UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Prozedurale Generierung von 3D-Stadtmodellen

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Simeon Hermann

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Bastian Krayer, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Oktober 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Zusammenfassung

Die folgende Bachelorarbeit gibt einen Überblick über verschiedene Ansätze und Verfahren zur prozeduralen Generierung von dreidimensionalen Stadtmodellen. Dabei wird vor allem die Nutzung generativer Grammatiken näher untersucht und in einer eigens implementierten Anwendung integriert. Der Schwerpunkt war es, ein vorgegebenes, primäres Straßennetz einzubinden und darauffolgend ein sekundäres Straßennetz sowie verschiedene Gebäude prozedural zu generieren. Die Anwendung ermöglicht es, umfangreiche und unterschiedlich strukturierte Stadtmodelle auf effiziente Weise zu erzeugen. Hinsichtlich des Realismus und Variantenreichtums weisen die Ergebnisse jedoch Grenzen auf.

Abstract

The following bachelor thesis gives an overview of various approaches and techniques for procedural generation of three-dimensional city models. Especially the usage of generative grammars is being examined and later used for the implementation of an own application. Its focus was the embedding of predetermined primary street networks as well as the procedural generation of secondary street networks and different kinds of buildings. The application allows the efficient creation of extensive and variably structured city models. However, there are restrictions regarding the realism and variation of the results.

1 Einleitung

1.1 Motivation

In einer Zeit, in der mit steigender Tendenz bereits mehr als die Hälfte aller Menschen in Städten lebt, ist es nicht verwunderlich, dass deren Visualisierung in der Computergrafik ein großes Thema ist. Nachbildungen realer oder fiktiver Städte sind heute in verschiedensten Nutzungsszenarien vonnöten. In der Städteplanung ist es von Bedeutung, Bauprojekte visuell zu veranschaulichen. Navigationsgeräte und Kartendienste verwenden ebenfalls dreidimensionale Stadtmodelle, um die Orientierung für den Nutzer zu verbessern. Zu diesem Zweck müssen vollständige Kartensätze gelesen und als 3D-Modell integriert werden.
Gerade aber auch innerhalb der Unterhaltungsindustrie werden virtuelle Städte genutzt. Realfilme nutzen regelmäßig computergenerierte Städte als Kulisse. So besitzt man alle Freiheiten hinsichtlich der Beleuchtung und Effekte. Auch Kamerafahrten, die in Wirklichkeit nahezu unmöglich wären, lassen sich so darstellen.
Spätestens in der Videospielindustrie sind computergenerierte Städte seit vielen Jahren unerlässlich. Die Spielreihe *SimCity* beispielsweise verbindet seit 1989 Spiel sowie Simulation in einem und lässt den Spieler eigene Städte kreieren. Von zweidimensionalen, planaren Flächen mit gerade verlaufenden Straßen und repetitiven Gebäude hin zu dynamischen Straßenverläufen, formbarem Terrain und unzähligen Details hat sich der Anspruch im Laufe der Reihe deutlich erhöht (siehe Abb. 1).

Abbildung 1: SimCity (1989) [1], SimCity2000 (1993) [2], SimCity (2013) [3] (v.l.n.r.)

Zusätzlich sind Städte in vielen Spielen nicht nur eine Kulisse, die oberflächlich betrachtet wird. Meistens fungieren sie als Schauplätze, die von Spielern bis ins kleinste Detail erkundet werden. Modelliert man Städte in der nötigen Größe noch von Grund auf manuell, steht der Aufwand kaum noch in Relation zum Nutzen. Kleinere Entwicklerteams müssen sich ebenfalls an dieser Messlatte orientieren, können diesen Ansprüchen aber kaum noch gerecht werden.

Ob Modelle realer oder fiktiver Städte, es ist heute kaum noch möglich,

virtuelle Städte von Hand zu modellieren und dabei gleichzeitig den qualitativen und zeitlichen Anforderungen in der jeweiligen Branche gerecht zu werden.
Aus diesem Grund ist es keine Seltenheit mehr, dass Städte bis zu einem gewissen Grad oder vollständig durch den Computer generiert werden. Diese Methodik wird auch prozedurale Generierung genannt und findet in den verschiedensten Bereichen der Computergrafik Verwendung. Sie lässt sich sowohl auf die Erzeugung fiktionaler Städte, als auch der Modellierung real existierender Städte anwenden.

1.2 Zielsetzung

Ziel dieser Bachelorarbeit ist es, einen Einblick in die prozedurale Generierung von Städten zu gewähren. In diesem Rahmen werden in Kapitel 2 verschiedene Anwendungen vorgestellt und deren unterschiedlichen Ansätze und Schwerpunkte hervorgehoben. Als Grundlage einiger bekannter Anwendungen werden zudem die in der prozeduralen Generierung häufig anzutreffenden L-Systeme erläutert.
Mit diesem Wissen wurde eine Anwendung entwickelt, die es ermöglicht, prozedural generierte, dreidimensionale Städte zu erzeugen. Dabei besitzt der Nutzer hinsichtlich der groben Stadtstruktur einen bestimmten Grad an Kontrolle. In Kapitel 3 werden die bei der Implementierung dieser Anwendung ausgewählten Schritte und Algorithmen für die Generierung der wichtigsten Elemente einer Stadt veranschaulicht.
Kapitel 4 befasst sich mit der Bewertung der Anwendung anhand einer Auswahl an Kriterien und evaluiert die Ergebnisse vor dem Hintergrund des gewählten Schwerpunktes mit den heutigen Ansprüchen. Mögliche Erweiterungen oder Überarbeitungen werden ebenfalls thematisiert.

2 Grundlagen

2.1 Prozedurale Generierung

2.1.1 Definition

Mit dem Begriff Prozedurale Generierung, oft auch Prozedurale Synthese genannt, bezeichnet man in der Computergrafik ein Konzept, mit dem man mit nur wenigen Eingangsinformationen automatisch 3D-Objekte oder Texturen generieren lassen kann. Die Inhalte werden also nicht als statische Objekte vom Entwickler bis ins letzte Detail kreiert, sondern ausgehend von definierten Vorgaben und Regeln vom Computer erzeugt. Auf diese Weise können zeitsparend große Mengen an komplexen und variantenreichen Inhalten erstellt werden. Die Erzeugnisse basieren dabei auf deterministischen Algorithmen, sodass die Inhalte reproduzierbar sind. Es gibt also eine klare Abgrenzung gegenüber der Zufallsgenerierung. [4]

Zentrale Eigenschaften prozeduraler Verfahren sind zum einen die **Abstraktion** sowie die **parametrische Kontrolle** [5]. Durch die Abstraktion werden die geometrischen Daten nicht direkt spezifiziert, sondern in einem Algorithmus oder einer Kette an Instruktionen abstrahiert. Da man lediglich die implizite Spezifizierung, die zu den expliziten Ergebnissen führt, benötigt, spart man zudem Speicherplatz. Dieser Vorteil ist vor allem vor dem Hintergrund der parametrischen Kontrolle praktisch, da die Daten auf diese Weise flexibel verändert werden können und verschiedene Variationen erzeugen, ohne dass jede Variante separat gespeichert werden muss.

2.1.2 Anwendungsbereiche

Zu Beginn beschränkten sich die Anwendungsfälle prozeduraler Generierung primär auf natürliche Phänomene. Vor allem bei der Erzeugung von Texturen wurden und werden prozedurale Verfahren genutzt. Beispielsweise sorgt die Verwendung von Rauschfunktionen wie Perlin Noise für eine realistischere Gestaltung bestehender Texturen. Weiterhin wurden zellulare Strukturen von Haut, Marmor oder Pflasterstein durch die Nutzung von Voronoi Diagrammen in Texturen integriert.
Erst später wagte man sich auch an die prozedurale Generierung komplexer Geometrien. Anfangs erzeugte man vor allem Vegetation wie Bäume oder Pflanzen, später auch Terrains oder Seen. Somit ging es weiter bis zur prozeduralen Generierung ganzer Welten.
Bis dahin konzentrierte man sich wie bereits erwähnt auf natürliche und weniger auf menschengemachte Phänomene. Erst das Wachstum der Unterhaltungs- und Videospielbranche sorgte schließlich auch für ein gesteigertes Interesse an der effizienten Generierung von Städten und allen damit

einhergehenden Bestandteilen wie Straßen und Gebäude. [6]

2.2 Prozedurale Verfahren

2.2.1 L-Systeme

Lindenmayer-Systeme, kurz L-Systeme, sind eines der bekanntesten prozeduralen Verfahren. Sie stammen von Aristid Lindenmayer, nach dem die L-Systeme benannt wurden. Der ungarische Biologe erfand sie, um das Nachbarschaftsverhalten zwischen Pflanzenzellen und deren Entwicklung beschreiben zu können. [7]

Das wichtigste Prinzip der L-Systeme sind Produktionsregeln, die das Überschreiben und Ersetzen eines Strings durch einen anderen String festlegen. L-Systeme sind damit im Prinzip formale Grammatiken. Eine Besonderheit ist, dass bei L-Systemen alle Produktionsregeln parallel statt sequenziell ausgeführt werden.
Folgende Komponenten sind dabei die Bestandteile eines L-Systems:

- V, die Menge an Variablen, die durch Produktionsregeln ersetzt werden können
- S, die Menge an Symbolen, die nicht ersetzt werden können
- ω, die Menge an Anfangszuständen
- P, die Menge an Produktionsregeln, die definieren, wie eine Variable ersetzt wird

Ein einfaches Beispiel eines L-Systems ist die Bildung eines Wortes aus der Morsefolge:

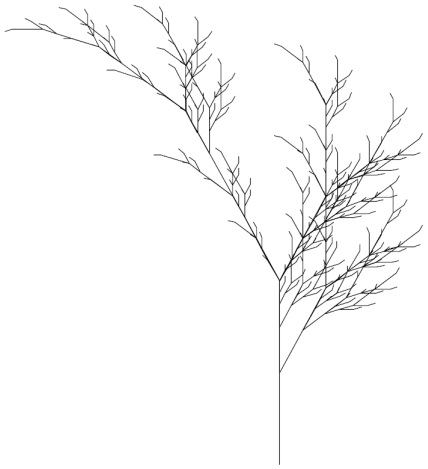
$$V = \{0,1\} \qquad \omega = 0 \qquad P_1: 0 \rightarrow 01 \qquad P_2: 1 \rightarrow 10$$

Die Menge der Variablen V besteht aus den Strings 0 und 1. Symbole S gibt es nicht, weshalb der finale Ausdruck allein an die Anzahl an Iterationen n geknüpft ist. Das Ergebnis nach den jeweiligen Iterationen sieht wie folgt aus:

$$\omega: 0 \qquad n = 1: 01 \qquad n = 2: 0110 \qquad n = 3: 01101001$$

Besonders Fraktale werden häufig durch L-Systeme beschrieben. Fraktale sind geometrische Objekte, die innerhalb ihrer Form einen Grad an Selbstähnlichkeit besitzen bzw. kleine Kopien von sich selbst beinhalten. In der Realität weisen insbesondere Pflanzen diese Eigenschaften oft innerhalb ihrer Verzweigung auf. Daher fanden L-Systeme gerade in diesem Bereich regelmäßig Verwendung. Die Symbole besitzen in diesen Fällen meistens eine geometrische Bedeutung, wie einen Schritt nach vorne oder

eine Abzweigung. Abb. 2 veranschaulicht eine derartige Beschreibung des Verzweigungsverhaltens einer Pflanze durch ein L-System mit einer Turtle-Grafik.



V = { X, F, +, -, [,] }
ω= X
P_1: X -> F - [[X] + X] + F[+ FX] - X
P_2: F -> FF

F: einen Schritt vorwärts
+: um α nach links drehen
-: um α nach rechts drehen
[: derzeitigen Stand auf Stack legen
]: zu Stand auf Stack zurückkehren

α: 22,5 °
n: 5

Abbildung 2: durch ein L-System erzeugtes Pflanzenwachstum [8]

2.3 Struktur einer Stadt

Mögliche Bereiche der prozeduralen Generierung von Städten sind quasi grenzenlos, da eine Stadt als menschenerschaffendes Phänomen unzählige Details beinhaltet. Noch mehr als ländliche Areale leben Städte nicht allein vom oberflächlichen Eindruck, sondern auch sehr stark von eben dieser großen Menge kleiner Details. Um sich nicht in diesen Details zu verlieren und eine Basis für die Stadtgenerierung zu schaffen, muss man sich auf die elementaren und prägenden Elemente für die Stadtstruktur fokussieren, auf denen alle restlichen Details aufbauen.

Bei einem Blick auf eine Stadtkarte fällt natürlich primär das Verkehrsnetz auf. Darunter fallen vor allem Straßen, aber auch Bahnlinien oder Wasserstraßen. Nicht nur für den strukturellen Blick von oben, sondern auch für den persönlichen Eindruck innerhalb einer Stadt spielen Straßen eine maßgebliche Rolle. Unabhängig vom Fortbewegungsmittel bilden diese in dicht besiedelten Städten den Hauptorientierungspunkt dafür, wo man sich befindet und wohin man will. Große Straßen bilden die Haupttransportwege, die jeden Tag von Zulieferern genutzt werden. Auch wenn Straßennetze heutzutage fast überall bis zu einem gewissen Grad ausgebaut sind, ist deren Relevanz innerhalb einer Stadt aufgrund der hohen Bebauungsdichte besonders hoch.
Während das Verkehrsnetz den größten Einfluss auf die Struktur der Stadt

hat, ist deren Bebauung letztlich wohl das wichtigste Element, um wirklich den Eindruck einer Stadt zu erzeugen. Dabei ist die Bebauung keineswegs homogen. In jeder Stadt gibt es verschiedene Areale mit unterschiedlichen Bebauungstypen. Das soll heißen, es lassen sich beispielsweise Geschäftsviertel, Wohnbezirke oder Industriegebiete, die sich optisch stark voneinander unterscheiden, vorfinden.

Viele der existierenden Anwendungen zur Städtegenerierung basieren vor diesem Hintergrund auf den Ausführungen von Kevin Lynch zum Verkehrsnetz einer Stadt [9] und Christopher Alexander zu Nachbarschaften [10].
Lynch unterteilte das primäre Verkehrsnetz in die Elemente Pfade und Knoten. Pfade bestehen aus Straßen und Wegen innerhalb der Stadt. Knoten enstehen vor allem durch Kreuzungen von Pfaden, an welchen Richtungswechsel eingeschlagen werden können.
Alexander betonte die Wichtigkeit von Nachbarschaften in einer Stadt. Jeder dieser zellenartigen Bereiche habe einen wiedererkennbaren Charakter, der sich von den anderen Nachbarschaften abhebt. Getrennt werden diese von visuell greifbaren Grenzen, sowohl natürlicher Art wie Flüsse oder Seen als auch menschengemachter Art wie Hauptstraßen oder Bahnlinien. Im zweiten Fall dienen also die von Lynch beschriebenen Pfade als Grenzen bzw. Kanten.
Fasst man diese Konzepte zusammen, sind die wichtigsten Elemente der Stadtstruktur Pfade des Verkehrsnetzes, daraus resultierende Knoten sowie Pfade, die als Kanten Nachbarschaften eingrenzen.

2.4 Prozedurale Verfahren für Städte

Es gibt sehr unterschiedliche Ansätze für die prozedurale Erzeugung von Städten. Häufig beruhen diese Unterschiede auf dem Anwendungszweck oder den gewählten Schwerpunkten bei der Entwicklung.
Die Methoden der prozeduralen Generierung von Städten lassen sich unter anderem in folgende Kategorien aufteilen [11]:
Generative Grammatiken, die mit Hilfe von Produktionsregeln die Erzeugung von Geometrien steuern. *Simulations-basierte* Anwendungen sollen durch die Simulation der Stadtentwicklung plausible Ergebnisse erzeugen. *Daten-gesteuerte* Anwendungen integrieren vorgegebene, geographische Datensätze und erzeugen daraus eine den Vorgaben entsprechende Stadt.
Die folgende Auflistung verwandter Anwendungen beinhaltet Beispiele für diese Kategorien und gibt einen Einblick in die verschiedenen Ansätze prozeduraler Städtegenerierung.

2.4.1 Undiscovered City

Die als *Undiscovered City* bezeichnete Anwendung für die Real-Time-Generierung dreidimensionaler Städte wurde 2003 von Stefan Greuter an der RMIT University in Melbourne konzipiert und entwickelt. Ihr Konzept unterliegt dabei hauptsächlich der Idee, ein einfaches Grid-Layout zu nutzen, auf welchem simple Gebäude platziert werden.

Auf der einen Seite gibt es also das Straßennetzwerk, welches im Stil einer Planstadt wie New York City schachbrettmusterartig strukturiert ist. Aus diesem Muster entstehen rechteckige Blöcke, deren Größe konstant, jedoch auch global anpassbar ist.

Abbildung 3: Stadt aus Undiscovered City [12]

In der Demo-Erweiterung Neverland wurde genau das Prinzip der Uniformität so abgeändert, dass auch ein irreguläres Grid mit verschiedenen Block-Größen und -Formen möglich ist. So erscheint die Stadt abwechslungsreicher und macht einen weniger generischen Eindruck.

Die Gebäude werden abhängig vom Grid-Layout in den entstehenden Blöcken platziert. Die Grid-Koordinaten werden als Seed für die Generierung genutzt. Daraus werden verschiedene Attributwerte wie die Höhe oder Breite des Gebäudes sowie die Anzahl an Stockwerken gesetzt.
Um nicht nur rechteckige Gebäude zu erzeugen, setzt sich deren Geometrie aus der Kombination mehrerer einfacher, geometrischer Primitive zusammen. Dafür wird das Gebäude in mehrere Abschnitte unterteilt. Von oben ausgehend fängt man an, eine einfache geometrische Form für einen der Abschnitte zu definieren. Schrittweise arbeitet man sich daraufhin nach unten und ergänzt die bereits bestehende Geometrie um eine weitere Geometrie, sodass für diesen Abschnitt eine neue, komplexere geometrische Form entsteht. Das Gebäude breitet sich so bis nach unten hin aus. [12]

2.4.2 CityEngine

Die Anwendung *CityEngine* verwendet L-Systeme für die prozedurale Generierung von Städten. Sie wurde 2001 von Parish und Müller in „Procedural Modeling of Cities“ [13] näher beschrieben. Ausgehend von geographischen und statistischen Eingangsdaten können mit dieser Engine Straßen generiert, aber auch detaillierte Gebäude sowie Fassaden konstruiert werden.

Dabei nutzt die *CityEngine* für die Straßengenerierung eine erweiterte Form eines L-Systems, auch selbst-sensitiv genannt. Dieses kreiert das Straßennetzwerk unter Berücksichtigung des bereits enstandenen Wachstums. Der Input besteht aus 2D-Bildkarten, vor allem geographischer Natur wie der Höhe, der Vegetation oder Wasser. Aber auch statistische Informationen wie Bevölkerungsdichte, Stadtzonen, Gebäudehöhen oder die Stadtmorphologie können integriert werden. [12]
Aufgrund der Nutzung eines L-Systems kann die Methodik der *CityEngine* als *generative Grammatik* eingeordnet werden. Durch die Integration von existierenden oder vorgefertigten Kartendaten arbeitet sie jedoch auch *daten-basiert*.

Straßengenerierung

Das Wachstum der Straßen kann man sich ähnlich wie das Pflanzenwachstum aus Abb. 2 als eine von einem Punkt ausgehende Folge an Abzweigungen und Schritten nach vorne vorstellen. Jeder Schritt nach vorne bildet dabei ein neues Straßensegment.
Der Kern des L-Systems stellt als nächstfolgendes Straßensegment immer nur einen Vorschlag auf, auch *ideal successor* genannt. Er dient zunächst quasi als Platzhalter, bei dem weder Parameter noch die Baubarkeit berücksichtigt wurden. Einzelheiten, welche die letztendliche Platzierung des Straßensegments beeinflussen, werden in separate Funktionen ausgelagert. In diesen können neue Parameter und Bedingungen ergänzt werden, ohne die Struktur des L-Systems zu verändern.
Eine dieser Funktionen ist die *globalGoals*-Funktion. Diese setzt Parameter, die für alle neuen Straßensegmente in einer Region gelten. Das können unter anderem die Häufigkeit von Abzweigungen oder die Länge der Straßen sein. Sie werden beispielsweise durch gewählte *road patterns* wie die Schachbrettstruktur oder radiales Wachstum beeinflusst.
Durch die *localConstraints*-Funktion wird geprüft, ob ein Straßensegment an der vorgeschlagenen Stelle platziert werden darf. Nicht bebaubare Flächen wie Wasser oder die Nähe zu anderen Straßen sorgen dafür, dass das Segment nicht gebaut wird oder dessen Parameter angepasst werden. Im

zweiten Fall sorgt diese unter anderem dafür, dass sich die naheliegenden Straßen ab einem in der *globalGoals*-Funktion gesetzten Schwellwert verbinden oder kreuzen. Eben diese Berücksichtigung des eigenen (Straßen)wachstums ist der Grund dafür, weshalb das L-System auch als selbstsensitiv bezeichnet wird.

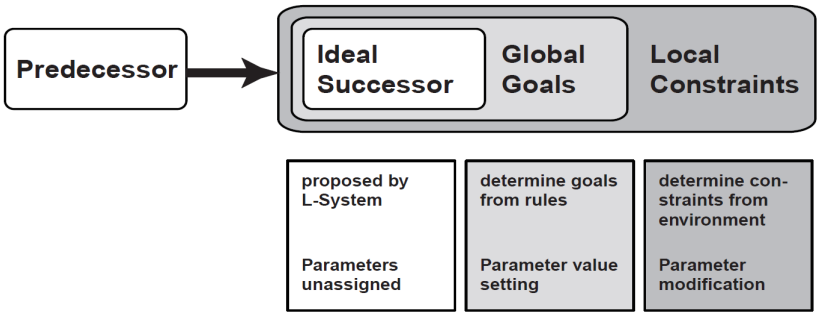


Abbildung 4: Ablauf bei der Bestimmung des nachfolgenden Straßensegments [13]

Besonderheiten des in Abb. 5 dargestellten L-Systems sind zum einen die Kontextsensitivität. Das bedeutet, dass die Variablen links und rechts von der zu überschreibenden Variable im String ebenfalls die Wahl der Produktionsregel beeinflussen können.
Zugleich ist es auch ein parametrisches L-System, da die Variablen eigene Attribute besitzen. Diese werden in den ausgelagerten Funktionen gesetzt. Im Falle von *delay* und *state* entscheiden diese Attribute sogar über die Wahl der Produktionsregel.
Ohne auf alle Einzelheiten einzugehen, lassen sich folgende Abläufe aus den Produktionsregeln des L-Systems festhalten:

- Für die Stellen, an denen das Wachstum beginnt, existiert ein Anfangszustand mit den initialen Attributen. (ω)
- Wird ein neues Straßensegment R vorgeschlagen, folgt in jedem Fall die Abfrage I, welche prüft, ob R platziert werden kann. Das Ergebnis der Abfrage wird im Attribut *state* zurückgegeben. (ω, p_2, p_5)
- Gibt die Abfrage I FAILED als *state* zurück, kann das Straßensegment nicht gebaut werden. (p_3)
- Gibt die Abfrage I SUCCEED als *state* zurück, wird das Straßensegment gebaut. Zusätzlich werden zwei abzweigende Branchsegmente und ein gerade weiterführendes Straßensegment vorgeschlagen. (p_2)
- Ist das Attribut *delay* eines Straßen- oder Branchsegments negativ, soll die Straße in diese Richtung nicht mehr fortgeführt werden. (p_1, p_6)

- Ist *delay* == 0 wird aus dem Branchsegment ein Straßensegment. Dieses kann in der nächsten Iteration gebaut werden. Ist das *delay* eines Branch > 0 wird ein neues Branchsegment mit um 1 dekrementiertem *delay* erstellt. Das *delay* legt daher fest, wie häufig eine Abzweigung auftritt. Ein *delay* von 3 würde bedeuten, dass sich die Straße erst nach drei weiteren Straßensegmenten abzweigt. (p_4)

- Die Veränderung bzw. Weitergabe der Parameter wird in den ausgelagerten Funktionen *globalGoals* (p_2) und *localConstraints* (p_8) realisiert

Variablen
R()...Straßensegment B()...Abzweigung ?I()...Abfrage localConstraints

Symbole
ε...nichts wird gebaut +()F()...Platzieren eines Straßensegments

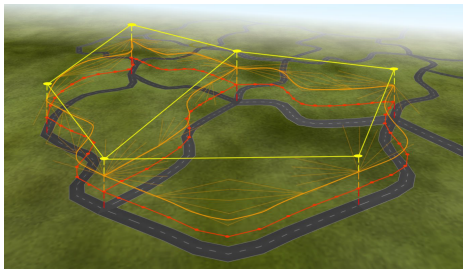
Axiome
ω= R(0, initialRuleAttr) ?I(initRoadAttr, UNASSIGNED)

Produktionsregeln
p_1: R(del, ruleAttr) : del<0 -> ε
p_2: R(del, ruleAttr) > ?I(roadAttr,state) : state==SUCCEED
{globalGoals(ruleAttr,roadAttr) creates parameters for: pDel[0-2], pRuleAttr[0-2], pRoadAttr[0-2]} -> +(roadAttr.angle)F(roadAttr.length)
B(pDel[1],pRuleAttr[1],pRoadAttr[1]), B(pDel[2],pRuleAttr[2],pRoadAttr[2]), R(pDel[0],pRuleAttr[0]) ?I(pRoadAttr[0],UNASSIGNED)
p_3: R(del, ruleAttr) > ?I(roadAttr, state) : state==FAILED -> ε
p_4: B(del, ruleAttr, roadAttr) : del>0 -> B(del-1, ruleAttr, roadAttr)
p_5: B(del, ruleAttr, roadAttr) : del==0 -> [R(del, ruleAttr)?I(roadAttr, UNASSIGNED)]
p_6: B(del,ruleAttr,roadAttr) : del<0 -> ε
p_7: R(del,ruleAttr) < ?I(roadAttr,state) : del<0 -> ε
p_8: ?I(roadAttr,state) : state==UNASSIGNED localConstraints(roadAttr) adjusts the parameters for: state, roadAttr -> ?I(roadAttr, state)
p_9: ?I(roadAttr,state) : state!=UNASSIGNED -> ε

id: linker Kontext < Vorgänger > rechter Kontext : Bedingung -> Nachfolger [14]

Abbildung 5: L-System von Parish und Müller [13]

Wurde das Straßennetz vollständig erzeugt, werden entstandene Häuserblocks in Grundstücke aufgeteilt. Deren Form dient als Grundlage für den Gebäudeumriss und ist somit der Anfangszustand für die Gebäudegenerierung, die ebenfalls durch ein L-System beschrieben wird.
Die dazugehörigen Produktionsregeln verfeinern die durch den Umriss gegebene Grundform des Gebäudes durch Skalierung, Translation und leicht abgeänderte Teilgeometrien.
Die Fassadentexturierung der Gebäude wird durch das Übereinanderlegen verschiedener Layer auf eine halbautomatische Weise umgesetzt. Ver-



(a) Knoten und Kanten des primären Straßennetzes

(b) Wachstum des sekundäres Straßennetzes in einer Region

Abbildung 6: Straßennetzgenerierung aus *CityGen* [6]

knüpft man diese Layer logisch miteinander, kann man beispielsweise verschiedene Ergebnisse zur Lage und Größe der Fenster oder Türen erhalten.

2.4.3 CityGen

Bei der Entwicklung der Anwendung *CityGen* war das Ziel die Erstellung und Manipulation von Städten in Echtzeit. Sie wurde von George Kelly und Hugh McCabe entwickelt und im Jahre 2007 vorgestellt [15].
Sie lässst sich in die Kategorie *generative Grammatik* zuordnen und orientiert sich in vielen Aspekten an der *CityEngine*. Bei ihrer Nutzung werden jedoch keine Realdaten integriert. Stattdessen wurde die dynamische Manipulation und Anpassung der Stadt in Echtzeit ergänzt.
CityGen lässt sich in drei Schritte aufteilen: die Generierung der Primärstraßen, der Sekundärstraßen und der Gebäude.

Die Anwendung gibt ausgehend von einem vorgebenem Straßenmuster (z.B. Radial oder Schachbrett) ein primäres Straßennetzwerk vor. Dieses wird durch Knoten als Kontrollpunkte beschrieben (vgl. Abb. 6a). Zwischen den Knoten verlaufen an das Terrain angepasste Primärstraßen. Die vorliegenden Knotenpunkte können jedoch bewegt werden und auch das Hinzufügen oder Entfernen von Knoten ist möglich. Die Primärstraßen passen sich dabei in Echtzeit an die Manipulationen an. Eine Bedingung dabei ist, dass eine Menge an Kanten zu jeder Zeit eine von Primärstraßen umschlossene Region bildet.
Sekundärstraßen sind schmalere Straßen, welche in die umschlossenen Regionen hineinwachsen (vgl. Abb. 6b). Die Generierung dieser wird prozedural durch paralleles Wachstum ermöglicht. Es orientiert sich dabei ebenfalls an einem ausgewählten Straßenmuster. Genau wie das primäre Straßennetz reagiert das sekundäre Straßennetz auf Veränderung der Kontrollknoten.
Für die Generierung der Gebäude werden die durch Sekundärstraßen ein-

geschlossenen Regionen berechnet und in Grundstücke aufgeteilt. Auf diesen werden schließlich die Gebäude platziert.

2.4.4 CityBuilder

Die *CityBuilder*-Engine ist eine Agent Based Simulation und konzentriert sich weniger auf die tatsächliche Modellierung einer Stadt. Stattdessen liegt ihr Hauptaugenmerk darauf, die Struktur einer Stadt durch die Simulation des Stadtwachstums realistisch zu erfassen. Die *CityBuilder*-Engine ist daher ein Beispiel für einen *simulations-basierten* Städtegenerator.

Das Straßennetz entwickelt sich grundlegend auf einem uniformen Grid, kann sich jedoch durch einen Abweichungsparameter auch organischer entwickeln. Die Straßendichte und der Abstand zwischen Abzweigungen bzw. Kreuzungen kann beispielsweise verändert werden.
Die Entwicklung des Straßensystems erfolgt durch zwei „Agententypen". Diese haben bestimmte Ziele, die bei dem Bau eines neuen Straßensegments von Bedeutung sind. Extender suchen nach Flächen, die noch nicht vom Straßennetz abgedeckt sind. Connector suchen innerhalb des bestehenden Straßennetzes nach möglichen Verbindungsstraßen, die zwei Punkte oder Areale schneller untereinander anbinden als bisher.
Die Gebäudegenerierung befasst sich lediglich mit der Landnutzung durch die verschiedenen Stadtzonen Wohngebiet, Industriegebiet und Geschäftsviertel. Diese werden durch je einen Agenten repräsentiert, der versucht nach den Präferenzen der jeweiligen Stadtzone zu expandieren. Die Vorschläge werden evaluiert und im Fall, dass sie eine positive Auswirkung auf die Bevölkerung haben oder den Wert des Landes erhöhen, genehmigt. Die finale Visualisierung ist kein Teil des Systems, sondern wurde extern in der *SimCity* Game Engine realisiert. [12]

2.4.5 Template Based Generation

Das Konzept der template-basierten Straßennetzwerkgenerierung wurde 2002 von Sun, Baicu u.a. entworfen.
Zu Beginn stehen verschiedene Templates zur Verfügung, welche die ideale Ausbreitung des Straßennetzes vorgeben. Als Input benötigt man ein Satellitenbild und eine Höhenkarte sowie im Falle eines bevölkerungsbasierten Templates eine Karte mit der Bevölkerungsdichte. Daher lässt sich die Methodik dieses Verfahrens als *daten-basiert* kategorisieren.

Die Ausbreitung des Straßennetzes orientiert sich bei diesem Konzept so weit wie möglich an dem gewählten Template. Im radialen Template breiten sich Straßen geradlinig von innen nach außen hin aus und bilden in bestimmten Intervallen ringartige Verbindungen zwischen den ge-

raden Straßen. Im Raster-Template gilt eine einfache Schachbrettstruktur als Idealentwicklung. Beim bevölkerungsbasierten Template werden die Extremstellen der Bevölkerungsdichte für ein Voronoi-Diagramm hergezogen. Die entstehenden Grenzen im Diagramm dienen dann als Verbindungsstraßen.
Während die Templates bei der Entwicklung des Straßennetzes also immer als Orientierung dienen, gibt es dennoch lokale Einschränkungen, die von den Satelliten- und Höhenkarte ausgehen. Flächen, an denen sich Wasser befindet, müssen umgangen werden und Wege mit großer Steigung werden ebenso gemieden. Das Straßennetz entwickelt sich so letztlich organischer und weicht von der idealen Struktur des Templates ab.
Ein Konzept für die Generierung von Gebäuden ist nicht enthalten. [12]

3 Implementierung

3.1 Konzept

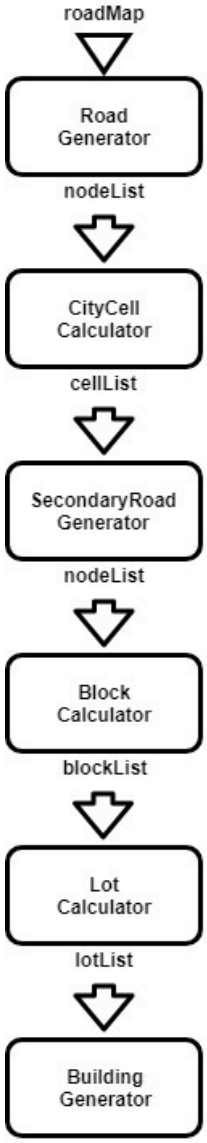
Das Ziel dieser Bachelorarbeit war die Entwicklung einer Anwendung, die ein dreidimensionales Stadtmodell ausgehend von einer einfachen Straßenkarte erzeugt. Der Fokus dieser Anwendung lag aus diesem Grund bei der Integration des Inputs sowie insbesondere der prozeduralen Generierung der Straßen und Gebäude. Diese beiden Elemente sind die Kernbestandteile, um den Eindruck einer Stadt zu erzeugen.
Es soll möglich sein, einfache Entscheidungen zur Struktur und dem Aufbau im Vorhinein treffen zu können. Der Rest der Entwicklung soll jedoch prozedural generiert werden. Durch die Wahl von sogenannten Seeds für Straßen, Grundstücke und Gebäude ist es möglich, die Ergebnisse reproduzierbar abzuändern.
Geht man nach den in Kapitel 2.4 vorgestellten Kategorien der Generierung, lässt sich die Methodik der Anwendung vor allem in die Klasse *generative Grammatik*, zu einem geringen Teil allerdings auch in *daten-orientiert* einordnen. Der Anspruch einer *Simulation* wurde nicht gesetzt.

3.2 Rahmen der Umsetzung

Die Entwicklung der Anwendung wurde in der Spiel-Engine *Unity* und mit der Programmiersprache *C#* umgesetzt.
Die Vorteile in der Nutzung von *Unity* lagen beispielsweise darin, dass einfache Objekte direkt in der Engine erzeugt und gespeichert werden können. Verschiedene Typen von Häusern oder Straßen sind auf diese Weise schnell kreiert und werden direkt in einem beleuchteten, dreidimensionalen Raum dargestellt.
Die Logik der prozeduralen Generierung wird in den *C#*-Scripts definiert. Diese Skripte können Objekten in der Welt angehangen und optional bei deren Erzeugung ausgeführt werden. Skripte können sich auch untereinander Daten übergeben. Die Schritte der in Kapitel 3.3 beschriebenen Pipeline lassen sich auf diese Weise in einzelne Skripte aufspalten, welche nach Abschluss der eigenen Aufgaben den nächsten Schritt initiieren.
Ein weiter Vorteil lag in der Möglichkeit, die Werte von Parametern, Attributen oder Objekten direkt in der Oberfläche von *Unity* zu verändern. So ist es möglich, Input-Maps oder 3D-Modelle schnell zu wechseln. Auch Veränderungen während der Laufzeit sind prinzipiell möglich.

3.3 Pipeline

Der Ablauf des entwickelten Städtegenerators lässt sich in folgende Schritte aufteilen:

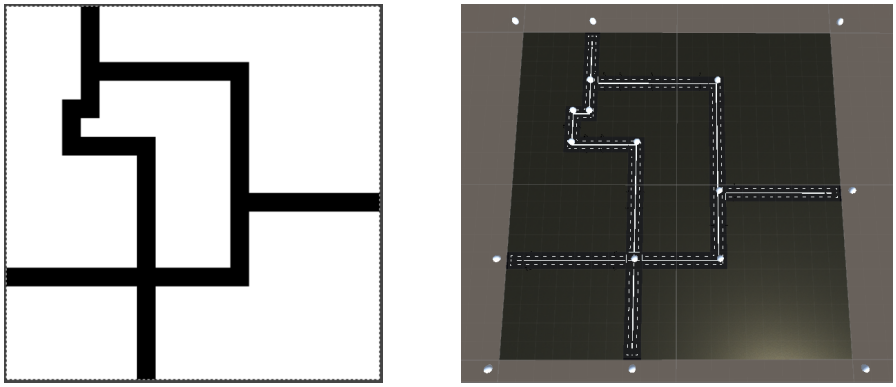


1. **RoadGenerator**
   Eine Straßenkarte gibt an, wo die Primärstraßen in der Stadt verlaufen. Die Karte wird gelesen und das primäre Straßennetzwerk wird erzeugt.
2. **CityCellCalculator**
   Die Primärstraßen fungieren als Grenzen zwischen den Nachbarschaften. Die Lage der Nachbarschaftszellen wird berechnet.
3. **DistrictChooser** (optional)
   Der Nutzer ordnet jeder Nachbarschaft einen Gebietstyp zu.
4. **SecondaryRoadGenerator**
   Innerhalb jeder Nachbarschaft breitet sich das sekundäre Straßennetz aus.
5. **BlockCalculator**
   Im gesamten Straßennetz sind nun von Straßen umschlossene Häuserblocks entstanden. Die Lage dieser Blockzellen wird analog zu Schritt 2 berechnet.
6. **LotCalculator**
   Jeder Häuserblock wird in Grundstücke aufgeteilt.
7. **BuildingGenerator**
   Auf den Grundstücken mit Straßenzugang werden Gebäude platziert.

Abbildung 7

3.3.1 RoadGenerator

Die Straßenkarten (*roadMaps*), die als erster Input verwendet werden, sind einfache binäre Bitmaps. Sie beinhalten zwei Informationen zur Stadt: Zum



(a) roadMap der Größe 15x15 **(b)** primäres Straßennetz mit Knoten

Abbildung 8: Platzierung der Knoten des aus der *roadMap* gelesenen primären Straßennetzes

einen deren Größe und zum anderen den Verlauf des primären Straßennetzwerks.
Die Größe der Stadt ergibt sich aus der Höhe und Breite der Bitmap. Für die Stadt ist dadurch in jedem Fall eine rechteckige Form gegeben.
Der Verlauf des primären Straßennetzwerks wird durch die schwarzen Pixel im Raster vorgegeben. Ein schwarzes Pixel steht für ein quadratisches Straßensegment. Das Straßennetzwerk orientiert sich auch weiterhin an der Grid-Struktur der *roadMap*. Ähnlich wie in *Undiscovered City* (siehe Kapitel 2.4.1) verlaufen die Straßen so lediglich horizontal und vertikal. Um klar den Verlauf der Straßen erkennen zu können, gibt es bei der Erstellung der Bitmaps die Bedingung, dass die gezeichneten Linien lediglich einen Pixel breit sein dürfen.

Der *RoadGenerator* iteriert nun durch die *roadMap* und wählt im Falle eines schwarzen Pixels den richtigen Straßentypen aus. Dafür muss er die Pixel in seiner Vierer-Nachbarschaft prüfen. Befinden sich in allen vier Himmelsrichtungen schwarze Nachbarpixel, muss beispielsweise eine Kreuzung generiert werden. Befinden sich oben und unten oder links und rechts schwarze Pixel, wird ein gerade verlaufendes Straßensegment gewählt, das zusätzlich noch korrekt ausgerichtet werden muss.
Parallel zur Wahl des Straßenbausteins, werden auf der Karte Knoten platziert. Durch die Verbindung der Knoten durch horizontal und vertikal verlaufende Kanten sollen später die Grenzen der Nachbarschaften definiert werden. Aus diesem Grund werden die Knoten an sämtlichen kritischen Punkten, an welchen der Straßenverlauf nicht ausschließlich geradeaus weitergeht, platziert. Darunter fallen zum einen Kreuzungen, Kurven und Sackgassen auf den Straßen. Außerhalb der Stadtgrenzen werden Knoten zudem an den vier Eckpunkten sowie an Stellen, an denen Straßen an die

äußeren Ränder der Stadt führen, platziert. Durch diese beiden Fälle werden auch die äußeren Grenzen der Stadt erfasst.
Jeder der zu diesem Zeitpunkt erzeugten Knoten besitzt folgende Attribute: die Position im Raster, einen eindeutigen Index sowie die Orientierung. Letztere speichert, in welche Richtungen sich von ihm ausgehend Straßen ausbreiten. Bei äußeren Knoten wird zusätzlich bedacht, wo von ihnen aus der Stadtrand entlangläuft. Die möglichen Orientierungen bestehen aus oben, rechts, unten und links.
Ist die Iteration des *RoadGenerator* durch die *roadMap* abgeschlossen, wird die Liste mit allen Knoten an den *CityCellCalculator* weitergegeben.

3.3.2 CityCellCalculator

Zu dem Zeitpunkt, an dem der *CityCellCalculator* aufgerufen wird, sind die Primärstraßen bereits in der Welt platziert und das primäre Straßennetz wird durch eine Menge an Knoten beschrieben.
Die Knoten wurden so platziert, dass durch deren Verbindungen mit geradlinigen Kanten der Verlauf aller Straßen sowie der Stadtgrenzen abgedeckt werden kann. Der darauffolgende Schritt ist die Berechnung der Nachbarschaftszellen, welche vom primären Straßennetz und den äußeren Stadtgrenzen umschlossen werden.

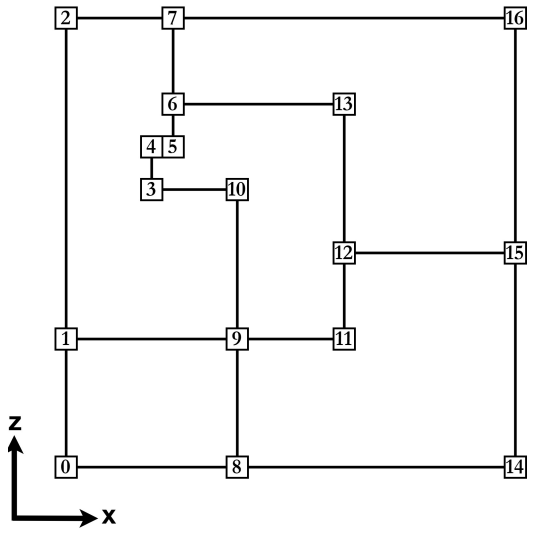


Abbildung 9: Darstellung des planaren Graphen

Der erste Teilschritt ist, diese Kanten als Verbindungen der Knoten aufzustellen. Die Menge der Knoten und Kanten kann durch einen ungerichteten, planaren Graphen beschrieben werden. Um das zu ermöglichen, wird eine Adjazenzliste erstellt. Diese speichert für jeden Knoten die direkt verbundenen Nachbarknoten.

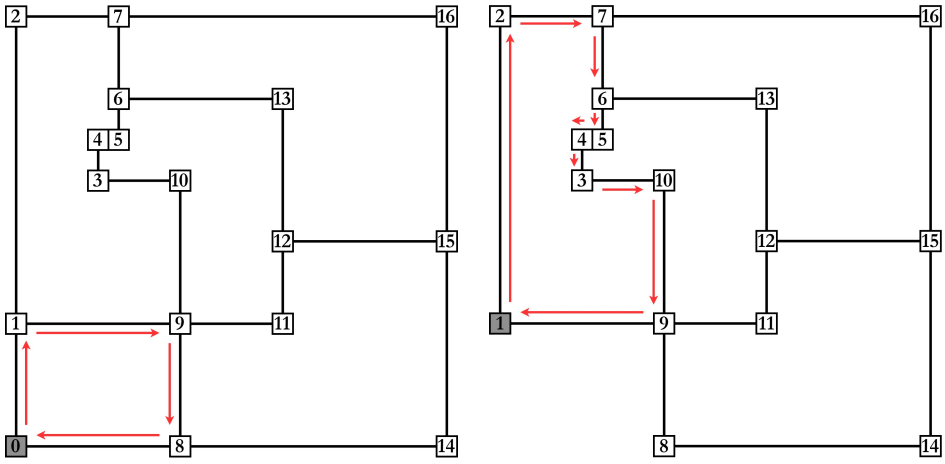
Implementiert wird die Berechnung der Adjazenzliste, indem vom jeweiligen Ausgangsknoten aus alle gegebenen Orientierungen entlang iteriert wird. Wurde der gesuchte Nachbarknoten gefunden, wird dieser in der Adjazenzliste als Nachbar des Ausgangsknoten in der jeweiligen Richtung gespeichert.
Die Adjazenzliste ermöglicht es nun, die Nachbarschaftszellen zu berechnen. Gesucht ist dafür die *Minimum Cycle Basis*. Sie ist die minimale Menge an Kreisläufen im Graph, aus denen alle anderen möglichen Kreisläufe konstruiert werden können [15]. Veranschaulicht sind das die Abfolgen an Kanten, die eine umschlossene Region begrenzen. In diesen Abfolgen kann keine Kanten entfernt werden, ohne dass der Kreislauf unterbrochen wird.

Minimum-Cycle-Basis-Algorithmus

Der *Minimum-Cycle-Basis-Algorithmus* [16] extrahiert die Nachbarschaftszellen aus Kartensicht von links nach rechts. Aus diesem Grund müssen im Vorhinein die Knoten nach ihrer x-Position und bei Gleicheit nach z-Position sortiert werden. So ist der nächstfolgende Knoten in der *nodeList* auch immer der aus geometrischer Sicht nächstfolgende.

Bei der Extrahierung der Zellen (vgl. Algorithmus 1) folgt man dem Prinzip der priorisierten Orientierung im Uhrzeigersinn. Das bedeutet, dass nach jeder entlanggelaufenen Kante versucht wird, im Uhrzeigersinn eine Orientierung weiterzugehen. Gibt es keinen Nachbarknoten in die gewünschte Richtung, probiert man stufenweise die folgenden Orientierungen gegen den Uhrzeigersinn aus.
Zu Beginn wird die Ausgangsorientierung nach oben festgelegt. Veranschaulicht an Abb. 10b bewegt man sich also vom Ausgangsknoten N_1 zum oberen Nachbarknoten N_2 und bildet dazwischen eine Kante. Diese besitzt N_1 als Startknoten und N_2 als Endknoten.
Als nächstes gilt es als Priorität, einen rechten Nachbarknoten vom jetzigen Startknoten N_2 zu finden, da rechts im Uhrzeigersinn die nächste Orientierung ist. Existiert kein Nachbarknoten in die priorisierte Richtung, werden schrittweise die nächsten Richtungen gegen den Uhrzeigersinn geprüft. So kann man von N_6 ausgehend statt nach links lediglich nach unten zu N_5 gehen. Von N_4 ausgehend ist es sogar nur möglich nach unten anstatt nach oben zu gehen. Sackgassen können auf diese Weise auch erkannt und berücksichtigt werden.
Dieser Algorithmus wird für eine Zelle solange fortgeführt bis man erneut beim Ausgangsknoten angekommen ist und damit einen geschlossenen Kreislauf erzeugt hat. Die gespeicherte Folge an Kanten wird daraufhin als eine Zelle gespeichert.
Bevor die nächste Zelle berechnet wird, werden die nicht mehr benötig-

ten Kanten und Knoten aus dem gerade berechneten Kreislauf entfernt. So fängt man den *Minimum-Cycle-Basis-Algorithmus* nur noch von Knoten an, die Teil eines noch nicht erfassten Kreislaufs sind.



(a) Durchlauf 1 **(b)** Durchlauf 2

Abbildung 10: Ablauf des Minimum-Cycle-Basis-Algorithmus

Algorithm 1 Zelle von einem Knoten ausgehend extrahieren

```
if pNode is active then
    currentNode = pNode;
    prioritizedOrientation = up;
    do
        orientation = prioritizedOrientation;
        while no neighbour at this orientation do
            turn orientation anti-clockwise;
        neighbour = adjacencyList[currentNode, orientation];
        edge = new Edge(currentNode, neighbour, orientation);
        cell.Add(edge);
        prioritizedOrientation = turn orientation clockwise;
        currentNode = neighbour;
    while currentNode != pNode
```

Entfernen nicht mehr benötigter Knoten und Kanten

Zuerst wird berechnet, welche Kanten nicht mehr benötigt und somit entfernt werden können. Sind alle überflüssigen Kanten entfernt worden, kann auch bemessen werden, welche Knoten nicht mehr benötigt werden (vgl. Algorithmus 2).

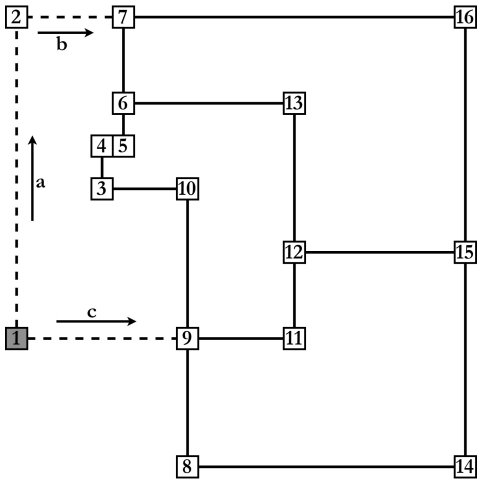


Abbildung 11: Traversierung entlang der zu entfernenden Kanten

Zu Beginn muss man festhalten, dass der Ausgangsknoten des Kreislaufs aufgrund der Sortierung von allen übrigen Knoten immer am weitesten unten links liegt. Angesichts der Position kann der Ausgangsknoten kein Teil eines weiteren Kreislaufs mehr sein. Alle potenziellen Zellen links sowie unterhalb von ihm bereits wurden bearbeitet und rechts oberhalb von ihm befindet sich in jedem Fall der gerade berechnete Kreislauf.
Die von diesem Knoten ausgehende Kante nach oben kann aus diesem Grund prinzipiell entnommen werden und wird in der Adjazenzliste für beide Knoten entfernt (vgl. Abb. 11 Kante a).
Mit jeweils einer Traversierung im und gegen den Uhrzeigersinn können weitere nicht mehr benötigte Kanten identifiziert werden.
Im ersten Durchgang (im Uhrzeigersinn) wird eine Kante entfernt, wenn ihr Startknoten weniger als zwei noch angebundene Nachbarknoten hat. N_2 ist zum Beispiel aufgrund der Herausnahme von Kante a nur noch an N_7 gebunden (vgl. Abb. 11 Kante b). In dem Fall, dass der Startknoten nicht mehr als eine Verbindung zu einem Knoten besitzt, kann garantiert werden, dass die zugehörige Kante für keinen weiteren Kreislauf mehr infrage kommt.
Erreicht man eine Kante, bei dem das nicht der Fall ist, kann die Traversierung beendet werden. Analog dazu traversiert man noch gegen den Uhrzeigersinn und prüft jeweils den Endknoten einer Kante auf die Anzahl an Nachbarknoten. So kann Kante c ebenfalls entfernt werden, da ihr Endknoten N_1 nur noch die Verbindung zu N_9 besitzt.

Knoten, die nach Entfernung der Kanten in der Adjazenzliste keine Nachbarknoten mehr vorweisen, werden deaktiviert. Diese kommen als Ausgangsknoten für den *MCB-Algorithmus* nicht mehr in Frage. So werden

in diesem Fall sowohl N_1 als auch N_2 deaktiviert. Der Ausgangsknoten für die nächste Zelle ist daher nicht N_2, sondern N_3. Gibt es keine aktiven Knoten mehr in der *nodeList*, wurden alle Nachbarschaftszellen auf der Karte berechnet.

Algorithm 2 Knoten und Kanten nach Extrahierung einer Zelle entfernen

```
remove(cell.edges[0]);
for all other edges in clockwise direction do
    if edge.startNode has less than 2 neighbour nodes then
        remove(edge);
    else
        break;
for all other edges in anti-clockwise direction do
    if edge.endNode has less than 2 neighbour nodes then
        remove(edge);
    else
        break;
for all nodes in the cell do
    if node has no more neighbours then
        deactivate(node);
```

3.3.3 DistrictChooser

Der *DistrictChooser* ist nach der Erzeugung der Straßenkarte und der Wahl der Seeds die letzte Steuerungsinstanz durch den Nutzer. Sie findet während der Laufzeit statt. Zu diesem Zeitpunkt sieht man bereits die Grundform der Stadt und das primäre Straßennetz. Der Verlauf der Nachbarschaftszellen wird bereits ähnlich wie in Abb. 12 dargestellt. Dabei werden die Zellen jedoch nur nacheinander angezeigt. Durch das Drücken der Tasten 1-3 kann der Nutzer der in dem Moment ausgewählten Nachbarschaft einen Gebietstyp zuordnen. Dabei ordnet die 1 ein Wohnviertel zu, die 2 ein Hochhausviertel und die 3 ein Industrieviertel.
Optional kann der *DistrictChooser* auch deaktiviert werden. In diesem Fall werden den Nachbarschaften automatisch Gebietstypen zugeordnet.

3.3.4 SecondaryRoadGenerator

Der *SecondaryRoadGenerator* erhält die Liste an Nachbarschaftszellen mit dem zugeordneten Gebietstyp als Input. Seine Aufgabe ist nun, innerhalb jeder dieser Nachbarschaften das sekundäre Straßennetz aufzuspannen. Dies geschieht unabhängig für jede Nachbarschaft, jedoch abhängig vom Gebietstyp.

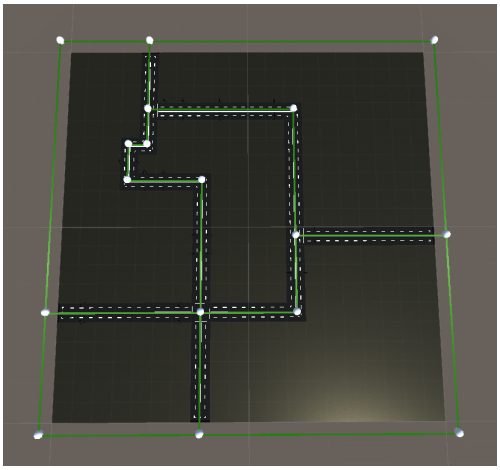


Abbildung 12: Visualisierung der Grenzkanten zwischen den Nachbarschaften

Die Generierung des Straßennetzes basiert auf den Ansätzen des parallelen Wachstums von Kelly und McCabe [15] sowie auf dem dort ebenfalls herangezogenen und in Kapitel 2.4.2 beschriebenen L-System von Parish und Müller [13], das auch für die Entwicklung der *CityEngine* genutzt wurde.

L-System-Ansatz

Der Ausgangspunkt bei der Entwicklung des Algorithmus war das L-System von Parish und Müller. Es stellte sich jedoch heraus, dass dieses an einigen Stellen deutlich vereinfacht werden kann und die Struktur eines L-Systems nicht zwingend nötig ist. Von bestimmten Seiten wurde das L-System lediglich als umständlicher Formalismus bezeichnet. So vereinfachte beispielsweise der Spieleentwickler Sean Barrett in einem Blogartikel [17] das L-System so weit, dass es strukturell einem L-System nicht mehr ähnelte. Auch die Entwickler von *CityGen* entfernten sich trotz des gleichen Ansatzes vom Konzept des L-Systems.
Aus diesem Grund wurde zwar der prinzipielle Ablauf sowie die Idee des parallelen Wachstums übernommen. Formal hält sich der Algorithmus jedoch nicht an das L-System. Folgende Vereinfachungen des L-Systems (Abb. 5) sind dabei elementar:

1. Die Erstellung eines Branch B sowie einer Abfrage I muss nicht in mehrere Schritte unterteilt und verzögert werden.

    (a) Anstatt wie in p_2 zunächst abzweigende Branches zu erstellen, welche in der nächsten Iteration im Falle delay == 0 in ein Straßensegment R umgewandelt werden, können alle folgenden Segmente direkt als Segment R behandelt werden. Durch einen Ver-

merk, ob es sich um eine Gerade oder eine linke oder rechte Abzweigung handelt, kann man diese dennoch unterscheiden.

(b) Anstatt wie in ω, p_2 oder p_5 zunächst eine Abfrage anzuhängen, deren Ergebnis erst in der übernächsten Abfrage evaluiert wird, kann die Abfrage in der Implementierung direkt gestartet und deren Ergebnis evaluiert werden.

2. Aufgrund der Punkte 1(a) und 1(b) kann die Menge der anzuhängenden Variablen auf R reduziert werden.

3. Da es nur noch eine Variable gibt, erscheint der Formalismus eines L-Systems nicht mehr nötig. Stattdessen kann eine Liste genutzt werden, die für jedes potenzielle, neue Straßensegment einen generischen Kandidaten R (ähnlich dem *idealSuccessor*) anhängt.

Algorithmus zur Erzeugung des sekundären Straßennetzes

Zu Beginn anzumerken ist, dass auch das sekundäre Straßensystem auf der Grid-Struktur basiert. Somit gibt es weiterhin lediglich gerade Straßenverläufe. Neu entstehende Branchsegmente wie aus p_2 biegen damit immer um 90° nach links oder rechts ab.
Im Vergleich zu den Primärstraßen sind die Sekundärstraßen zudem nur halb so breit und lang. Das bedeutet, dass ein Segment einer Sekundärstraße lediglich ein Viertel einer Gridfläche belegt.

Der Algorithmus (vgl. Algorithmus 3) fängt damit an, pro Zelle zwei oder mehr Ausgangspunkte festzulegen. Von diesen soll das sekundäre Straßensystem parallel ins Innere der Zelle hineinwachsen. Gewählt werden dafür die Mittelpunkte der längsten Kante der Zelle.
Die Startpunkte sind die ersten Kandidaten, die in die Kandidatenliste eingefügt werden. Jeder Kandidat speichert seine Position und die Orientierung, also die Richtung, in die das Straßenwachstum von ihm aus gerade weitergehen würde. Zusätzlich besitzt jeder Kandidat *roadAttributes*, die gesamtheitlich für eine gerade verlaufende Straße gelten und an nicht abzweigende Nachfolgesegmente weitergegeben werden. Innerhalb dieser *roadAttributes* wird beispielsweise das *delay* für Abzweigungen gespeichert. Weitere Attribute wie die Straßenlänge könnten dort auch ergänzt werden.
Pro Ausgangspunkt, von dem das Straßensystem aus wächst, wird ein Objekt der Klasse *streetTree* erzeugt. Dieses speichert für jeden Wachstumsverlauf die derzeitige Position und die Kandidatenliste. Notwendig ist das, da der Algorithmus alle *streetTrees* einer Zelle parallel bearbeitet bzw. immer kandidatenweise zwischen den *streetTrees* wechselt. Erst wenn die Listen aller *streetTrees* einer Zelle keine weiteren Kandidaten mehr enthalten, ist die Generierung in der jeweiligen Zelle abgeschlossen.

Befinden sich Kandidaten in der Liste, wird in jeder Iteration der oberste Kandidat herausgenommen. Bis zu diesem Zeitpunkt wurde dieser lediglich als Platzhalter in die Liste eingefügt. Daher wird nun seine Position berechnet und die *localConstraints*-Funktion initiiert. Geprüft wird dabei, ob die Position des vorgeschlagenen Segments bereits bebaut ist oder sich Straßen in unmittelbarer Nähe befinden.
Ist die Straße bereits bebaut, wird der Kandidat nicht weiter verfolgt und es gehen auch keine weiteren Straßen von ihm aus.
Ist die Stelle frei, aber in der aus Orientierungssicht vorderen Umgebung befinden sich Straßen, wird zusätzlich ein Snap-Test (siehe Abb. 13) durchgeführt. Dieser prüft, ob sich die Straße mit den benachbarten Straßenteilen verbinden darf. Da nicht vier Straßensegmente zusammen in einem Block stehen dürfen, gibt es verschiedene Fälle, die ein Verbinden mit der Nachbarstraße nicht erlauben.

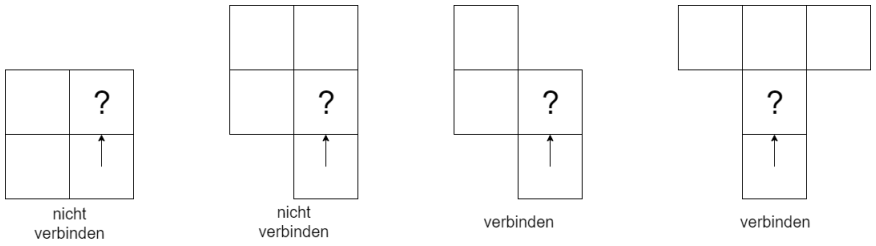


Abbildung 13: Snap-Tests bei Straßensegmenten in der Nachbarschaft (?... zu prüfendes Segment, Pfeil...Ausgangsrichtung)

Nach jedem platziertem Straßensegment wird zudem getestet, ob die Straße an dieser Stelle abzweigen soll. Entschieden wird zunächst anhand der *delay*-Parameter in den *roadAttributes* des Kandidaten. Diese gibt es für die Fortführung der Straße sowie die linke und rechte Abzweigung. Sowohl für links als auch rechts werden diese Parameter bei jedem neuen Straßensegment runtergezählt bis sie gleich null sind. Danach wird der jeweilige Parameter neu gesetzt.
Implementiert wurde die Belegung der *delays* mit Hilfe eines Pseudo-Zufallsgenerators, der variierende Werte aus einem je nach Gebietstyp vorgegebenem Wertebereich erzeugt. Bei gleichem Seed entstehen also immer die gleichen Ergebnisse. Dieser Wert kann in *Unity* durch den Nutzer verändert werden.
Sind die Werte im Wertebereich klein, entsteht im Gesamten ein dichtes Straßennetz. Sind sie größer, ist das Netz weniger dicht und die Straßen verlaufen geradliniger. So lassen sich für die Straßennetze der Gebietstypen unterschiedliche Eigenschaften festlegen. Sichtbar wird das in Abb. 14: In den zugeordneten Wohngebieten unten links und rechts oben entsteht ein sehr enges Straßennetz, während im Hochhausviertel oben links und unten rechts oder im Industriegebiet in der Mitte ein gröberes Netz ent-

steht.

Algorithm 3 sekundäres Straßenwachstum innerhalb einer Nachbarschaft

```
for all startPoints do
    streetTrees.Add(new StreetTree());
    c = Candidate(tree.pos, initRoadAttr, "Start");
    treeList.Add(c);
while not all treeLists are empty do
    for all streetTrees do
        if treeList[0] is buildable then
            place segment;
            if delayLeft == 0 then
                bL = new Candidate(tree.Pos, newRoadAttr, "Left");
                treeList.Add(bL);
            else
                decreaseDelayLeft;
            if delayRight == 0 then
                bR = new Candidate(tree.Pos, newRoadAttr, "Right");
                treeList.Add(bR);
            else
                decreaseDelayRight;
            if delay == 0 then
                c = new Candidate(tree.Pos, roadAttr, "Straight");
                treeList.Add(c);
```

Platzierung der Knoten

Wenn für jede Nachbarschaftszelle das sekundäre Straßennetz berechnet wurde, iteriert der *SecondaryRoadGenerator* durch die um die Sekundärstraßen ergänzte Karte und wählt die richtigen Straßentypen aus.
Bei der Wahl der Sekundärstraßen muss die Lage zu Primär- und anderen Sekundärstraßen beachtet werden. Sekundärstraßen sind Primärstraßen untergeordnet und müssen anders als bei gleichrangigen Sekundärstraßen die Vorfahrt der kreuzenden Straße beachten. Daher musste eine höhere Variation an Straßentypen integriert werden als noch im *RoadGenerator*.
Ebenso wie beim primären Straßennetz soll das sekundäre Straßennetz durch einen planaren Graphen bestehend aus Knoten definiert werden. Ab diesem Zeitpunkt muss zwischen *Primary* und *Secondary Nodes* unterschieden werden. *Primary Nodes* liegen auf Primärstraßen und wurden größtenteils bereits im *RoadGenerator* erzeugt. *Secondary Nodes* liegen auf den gerade er-

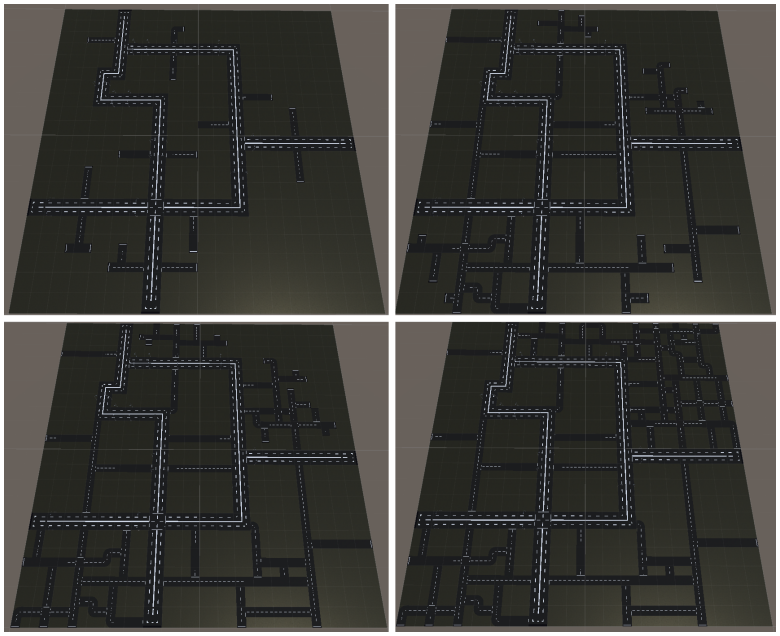


Abbildung 14: Wachstumsfortschritt des sekundären Straßennetzes

zeugten Sekundärstraßen und müssen im Folgenden gesetzt werden. Die Differenzierung zwischen *Primary* und *Secondary Nodes* ist im weiteren Verlauf des Programms wichtig.
Die Platzierung der neuen Knoten auf dem sekundären Straßennetz erfolgt exakt wie im *RoadGenerator* an „kritischen Punkten“. Zusätzlich zu beachten ist jedoch, dass ein *Primary Node* ergänzt werden muss, wenn das sekundäre Straßennetz auf eine Primärstraße trifft. In diesem Fall entsteht eine neue Kreuzung auf dem primären Straßennetz.

3.3.5 BlockCalculator

Der *BlockCalculator* erhält die Liste an *Primary* und *Secondary Nodes* vom *SecondaryRoadGenerator*. Durch diese Liste ist das nun vollständige Straßennetz aus Primär- und Sekundärstraßen definiert.
Berechnet werden sollen nun die aus dem Straßennetz resultierenden Häuserblocks. Häuserblocks sind die Zellen, die von Primär- und Sekundärstraßen sowie den äußeren Stadtgrenzen umschlossen werden. Im Grunde verrichtet der *BlockCalculator* also auf dem nun vollständigen Straßennetz die gleiche Arbeit wie der *CityCellCalculator*. Der Ablauf ist daher auch nahezu identisch: Man berechnet die Adjazenzliste und wendet auf diese den *Minimum-Cycle-Basis-Algorithmus* an.
Zusätzlich beachtet werden musste die Verbindung zwischen *Primary* und

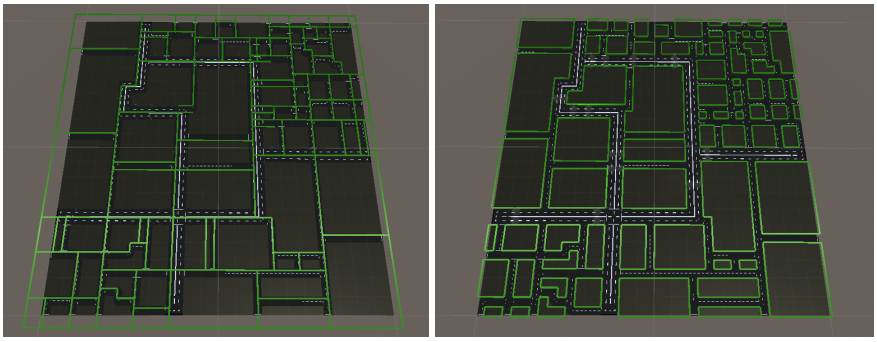


Abbildung 15: die berechneten Häuserblocks vor (links) und nach (rechts) genauer Verlegung der Kanten

Secondary Nodes. Während *Primary Nodes* wie bereits erwähnt ein vollständiges Gridfeld abdecken, ist es bei *Secondary Nodes* lediglich ein Viertel eines Feldes. Um bei den aufzuspannenden Kanten bestimmen zu können, ob diese die Begrenzung für eine ganze oder nur eine halbe Gridbreite darstellen, muss berücksichtigt werden, von welchem Typ der jeweilige Knoten ist.
Sind die Blockzellen berechnet worden, müssen die Kanten noch an die exakten Ränder der Blocks verschoben werden (vgl. Abb. 15). Bisher beschrieben Kanten die Grenzen zwischen den umschlossenen Bereichen. Für den weiteren Verlauf sollen die Kanten jedoch exakt die innere Grenze eines Blocks beschreiben. Die Straßen werden bei der Eingrenzung nun vollständig ausgeschlossen.

3.3.6 LotCalculator

Input für den *LotCalculator* ist die Liste an Häuserblocks. Dessen Aufgabe ist es nun, die Blocks in Grundstücke (Lots) aufzuteilen. Diese geben den Bereich vor, auf dem später Gebäude platziert werden sollen. Der Algorithmus basiert dabei erneut auf den Algorithmen aus "Procedural Modeling of Cities" [13] und "Citygen: An Interactive System for Procedural City Generation" [15].

Der Algorithmus (vgl. Algorithmus 4) verkleinert einen Häuserblock schrittweise in kleinere Regionen. Entscheidend bei der Unterteilung ist die *lotSize*. Sie legt fest, ab welcher Größe die Regionen nicht mehr verkleinert werden sollen bzw. wie groß Grundstücke maximal sein dürfen.
Zu Beginn wird auf der längsten Kante der Region der Mittelpunkt ausgewählt. Von diesem Punkt ausgehend stellt man eine orthogonal zur Kante verlaufende Gerade auf und berechnet deren ersten Schnittpunkt mit einer anderen Kante der Region. Entlang der Linie zwischen den beiden Punkten

wird die Region daraufhin geteilt.

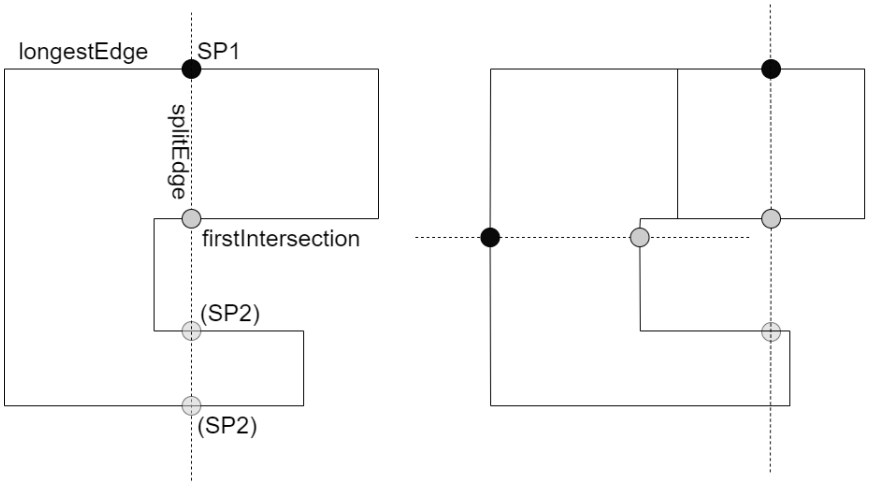


Abbildung 16: zwei Unterteilungsschritte eines Beispielblocks

Liegt die Länge der zu Beginn berechneten längsten Kante einer Region unter der *lotSize*, wird die Region nicht mehr verkleinert. Deren Form ist somit final und wird als Grundstück gespeichert. Die restlichen Regionen des Blocks können jedoch weiterhin noch zerteilt werden.
Die Zielgröße *lotSize* wird ebenso wie die *delay*-Parameter aus dem *SecondaryRoadGenerator* mit Hilfe eines Seeds pseudo-zufällig für jede Region berechnet. Der Wertebereich variiert ebenso nach Gebietstyp. Während Grundstücke in Wohngebieten eher klein sind, sind sie in Industriegebieten beispielsweise signifikant größer (vgl. Abb. 17).

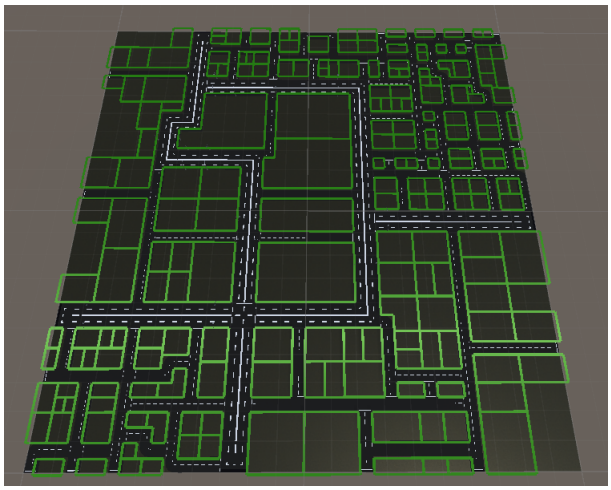


Abbildung 17: Unterteilung der Häuserblöcke in Grundstücke

Ist die Unterteilung eines Blocks in Grundstücke abgeschlossen, werden

diese auf Straßenzugang getestet. Gebäude ohne Straßenzugang sind in echten Städten die Ausnahme. In den meisten Fällen werden freie Flächen inmitten eines Häuserblocks als Innenhof, Garten oder anderweitig genutzt. Liegt ein Grundstück daher nicht an der Straße, wird es auch nicht an den *BuildingGenerator* weitergegeben.

Algorithm 4 Berechnung der Grundstücke eines Blocks

```
regionQueue.Add(block);
while regionQueue is not empty do
    region = regionQueue[0]
    if region.longestEdge.size > lotSize then
        splitPoint1 = region.longestEdge.midpoint;
        splitEdge = splitPoint1.perpendicular;
        minimalDistance = unreachable high value;
        for all orthogonal edges do
            splitPoint2 = intersection(splitEdge, orthogonalEdge);
            if distance(splitPoint1, splitPoint2) < minimalDistance then
                firstIntersection = splitPoint2;
        (region1, region2) = splitRegion(splitPoint1, firstIntersection);
        regionQueue.Add(region1);
        regionQueue.Add(region2);
    else
        lotList.Add(region);
    regionQueue.removeAt(0);
```

3.3.7 BuildingGenerator

Der Input des *BuildingGenerator* ist die Liste an Grundstücken. Die Ausgangsform der Gebäude ist mit einer Quaderform, die an Breite und Länge der Grundstücksmaße angepasst ist, simpel gehalten. Die Höhe des Gebäudes hängt vor allem vom Gebietstyp ab. Auch ansonsten besitzen die Gebäude in den drei Gebietstypen bestimmte Eigenschaften:

- In **Wohnvierteln** sind die Gebäude eher klein und nicht sonderlich hoch. Der Abstand zu den Nachbarhäusern ist recht groß. Zudem besitzen sie Schrägdächer.
- In **Hochhausvierteln** sind die Grundstücke mittelgroß. Die Gebäude nehmen die Form größerer Bürogebäude bis hin zu Wolkenkratzern mit einer überdurchschnittlichen Höhe an. Die Grundstücke werden so weit es geht bebaut, sodass die Gebäudedichte hoch und der Abstand zu Nachbargebäuden gering ist.

- In **Industrievierteln** sind die Gebäude sehr breit und mittelhoch. Die Grundstücke werden nicht vollständig bebaut, da auf Industriegeländen beispielsweise Platz für Anlieferung berücksichtigt werden muss. Schornsteine oder Firmenschilder werden als zusätzliches Accessoires auf dem Dach platziert.

Für jedes Gebäude wird ein Objekt der Klasse *building* instanziiert. Dort werden alle Attribute eines Gebäudes festgehalten und beispielsweise die Skalierungswerte berechnet.
In dem Fall, dass die Grundstücksfläche nicht aus vier sondern mehr Kanten besteht, wird das Gebäude innerhalb der Klasse zudem in mehrere rechteckige *buildPieces* unterteilt, die zusammen die erforderliche Gebäudeform ergeben. Der *BuildingGenerator* erzeugt letztlich alle vorliegenden *buildPieces* eines *building*.
Die Höhe der Gebäude wird ebenfalls durch einen Pseudo-Zufallsgenerator bestimmt. Innerhalb eines Blocks wählt dieser für jedes Gebäude eine Höhe innerhalb eines vorher gesetzten Wertebereichs aus.

Die Abstände der Gebäude zum Straßenrand werden durch die Platzierung der Gebäude in den jeweiligen vorgefertigten Objekten (Prefabs) in *Unity* realisiert. Lässt ein Gebäude viel Platz zum Straßenrand, wird es weiter von den Ursprungskoordinaten des Prefabs platziert.Für jeden Gebietstyp wurde eine Menge an Prefabs mit unterschiedlichen, einfachen Texturen vorgefertigt. Industriegebäude haben außerdem mit Schornsteinen oder Firmenschildern sowie Wohnhäuser mit Schrägdächern je ein weiteres variierendes Merkmal. Die Wahl der Prefabs wird ebenfalls von einem pseudo-zufälligen Generator gesteuert, sodass die Ergebnisse reproduzierbar bleiben.

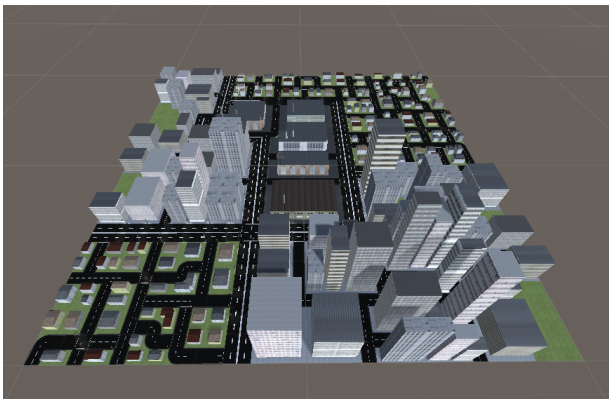


Abbildung 18: finale Stadt nach Platzierung der Gebäude

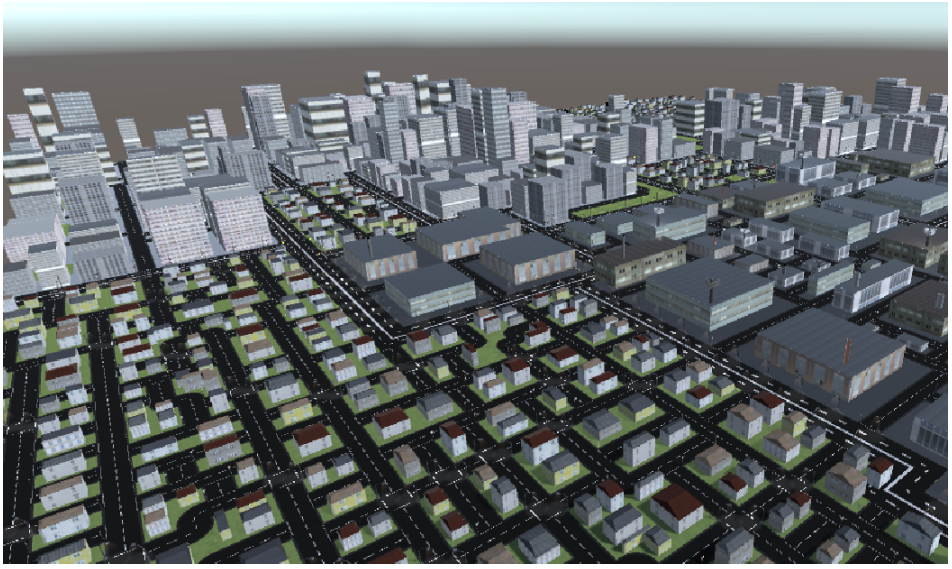


Abbildung 19: Stadt aus einer anderen *roadMap*

4 Bewertung

Im Allgemeinen lassen sich Anwendungen zur prozeduralen Städtegenerierung nicht immer anhand der gleichen Kriterien bewerten, da verschiedene Anwendungen unterschiedliche Schwerpunkte besitzen. Sei es die Rekonstruktion realer Städte, die Möglichkeit, Städte in Echtzeit zu bearbeiten oder die Entwicklung von Städten möglichst realistisch zu simulieren.
Um die im Rahmen dieser Bachelorarbeit entwickelte Anwendung aus verschiedenen Gesichtspunkten zu bewerten, wird ein Framework aus „A Survey of Procedural Techniques for Procedural City Generation“ [12] genutzt, das Anwendungen in diesem Bereich anhand von sieben verschiedenen Kriterien bewertet.

4.1 Realismus

Die Straßengenerierung stützt sich auf ein Rastersystem mit gleichförmigen, quadratischen Flächen. Die Dynamik des Straßennetzes ist daher begrenzt. Schräg verlaufende Straßen, verschiedene Kurvenverläufe oder unterschiedlich lange Straßensegmente sind nicht möglich. Straßennetze organisch gewachsener Städte, beispielsweise in Ringform, können so höchstens angenähert werden (vgl. Abb. 21).
Näher kommen die Ergebnisse an die Schachbrettmuster amerikanischer Planstädte. Dort verlaufen die Straßen entweder parallel oder orthogonal

zueinander. Insbesondere in den Hochhausvierteln ähnelt sich der Aufbau. Ein tatsächlich einheitliches Straßennetz lässt sich durch Anpassung der Parameter zwar annähern, der Algorithmus hat jedoch keinen Überblick über das Gesamtwachstum und den noch freien Raum. So reagiert er nur lokal auf das eigene Wachstum und kann Abweichungen in der Blockgröße nicht vollständig vermeiden. In den meisten Fällen sorgt die Anwendung daher für ein wechselhaftes Straßennetz mit statischen Straßenzügen.

Abbildung 20: entfernte Aussicht auf Hochhausviertel in der Anwendung (links) und auf Los Angeles in GoogleMaps [18] (rechts)

Die Gebäude basieren alle auf einfachen Quadern und unterscheiden sich damit nie grundlegend voneinander. Durch die Texturierung, Skalierung und zusätzliche Geometrie wie Dächer oder Reklame kann die einfache Struktur jedoch vor allem aus der Distanz kaschiert werden. Trotz fehlender detaillierten Modellierung der Fassaden sind Gebäudetypen leicht erkennbar.
Auffallend sind die fehlenden Übergänge zwischen den Nachbarschaften vor allem aus der Vogelperspektive (vgl. Abb. 21) und weniger aus geringer Distanz (vgl. Abb. 22b). Tatsächliche Grenzen zwischen Nachbarschaften sind im Normalfall fließend, sodass eine hoher Bürokomplex nicht nur eine Straße von einem zweistöckigen Eigenheim entfernt liegt.
Zusätzlich fehlen natürliche Aspekte wie Vegetation, Wälder, Flüsse oder Seen, die eine Stadt gerade in weniger urbanen Stadtteilen authentischer aussehen lassen.

Ein realistischer Eindruck einer Stadt entsteht aus heutigen Ansprüchen also nur sehr bedingt. Einzelne homogene Nachbarschaften wie die an Stadtzentren amerikanischer Großstädte erinnernden Hochhausviertel erzeugen einen verhältnismäßig stimmigen Eindruck. Für einen stimmigen Gesamteindruck fehlt es jedoch an dynamischeren Straßenverläufen sowie Übergängen zwischen den Stadtgebieten.

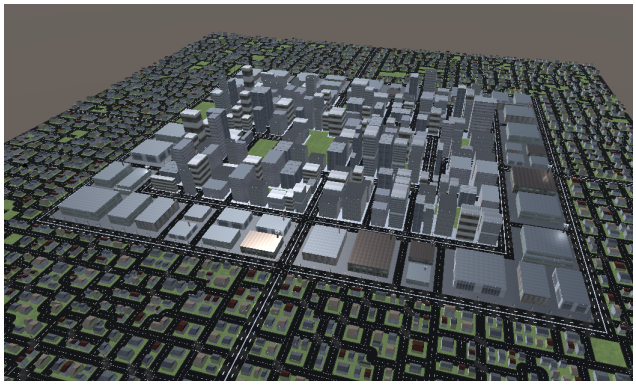


Abbildung 21: Stadt mit in *roadMap* angenäherter Ringform

4.2 Skalierung

Die Größe der zu generierenden Städte ist lediglich durch die Größe der Input-*Roadmaps* begrenzt. In der Theorie gibt es bei der Skalierung also keine Grenzen. Lediglich die Performanz kann die praktischen Möglichkeiten bei der Skalierung einschränken.

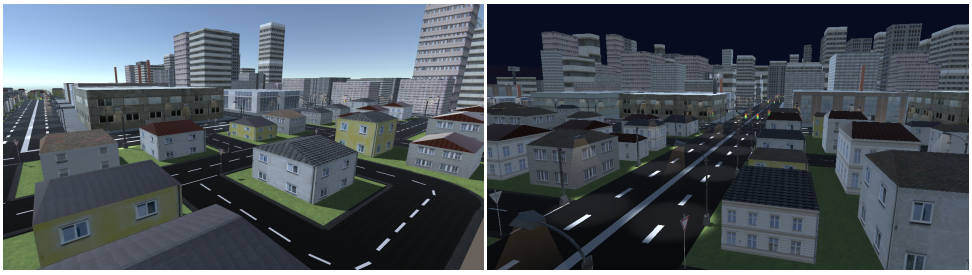
4.3 Variation

Das Straßennetz bietet vor allem Variationen hinsichtlich seiner Dichte. Während es in Wohngebieten sehr verwinkelt ist und viele kleine Häuserblocks entstehen, decken die Straßen Industriegebiete großzügiger ab und lassen größere Blocks entstehen. Zudem gibt es mit den Primär- und Sekundärstraßen zwei unterschiedliche Straßengrößen.
Durch das Fehlen dynamischer Richtungswechsel und längerer Straßensegmente kann sich das Straßennetz jedoch nie vollständig von den Ähnlichkeiten zum Schachbrettmuster lösen.

Die Gebietstypen unterscheiden sich optisch erheblich voneinander. Dies erklärt sich insbesondere an deren verschiedenen Gebäudetypen und -maßen. Der Unterschied zwischen kleinen Eigenheimen, großen Industriehallen oder hohen Bürogebäuden ist auffallend.
Die Variation innerhalb eines Gebietstyps ist jedoch eher gering. Zwar haben die Gebäude unterschiedliche Grundformen, jedoch basieren alle Gebäude auf einfachen rechteckigen Grundrissen. Komplexere Formen und abwechslungsreiche Gebäudefassaden fehlen.

4.4 Input

Als Input wird eine von Hand erstellte Straßenkarte benötigt. Diese gibt durch die Begrenzung der Nachbarschaften die grobe Struktur der Stadt



(a) bei Tag **(b)** bei Nacht

Abbildung 22: Städte aus geringer Entfernung

vor. Ansatzweise wird also Kartenmaterial eingelesen. Informationen zur Zuordnung der Gebietstypen werden zudem auch während der Laufzeit vom Nutzer übergeben.
Echte geographische Karten mit umfangreichen Daten können jedoch nicht eingelesen werden.

4.5 Effizienz

Die Performanz hängt maßgeblich von der Größe der Stadt sowie den gewählten Gebietstypen ab. Mit zunehmender Dichte des Straßennetzes, steigt die Menge an Knoten, Kanten und Zellen, die berechnet werden müssen. Kleinere Grundstücke erhöhen die Laufzeit des *LotCalculator* und führen zu einer höheren Anzahl zu bauender Gebäude.
Als Beispiele wurden drei Städte mit den Maßen 15x15, 50x50 und 100x100 getestet. Sechs gleich große Nachbarschaften wurden so bebaut, dass jeder Gebietstyp je zweimal auftaucht und je ein Drittel der Stadtfläche ausmacht.

	15x15	50x50	100x100
RoadGenerator	0,014 sec	0,039	0,063
CityCellCalculator	0,007	0,008	0,009
SecondaryRoadGenerator	0,034	0,812	1,575
BlockCalculator	0,019	0,651	9,865
LotCalculator	0,008	0,040	0,136
BuildingGenerator	0,035	0,426	1,369
Gesamtzeit	0.117	1,976	13,017
Gebäude gebaut	84	1179	4911

(CPU: Intel Core i5-8250U 1.6GHz/1.8GHz RAM: 8GB)

Tabelle 1: Laufzeit der Pipeline-Stufen

Kleinere bis mittlerer Städte benötigen mit bis zu 2 Sekunden Ausführungszeit nicht sonderlich viel Zeit zu Berechnung. Bei größeren Städten steigt durch das umfangreiche sekundäre Straßennetz insbesondere die Zeit, die für die Berechnung der Häuserblocks benötigt wird, sehr stark an.

4.6 Steuerung

Die Steuerungsmöglichkeiten der Anwendung sind identisch mit den zu nutzenden Input-Daten. Durch die Roadmap und die Wahl der Gebietstypen während der Laufzeit ist es möglich, die Ergebnisse im Voraus zu steuern. Zudem können mit der Abänderung der Seeds unterschiedliche Ergebnisse forciert werde. Der Nutzer hat allerdings keinen Einfluss darauf, wie sich das Ergebnis verändert.
Nach der Generierung der Stadt gibt es keine Möglichkeit, die Stadt während der Laufzeit dynamisch anzupassen.

4.7 Echtzeit

Die erzeugte Stadt kann in Echtzeit erkundet werden. Es wurden keine Renderoptimierungen wie verschiedene Level of Detail getroffen.
Die Objekte sind zwar alle polygonarm, doch bei einer 100x100 Stadt mit allein etwa 5000 Gebäuden fällt die Bildrate je nach Hardware deutlich unter Echtzeitniveau.

4.8 Analyse

Die für die Bachelorarbeit entwickelte Anwendung erzeugt ausgehend von simplem Nutzer-Input eine fiktive, dreidimensionale Stadt. Ausgehend von der Zielsetzung sind die wichtigsten Kriterien der Realismus, die Variation sowie der Input.

Der Realismus und die Ähnlichkeit zu realen Städten ist nach heutigen Standards in der Unterhaltungsindustrie eher gering. Aufgrund der statischen Straßenverläufe können hauptsächlich Städte mit schachbrettartigen Straßenverläufen angenähert werden. Für authentischere und variantenreichere Verläufe allerdings müsste unter anderem eine freie Richtungswahl der Straßen ergänzt werden. Für dynamischere Straßensegmente könnten Knoten ähnlich wie in *CityGen* lediglich als Kontrollpunkte für Spline-artige Straßenverläufe dienen. In diesem Fall müssten allerdings viele Aspekte des Grundgerüstes mit verändert werden. Alternativ würde eine Erweiterung der globalen Parameter um Faktoren wie Straßenlänge oder Varianz dazu beitragen, dass mehr Vorgaben zur Straßenentwicklung definiert werden könnten.
Die eingeschränkte Vielfalt der Gebäude ist spätestens bei genauem Blick

sichtbar. Die prozedurale Generierung der Gebäudeformen und Fassaden wäre eine mögliche Erweiterung, um für eine größere Variation und einen höheren Detailgrad zu sorgen.
Für ein deutlich stimmigeres Stadtbild würde zudem ein fließender Übergang zwischen den Gebietstypen sorgen, beispielsweise durch Interpolation zwischen den Zentren der Nachbarschaften. Die strikte Begrenzung durch Verkehrswege müsste dafür aufgelockert werden.

5 Fazit

In dieser Arbeit wurden verschiedene Ansätze zur prozeduralen Generierung von Städten vorgestellt. Diese reichen von generativen Grammatiken, über Simulationen bis hin zu der Integration von Datensätzen.
Für die im Rahmen der Arbeit entwickelte Anwendung wurde die Nutzung einer Grammatik mit Ansätzen der Datenintegration gewählt. Orientiert wurde sich insbesondere an den Anwendungen *CityEngine* und *CityGen*.

Das erste Kernelement bei der Generierung einer Stadt ist das Aufstellen des Verkehrsnetzes und dabei vor allem des Straßennetzes. Die dafür ursprünglich geplante Nutzung eines L-Systems im Stil von Parish und Müller erwies sich als ein eher umständlicher Formalismus um das parallele Wachstum eines Straßennetzes zu beschreiben. Stattdessen konnte es durch einen vereinfachten Algorithmus, der schrittweise neue Straßensegmente in eine Liste anhängt und auf Platzierbarkeit prüft, implementiert werden. Wichtige Aspekte sind zum einen die Selbst-Sensitivität und zum anderen die Integration globaler Parameter, wie die Abzweigungshäufigkeit.
Elementar für die Weiterverarbeitung des Straßennetzes ist dessen Beschreibung durch einen planaren Graphen mit Knotenpunkten und verbindenden Kanten. Durch diesen können umschlossene Bereiche, z.B. Nachbarschaften und insbesondere Häuserblocks, berechnet werden. Letztere sind der Ausgangspunkt für die Generierung der Gebäude, welche das zweite Kernelement für die Stadtgenerierung darstellen. Durch das Aufteilen der Häuserblocks in Unterregionen entstehen Grundstücke als Grundlage für Position und Form der Gebäude.

Die Algorithmen eigneten sich gut um effizient umfangreiche Straßensysteme generieren zu lassen, dieses strukturell zu erfassen und daraus die Stadt an passenden Stellen zu bebauen. Auf Grenzen stieß die Straßennetzgenerierung bei den Punkten Realismus und Variation. Vor allem die Beschränkung auf gerade, quadratische Straßensegmente limitiert den Variantenreichtum an Straßenverläufen, den man in vielen organisch gewachsenen Städten vorfindet. Hinsichtlich der Bebauung ließe sich durch die Ergänzung fließender Übergänge zwischen den Stadtzonen und aufwendiger modellierten Gebäuden ein stimmigeres Stadtbild erzeugen.

Literatur

[1] SimCity (1989). https://www.gamepressure.com/games/view_screen.asp?ID=25517. abgerufen: 8. Oktober 2019.

[2] SimCity 2000 (1993). https://www.giga.de/spiele/sim-city/news/sim-city-2000-jetzt-vollkommen-kostenlos-bei-origin/. abgerufen: 8. Oktober 2019.

[3] SimCity (2013). http://simcitizens.com/simcity-2013-the-life-of-a-sim/. abgerufen: 8. Oktober 2019.

[4] Dominik HOFMANN and Armin HELLER. Prozedurale 3D-Stadtmodellierung mit der ESRI CityEngine. *STROBL, J., T. BLASCHKE, G. GRIESEBNER (HRSG., 2013): Angewandte Geoinformatik*, 2013.

[5] David S Ebert and F Kenton Musgrave. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

[6] George Kelly and H McCABE. An interactive system for procedural city generation. *Institute of Technology Blanchardstown*, 25, 2008.

[7] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. Graphical modeling using l-systems. In *The Algorithmic Beauty of Plants*, pages 1–50. Springer, 1990.

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

[9] Kevin Lynch. *The image of the city*, volume 11. MIT press, 1960.

[10] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[11] Joon-Seok Kim, Hamdi Kavak, and Andrew Crooks. Procedural city generation beyond game development. *SIGSPATIAL Special*, 10(2):34–41, 2018.

[12] George Kelly and Hugh McCabe. A survey of procedural techniques for city generation. *The ITB Journal*, 7(2):5, 2006.

[13] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[14] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 351–358. ACM, 1994.

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

[16] David Eberly. Constructing a cycle basis for a planar graph. *Geometric Tools*, 2016.

[17] Sean Barrett. L-Systems Considered Harmful. `http://nothings.org/gamedev/l_systems.html`, 2007-2009. Stand: 8. Oktober 2019.

[18] Los Angeles Downtown in Google Maps. `https://www.google.de/maps/place/Los+Angeles,+Kalifornien,+USA/@34.0780617,-118.2714806,404a,35y,147.56h,73.07t/data=!3m1!1e3!4m5!3m4!1s0x80c2c75ddc27da13:0xe22fdf6f254608f4!8m2!3d34.0522342!4d-118.2436849`. abgerufen: 10. Oktober 2019.