Verlauf des nächsten Straßenabschnittes auf Veränderung der Steigung geprüft wird. Wenn eine Änderung gefunden wird, kann diese frühzeitig an die Kontrollkomponente gemeldet werden, um die Latenz beim Einstellen des Tretwiderstandes am Ergometer zu minimieren.

Des Weiteren könnte die gesamte Geschwindigkeitsberechnung, die zur Zeit in der Kontrollkomponente des EmoBike System umgesetzt ist, durch die zentrale Verarbeitung aller Eingabewerte im Update-Schritt der Simulation ersetzt werden. Dies würde es ermöglichen Events zu erzeugen, die direkt Einfluss auf die Berechnungen nehmen, was zur Zeit ein großer

Aufwand ist, da zwei eigenständige System angepasst werden müssen. Außerdem könnte wahrscheinlich die Latenz weiter minimiert werden.

Editor

Da für diese Arbeit nur zwei Straßenmuster erstellt wurden, die zusammen ein häufig vorkommendes Straßennetz einer Stadt bilden, könnten diese Muster um mehr Variationen erweitert werden. Außerdem könnten neue Muster für Orts- oder Landstraßen hinzugefügt werden, was noch mehr Szenarien ermöglichen würde. Das Straßennetz selber könnte um die Generierung von Schilder und Straßenlaternen, sowie funktionierenden Ampelanlangen erweitert werden.

Des Weiteren könnte, wie bereits in Kapital 4.3.4 erwähnt, für die Gebäudeplatzierung ein komplexerer Algorithmus entwickelt werden, was ein deutlich realistischeres Stadtbild erzeugen würde.

A. Anhang

A.1. CityGen Tutorial

A.1.1. Grundvoraussetzung

Bevor mit dem Tutorial begonnen wird, sollten die UE4 Tutorial zur Instalation[1], Level Editor Quickstart Tutorial[2] und Blueprint Quickstart[3] gelesen werden.

A.1.2. Erstellung eines Stadtlevels

1. Neues Level erstellen
   Im Editor unter **File->New Level** das Template **CityGen** auswählen.

2. In den CityGen Editormodus wechseln
   Um Zugriff auf die Funktion des Straßeneditors zu haben, muss in den Editormodus gewechselt werden. Die Editormodi befinden sich auf der linken Seite des Editors. Um in den Citygen Editormodus zu wechseln, muss der 6. Tab ausgewählt werden(siehe Pfeil in der Abbildung A.1)

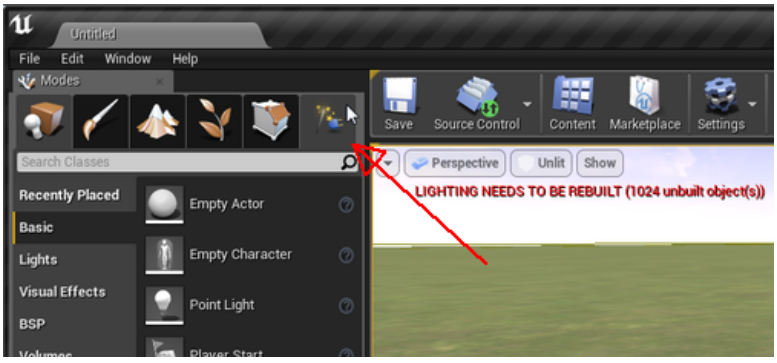


Abbildung A.1.: Editormodus

[1] https://docs.unrealengine.com/latest/INT/GettingStarted/Installation/index.html
[2] https://docs.unrealengine.com/latest/INT/Engine/QuickStart/index.html
[3] https://docs.unrealengine.com/latest/INT/Engine/Blueprints/QuickStart/index.html

3. Generierunglimits bestimmen
   Bevor das Straßennetz generiert werden kann, muss ein Bereich auf dem Terrain abgegrenzt werden, in dem die Stadt entstehen soll.

   Dazu einfach in der Toolbar des Editormodus auf **Begrenzungspunkt hinzufügen** klicken und dann den erstellten Punkt auf dem Terrain positionieren. Es werden mindestens drei Punkte benötigt, allerdings sind vier empfehlenswert, da das Straßennetz sonst auf ein Dreieck begrenzt ist.

4. Hauptstraßen generieren
   Die Hauptstraßen können jetzt einfach durch klicken auf **Hauptstraßen generieren** erstellt werden(siehe Punkt 1 in der Abbildung A.2). Unter Punkt 2 sind die Einstellungen für die Länge der zu erstellenden Straßenstücke, die minimal Länge und der Mindestabstand zu anderen Straßen einstellbar. Unter Punkt 3 sind die Einstellungen zur zufälligen Erstellungen von Zielpunkten. Range Min/Max definiert in welcher Entfernung ein neuer Punkt erzeugt wird.

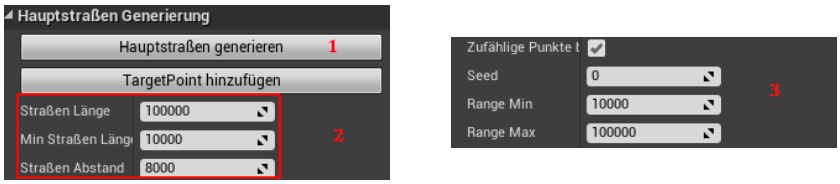


Abbildung A.2.: Hauptstraßenoptionen

5. Nebenstraßen generieren
   Nachdem die Hauptstraßen generiert sind, können die Nebenstraßen durch klicken auf den neuen Button **Nebenstraßen generieren** erstellt werden. Der erste Teil der Einstellung hat die gleichen Optionen. Im zweiten Teil kann gewählt werden, ob das Blockmuster parallel zur längsten Straße des Blockes erzeugt wird oder zu den zwei Richtungsvektoren, die darunter angegeben sind. Des Weiteren kann durch die Auswahl von Blöcken und klicken auf **Nur ausgewählte Blöcke** auch nur für die ausgewählten Blöcke weitere Straßen generiert werden.

6. Kreuzungen und Grundstücke visualisieren
   Kreuzungen und Grundstücke werden nicht automatisch visualisiert, um den Rechenaufwand bei der Generierung gering zu halten. Um sie darzustellen, muss auf den jeweiligen Button in der letzten Spalte geklickt werden.

7. Straßen per Hand anpassen
   Sollte das entstandene Straßennetz noch nicht optimal sein, können Straßen und Kreuzungen verschoben oder gelöscht werden. Außerdem kann durch Auswahl einer Kreuzung und Alt+Linksklick auf das Terrain ein Straßenstück von der ausgewählten Kreuzung ausgehend hinzugefügt werden. Ähnlich können Straßen durch Auswahl und Alt+Linksklick auf die ausgewählte Straße an der Stelle geteilt werden.

A.1.3. Erstellung einer Aufgabe

Um normale Elemente des Editors zu benutzen, muss in den Placement Modus(1.Tab) gewechselten werden.

Wie im Blueprint Quickstart Guide umgesetzten Beispiel, können wir hierfür einen leeren Actor erstellen, ihm ein Mesh zuweisen, in einen Blueprint Actor umwandeln und eine Kollisionserkennung hinzufügen. Einzige Anpassung ist das im Fall einer Kollision eine Nachricht an den EmoBike Message Broker gesendet werden muss.

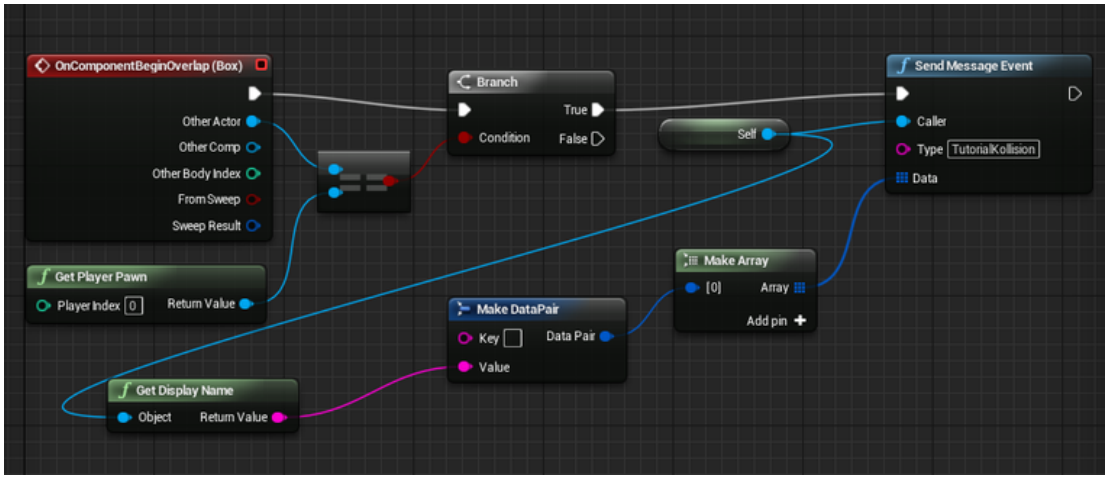


Abbildung A.3.: Beispiel Blueprint

Dies kann einfach in jedem Blueprint durch hinzufügen der Funktion *SendMessageEvent* umgesetzt werden. Für dieses Beispiel erstellen wir eine Nachricht des Typs *TutorialKollision* mit einem Datenfeld, das den Namen des Actors enthält.

A.1.4. Platzierung des Fahrrads

Bevor das Level spielbar ist, muss ein **Player Start** eingefügt werden. Dieser bestimmt wo das Fahrrad am Anfang der Simulation steht. Nach dem Platziren und Ausrichten des Player Starts

kann dieser leicht im Editor getestet werden, indem man auf **Play** klickt. Durch Drücken von M während des Testens kann man das Fahrrad auch ohne EmoBike per WASD Steuern.

A.1.5. Build des Stadtlevels zur Nutzung durch das EmoBike

Nachdem das Level fertiggestellt ist, muss der Build gestartet werden. **Build** ist in der oberen Toolbar auf der rechten Seite zu finden(Der Build sollte auf einem PC mit guten Prozessor und 16+GB RAM gestartet werden).

Als nächstes muss das neue Level in den Packaging Settings zu der Liste der Karten hinzugefügt werden. Diese können unter **File->Package Project ->Packaging Settings** gefunden werden. In dem Settings-Fenster die erweiterten Optionen anzeigen. Unter **List of maps to include in a package build** das Level mit dem + hinzufügen.

Abbildung A.4.: Ausschnitt der Packaging Settings

Als nächstes muss das Projekt gepackt werden(**File->Package Project->Windows->Windows(64-bit)**).

Erstes Packaging: Es müssen noch die externen Bibliotheken(activemq-cpp.dll, libapr-1.dll, libaprutil-1.dll, gen_test_char.exe) aus dem **Projekt/Binaries/Win64** Ordner in den **WindowsNoEditor/BikeSimulator/Binaries/Win64** Ordner in das gepackten Projekt kopiert werden.

Das gepackte Projekt kann nun auf den EmoBike Game PC kopiert werden und gestartet werden.

Literaturverzeichnis

[Apache Software Foundation a] Apache Software Foundation: *Apache ActiveMQ*. – URL http://activemq.apache.org/. – (letzter Zugriff: 05.03.2016)

[Apache Software Foundation b] Apache Software Foundation: *Apache ActiveMQ-CPP*. – URL http://activemq.apache.org/cms/cms-api-overview.html. – (letzter Zugriff: 05.03.2016)

[Barrett 2007–2009] Barrett, Sean: *L-Systems Considered Harmful*. 2007-2009. – URL http://nothings.org/gamedev/l_systems.html. – (letzter Zugriff: 05.03.2016)

[Baumgart 1975] Baumgart, Bruce G.: A polyhedron representation for computer vision. In: *Proceedings of the May 19-22, 1975, national computer conference and exposition* ACM (Veranst.), 1975, S. 589–596

[Eberly 2014] Eberly, David: *The Minimal Cycle Basis for a Planar Graph*. 2014. – URL http://www.geometrictools.com/Documentation/MinimalCycleBasis.pdf. – (letzter Zugriff: 20.11.2015)

[Epic Games 2016] Epic Games: *what is unreal engine 4*. 2016. – URL https://www.unrealengine.com/what-is-unreal-engine-4. – (letzter Zugriff: 14.03.2016)

[Epic Games Blueprints] Epic Games: *Blueprints Visual Scripting*. Blueprints. – URL https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html. – (letzter Zugriff: 05.03.2016)

[Epic Games Editor Mode] Epic Games: *FEdMode*. Editor Mode. – URL https://docs.unrealengine.com/latest/INT/API/Editor/UnrealEd/FEdMode/index.html. – (letzter Zugriff: 05.03.2016)

[Epic Games Landscape Editor] Epic Games: *Landscape Outdoor Terrain*. Landscape Editor. – URL https://docs.unrealengine.com/latest/INT/Engine/Landscape/index.html. – (letzter Zugriff: 05.03.2016)

[Epic Games Physics Simulation] Epic Games: *Physics Simulation*. Physics Simulation. – URL https://docs.unrealengine.com/latest/INT/Engine/Physics/index.html. – (letzter Zugriff: 05.03.2016)

[Epic Games Slate] Epic Games: *Slate UI Framework*. Slate. – URL https://docs.unrealengine.com/latest/INT/Programming/Slate/index.html. – (letzter Zugriff: 23.03.2016)

[Epic Games Splines] Epic Games: *Splines Content Examples*. Splines. – URL https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/Blueprint_Splines/#constructionscriptexamples. – (letzter Zugriff: 05.03.2016)

[Google 2016] Google: *Google Maps*. 2016. – URL https://www.google.com/maps. – (letzter Zugriff: 01.04.2016)

[Hornschuh 2015] Hornschuh, Jonas: Weiterentwicklung eines Fahrradergometers als intuitive Steuerung für virtuelleWelten. (2015)

[Kelly und McCabe 2007] Kelly, George ; McCabe, Hugh: Citygen: An interactive system for procedural city generation. In: *Fifth International Conference on Game Design and Technology*, 2007, S. 8–16

[Lindenmayer 1968] Lindenmayer, Aristid: Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. In: *Journal of theoretical biology* 18 (1968), Nr. 3, S. 300–315

[NVIDIA PhysX SDK 3.3.1] NVIDIA: *NVIDIA PhysX SDK 3.3.1 Documentation*. PhysX SDK 3.3.1. – URL http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Index.html. – (letzter Zugriff: 05.03.2016)

[NVIDIA Vehicles] NVIDIA: *Vehicles Documentation*. Vehicles. – URL http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/Vehicles.html. – (letzter Zugriff: 05.03.2016)

[Parish und Müller 2001] Parish, Yoav I. ; Müller, Pascal: Procedural modeling of cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* ACM (Veranst.), 2001, S. 301–308

[Prusinkiewicz und Lindenmayer 2012] Prusinkiewicz, Przemyslaw ; Lindenmayer, Aristid: *The algorithmic beauty of plants*. Springer Science & Business Media, 2012

[Shikin und Plis 1995] Shikin, Eugene V. ; Plis, Alexander I.: *Handbook on Splines for the User.* CRC Press, 1995

[Toussaint 1983] Toussaint, Godfried T.: Solving geometric problems with the rotating calipers. In: *Proc. IEEE Melecon* Bd. 83, 1983, S. A10

[Vanegas u. a. 2012] Vanegas, Carlos A. ; Kelly, Tom ; Weber, Basil ; Halatsch, Jan ; Aliaga, Daniel G. ; Müller, Pascal: Procedural generation of parcels in urban modeling. In: *Computer graphics forum* Bd. 31 Wiley Online Library (Veranst.), 2012, S. 681–690

Abbildungsverzeichnis

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 6. April 2016 Alexander Döring