UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Entwicklung einer Schnittstelle zur echtzeitfähigen Generierung futuristischer Städte und Straßennetze durch prozedurale Algorithmen

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Marcus Goitowski

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller Institut für Computervisualistik, AG Computergraphik
Zweitgutachter:	Dipl.-Inform. D. Grüntjens Arbeitsgruppe Computergrafik

Koblenz, im September 2010

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.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Diplomarbeit

Marcus Goitowski

(Matr.-Nr. 205210075)

Thema: **Entwicklung einer Schnittstelle zur echtzeitfähigen Generierung futuristischer Städte und Straßennetze durch prozedurale Algorithmen**

Durch die rapide Weiterentwicklung in der Multimediaindustrie hat sich in den letzten Jahren u. a. eine hohe Verbreitung von HD-fähigen Bildschirmen gezeigt. Hierdurch entstanden nicht nur bei der Videospielentwicklung sondern auch in der Animationsindustrie völlig neue Schwierigkeiten. Die Erwartung des Anwenders an die grafische Repräsentation ist zeitgleich mit der gestiegenen Hardwareleistung gewachsen, so dass die Industrie einen enorm höheren Entwicklungsaufwand zu bewältigen hat. Der Autorenprozess ist aktuell so umfangreich, dass viele Computerspieleschmieden versuchen dem Problem durch eine Reduzierung des Umfangs des Produktes entgegenzuwirken. Hierbei sind in den letzten Jahren viele Spiele entstanden, die eine sehr lineare und statische Spielwelt besitzen. Dem Nutzer wird so weder ein Freiraum eingeräumt noch ein Wiederspielwert geboten.

Ziel dieser Arbeit ist es Techniken zu zeigen, die es ermöglichen in der heutigen Zeit ohne sonderlichen Mehraufwand große und authentische Spielwelten zu schaffen. Die Verlagerung vom Modellierungsprozess auf Algorithmen zur prozeduralen und dynamischen Inhaltserzeugung (PCG) ist nötig. Es soll am Beispiel von futuristischen Städten gezeigt werden, dass durch diese Verlagerung glaubhafte Welten geschaffen werden können, so dass der Gestaltungsaufwand eines Levels sich wieder in einem annehmbaren Maß bewegt.

Schwerpunkte dieser Arbeit sind:

1. Identifizierung und Umsetzung geeigneter Verfahren
    a. zur Generierung glaubhafter Straßennetzmuster
    b. zur prozeduralen Gebäudegenerierung
2. Erstellung einer Programmschnittstelle zum Datenaustausch
    a. es können Parameter zur Beeinflussung des Ergebnisses angegeben werden
    b. als Ausgabe wird eine Beschreibung der generierten Gebäude und Straßen geliefert
3. softwaretechnischer Entwurf der zu implementierenden Schnittstelle
4. Erstellung einer Anwendung zur Darstellung des generierten Materials
5. Evaluierung der Programmschnittstelle mittels der darstellenden Anwendung und Dokumentation der Ergebnisse

Koblenz, den 01.04.2010

- Prof. Dr. Stefan Müller-

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

1 Einleitung

1.1 Motivation

Durch einen rapiden Fortschritt in der Multimediaindustrie haben sich hochauflösende Anzeigemedien immer mehr verbreitet. Hochauflösende Fernseher und Computerbildschirme wollen mit entsprechend detailreichen Materialien befüllt werden. Neben der gestochen scharfen Fußballübertragung von PayTV-Sendern werden auch Computerspiele und Kinofilme immer detaillierter.

Abgesehen von aktuellen Kassenschlagern wie „Inception“ haben bereits in der Vergangenheit Filmautoren imposante Welten, insbesondere Stadtlandschaften, geschaffen. Im Film „Das fünfte Element“ wird eine Stadt präsentiert, die aus Wolkenkratzern besteht und ein vielschichtiges Verkehrssystem besitzt. Der Verkehr bewegt sich auf mehreren übereinander liegenden Schichten, die über senkrechte Straßenkreuzungen gewechselt werden können. Dies geschieht ohne physikalisch existente Straßen allein durch schwebende Fahrzeuge. Gebäude besitzen eine immense Höhe, so dass Straßenbahnen an ihnen entlang verlaufen.

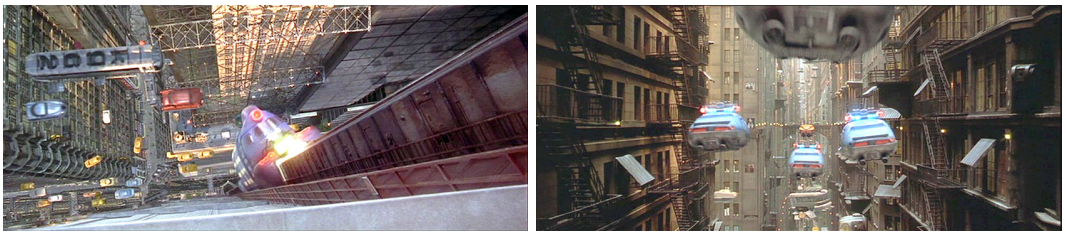


Abb. 1.1.1: Bilder aus „Das fünfte Element“ zeigen eine futuristische Stadt

Im Filmklassiker „Star Wars“ wird eine weniger in die Höhe orientierte Variante einer futuristischen Großstadt gezeigt. Vielmehr handelt es sich um einen Stadtplaneten; die Stadt Coruscant umgibt den gesamten Planeten. Die Stadt besteht aus einem homogenen Bild an Hochhäusern. Sie wird von einigen Prunkbauten durchzogen, jedoch ergibt sich ein Muster sehr ähnlich wirkender Gebäude.

Abb. 1.1.2: Konzeptzeichnungen aus „The Art of Star Wars, Episode III“ des Stadtplaneten Coruscant

Selbst fünf Jahre nach der Veröffentlichung von „Star Wars - Episode III“ werden Filme mit mäßig realistischem Stadtbild produziert. So kann im zuvor erwähnten Kinoschlager „Inception“ eine Traumstadt losgelöst von physikalischen Bedingungen bestaunt werden, jedoch gleicht ein Gebäude dem anderen. Dies könnte kontextbedingt jedoch durch den fehlenden Einfallsreichtum des Stadtschöpfers begründet werden. Denn auf Werbeplakaten des Filmes ist eine wahrscheinlich prozedural erzeugte Stadt zu sehen. In das bestehende Stadtbild sind in Form des Filmtitels eingefügte Gebäude enthalten.

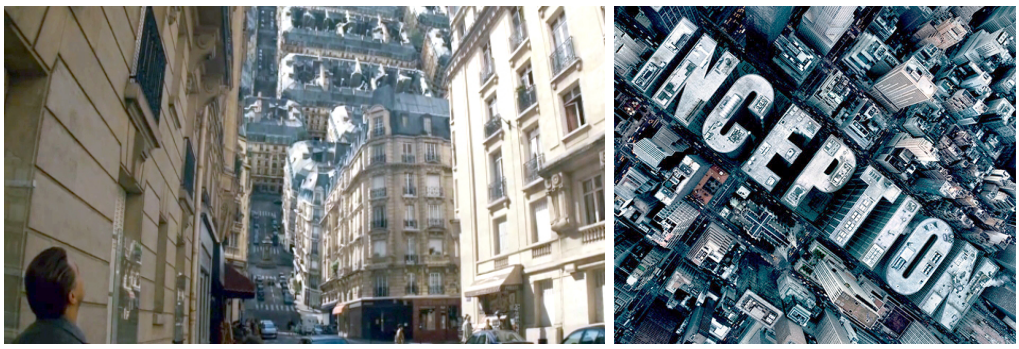


Abb. 1.1.3: Bild und Werbeplakat des Filmes „Inception“

Zu sehen ist, dass sich die Ergebnisse in der Filmindustrie Jahr um Jahr optisch deutlich verbessern und ansprechender werden. Das Optimum ist jedoch noch lange nicht erreicht.

Viel kritischer gestaltet sich der Fortschritt in der Computeranwendungs und -spieleindustrie, da hier das Kriterium der Echtzeitfähigkeit neben dem Grad des optischen Realismuses im Vordergrund steht. Filme können grafisch ein erstaunliches Ergebnis bieten, da die Erstellung solcher Bilder in einem deutlich weitläufigeren Zeitrahmen erstellt werden können.

Ein interessanter Entwicklungsfortschritt kann bei dreidimensionalen Hochhauswelten in Computerspielen des Spider-Man-Franchises (Abbildung 1.1.4 auf der nächsten Seite) festgestellt werden. Ist die Hochhauswelt in einem der ersten Arcade-Titel aus den 90er-Jahren (linkes Bild) noch sehr minimalistisch durch grobe Formen angedeutet, ändert sich das Aussehen bis in das Jahr 2000 in einem u.a. auf der Playstation 1 erschienenen

Spiel (mittleres Bild) erheblich zum Positiven. Der Titel „Spider-Man: Web of Shadows“ aus dem Jahre 2008 (rechtes Bild) zeigt wiederum einen großen Fortschritt im optischen Erscheinungsbild, jedoch trotz des Fortschrittes nicht dem Stand der Zeit entsprechend. Kritisiert wird neben dem nicht zeitgerechten Aussehen auch die Häufigkeit, mit der sich Gebäude in der Welt wiederholen. Die Ursache besteht, wie bei vielen Spielen mit offener Welt, im Mangel der nötigen Zeit um eine überzeugende und abwechslungsreiche Großstadt zu kreieren.

Abb. 1.1.4: Bilder aus verschiedenen Spielen der Spider-Man-Reihe

Auch renommierte Hersteller der Spieleindustrie kämpfen mit dem zunehmenden Entwicklungsaufwand, der durch hochauflösende Medien zustande gekommen ist. Der Publisher „Square Enix“ der Final Fantasy Serie hat ein Magazin (Enix, 2010) veröffentlicht. In diesem äußern sich Entwickler des für den Teil „Final Fantasy XIII“ zuständigen Studios zur Produktion des Spiels. Das Spiel wird - trotz einer hervorragenden Grafik - oft wegen seiner sehr linearen und schlauchartigen Levelstruktur und dem Fehlen von Städten kritisiert. Final-Fantasy-Titel sind normalerweise für ihre großen und offenen Spielewelten bekannt, was bei dem dreizehnten Teil zu Verwunderung führt. Begründung findet dies in einer Äußerung eines Entwicklers.

> „Die Linearität des Spieles kam dadurch, dass das Darstellen von Städten und so weiter, wie wir es bisher getan haben, in "HD" auf einer Konsole unmöglich ist - es war zu viel Arbeit.“[1]

Diese Problematik zeigt sich somit nicht nur bei mittelmäßigen Spieletiteln sondern auch in hochrangigen Produktionen. Der mittlerweile sehr umfangreiche Autorenprozess der Spielewelten veranlasst viele Computerspieleschmieden dazu, den Umfang ihres Produktes zu reduzieren. Dem Nutzer wird so jedoch der spielerische Freiraum genommen und ein Wiederspielwert ist kaum noch gegeben. Da der eingeschlagene Weg vieler Firmen nicht im Interesse des Anwenders sein kann, wird dies zum Anlass genommen, hierfür einen Lösungsansatz zu erarbeiten.

1.2 Zielsetzung

Es ist zu erkennen, dass viele Firmen mit dem wachsenden Entwicklungsaufwand zunehmend Probleme haben. Ziel dieser Arbeit ist es, Techniken zu zeigen, die es ermöglichen,

[1] „The game’s linearity was just because depicting towns and so on like we did before was impossible to do on an “HD” console – it was too much work.“

in der heutigen Zeit ohne besonderen Mehraufwand große und authentische Spielewelten zu schaffen. Nicht nur das räumliche Ausmaß der Welt soll ohne zusätzlichen Aufwand wachsen, sondern auch der Abwechslungsreichtum und somit Wiederspielwert soll ansteigen. All dies soll am Szenario einer futuristischen Großstadt gezeigt werden.

Bereits die US-amerikanische Spielefirma Blizzard Entertainment Incorporated hat mit der Spielereihe Diablo gezeigt, dass es möglich ist, authentische Welten aus einem kleinen Vorrat von Bausteinen zu erzeugen, so dass diese Welten qualitativ hochwertig sind und einen enormen Wiederspielwert besitzen.

Abb. 1.2.1: Bilder aus Diablo I, II und III

Es werden hierbei Algorithmen zur prozeduralen und dynamischen Inhaltserzeugung (PCG[2]) verwendet. Diablo verwendet diese jedoch nur zur Erzeugung zweidimensionaler Welten. Bei diesen werden größtenteils 2D-Texturen, die Bausteine der Welt darstellen, aneinandergesetzt. Werden dreidimensionale Welten gewünscht, ist eine entsprechende Erweiterung der 2D-Algorithmen durchzuführen. Anstelle zweidimensionaler Texturen werden dreidimensionale Modelle benötigt, die in die Welt eingesetzt werden oder zu einem Ganzen kombiniert werden. Durch Letzteres soll eine Verlagerung des Modellierungsaufwandes auf prozedurale Algorithmen geschaffen werden, so dass sich der Gestaltungsaufwand eines Levels wieder in einem annehmbaren Maß bewegt.

Akte (Level) aus Diablo werden vor deren Beginn einmalig dynamisch generiert und dann bis zum Ende nicht mehr verändert. Neben diesem Verfahren soll ein vollständig prozeduraler Ansatz entwickelt werden, der das Level in Echtzeit entstehen lässt und erweitert.

1.3 Aufteilung der Arbeit

Die Kapitel der Arbeit sind inhaltlich aufeinander aufbauend und wie folgt gegliedert:

In Kapitel 2 auf Seite 7 wird auf elementare Techniken und Grundbegriffe verschiedener Themenbereiche eingegangen, die für die Umsetzung von Bedeutung sind.

Kapitel 3 auf Seite 29 befasst sich mit dem aktuellen Fortschritt der auf dem Markt vorhandenen Techniken. Es werden Algorithmen und Vorgehen erläutert, die zum Zeitpunkt dieser Arbeit bereits existiert haben.

[2]PCG: Procedural Content Generation

In Kapitel 4 auf Seite 41 wird kurz auf die Entscheidungen, die für die Entwicklung des Systems getroffen worden sind, eingegangen und die eigentliche Umsetzung des Systems, insbesondere des Backends, im Detail behandelt.

Kapitel 5 auf Seite 79 befasst sich detailliert mit der Umsetzung des Frontends und den damit verbundenen Aufgaben. Das Frontend ist speziell an das Backend angepasst worden. Jedoch sind Anpassungen am Backend erforderlich gewesen, die in diesem Kapitel behandelt werden.

In Kapitel 6 auf Seite 89 findet die Bewertung des Backends statt.

Im letzten Kapitel wird ein Ausblick auf mögliche Erweiterungen und Verbesserungsmöglichkeiten im Front- und Backend gegeben sowie ein Fazit zu prozeduralen Techniken der Inhaltserzeugung gezogen.

Kapitel 2

2 Grundlagen

Dieses Kapitel der Arbeit befasst sich mit Begriffserklärungen aus verschiedenen Bereichen. Da es sich um eine fachbereichsübergreifende Thematik handelt, stammen die Erläuterungen aus verschiedenen Bereichen wie der Architektur, Informatik, Mathematik und Biologie. Neben Begriffen werden auch Verfahren aus diesen Bereichen erklärt. Die Techniken aus der Informatik lassen sich zudem in die Bereiche der Theoretischen Informatik und Computergrafik aufteilen. Die Anwendungsbereiche der Techniken werden in Kapitel 4 auf Seite 41 beschrieben.

2.1 Grundlagen der Mathematik

Ein großer Bestandteil der Arbeit besteht in der Entwicklung oder Einbindung verschiedener Geometriefunktionen. Diese werden für diverse Berechnungen im Zwei- und Dreidimensionalen benötigt und sind so konzipiert, dass sie aufeinander abgestimmt sind. Hierzu zählen u.a. Funktionen für Linien, Vektoren, Matrizen, Polygone, Rechtecke und Volumen. Aus diesen werden die Wichtigsten präsentiert.

2.1.1 Schnitttest mit Geradensegmenten

An vielen Stellen wird die Berechnung von Schnittpunkten bzw. die eines Schnittergebnisses zwischen zwei Geraden im 2D benötigt. Für das entwickelte System wird eine spezielle Form der Schnittberechnung benötigt. So müssen Segmente und keine Geraden geschnitten werden, es muss erkannt werden, ob zwei Segmente zueinander parallel oder koinzident[1] sind und ob eine Koinzidenz im gültigen Segmentintervall auftritt[2]. Außerdem muss festlegbar sein, ob ein Test inklusiv oder exklusiv der Start- und Endpunkte stattfinden soll. Aus diesen Kriterien ergeben sich einige Schnittmöglichkeiten:

- schneidet nicht (NOT_INTERSECTING)

[1] koinzident: deckungsgleich

[2] Koinzidenz ist bei Segmenten mit identischen Geraden gegeben, jedoch ist diese bei hintereinanderliegenden Segmenten auf einer Geraden nicht gewünscht.

- parallel (PARALLEL)
- schneidet (INTERSECTING)
- koinzident (COINCIDENT)

Die eigentliche Berechnung erfolgt über die zwei Geraden in Punkt-Richtungsform. Seien $\overline{P_0P_1}$ und $\overline{P_2P_3}$ zwei Geraden, dann lassen sich diese wie folgt ausdrücken:

$$\begin{aligned} G_0 &= P_0 + r \cdot (P_1 - P_0) \\ G_1 &= P_2 + s \cdot (P_2 - P_1) \end{aligned} \tag{2.1.1}$$

Um den Schnittpunkt der Gerade zu erhalten, werden die Gleichungen gleichgesetzt ($G_0 = G_1$).

$$P_0 + r \cdot (P_1 - P_0) = P_2 + s \cdot (P_2 - P_1)$$

Zerlegt man die Gleichung in x- und y-Komponenten der Punkte und löst die Gleichungen nach s bzw. r auf, erhält man:

$$\begin{aligned} d &= (y_3 - y_2)(x_1 - x_0) - (x_3 - x_2)(y_1 - y_0) \\ r &= \frac{(x_3 - x_2)(y_0 - y_2) - (y_3 - y_2)(x_0 - x_2)}{d} \\ s &= \frac{(x_1 - x_0)(y_0 - y_2) - (y_1 - y_0)(x_0 - x_2)}{d} \end{aligned}$$

Wird r bzw. s in 2.1.1 eingesetzt, liefert dies den Schnittpunkt der Geraden. Um jedoch die anfangs erwähnten Schnittmöglichkeiten zu erkennen, sind die Teilergebnisse zu überprüfen. Ist der Nenner d nahe null, so sind die Geraden parallel. Sind zusätzlich beide Zähler der Quotienten null, so sind die Geraden koinzident. Um festzustellen, ob die Segmente ebenfalls koinzident sind, sind Start- und Endpunkte darauf zu überprüfen, ob diese gegenseitig im Intervall [0..1] liegen. Falls dies zutrifft, sind die Segmente ebenfalls koinzident, ansonsten gibt es keinen Schnitt. Ist d nicht nahe null, ist zu prüfen, ob r und s jeweils im Intervall [0..1] (im exklusiven Fall im Intervall]0..1[) liegen. Ist das Kriterium erfüllt, ist ein Schnitt vorhanden, ansonsten nicht.

2.1.2 Berechnung des Upvektors

An einigen Stellen wird der Upvektor einer im 3D-Raum liegenden Kante benötigt, um beispielsweise die Spielfigur korrekt positionieren zu können. Das Problem besteht hier darin, dass ein eindeutiger Upvektor im Dreidimensionalen nur für eine Ebene, die von zwei Vektoren aufgespannt wird, bestimmt werden kann. Eine einzelne Kante kann jedoch keine Ebene aufspannen. Um den fehlenden Vektor zu kompensieren, wird als initiale Ausrichtung des Upvektors die y-Achse angenommen. Zu diesem und der

Kantenrichtung wird ein orthogonaler Vektor ($\overrightarrow{l}$) bestimmt. Anschließend kann über diesen Vektor der echte Upvektor ($\overrightarrow{u}$) errechnet werden.

$$\begin{aligned}\overrightarrow{y} &= (0,1,0)\\ \overrightarrow{d} &= \|P_2 - P_1\|\\ \overrightarrow{l} &= \overrightarrow{y} \times \overrightarrow{d}\\ \overrightarrow{u} &= \overrightarrow{d} \times \overrightarrow{l}\end{aligned}$$

Die Annahme der y-Achse als initiale Ebenennormale führt jedoch dazu, dass kein Upvektor einer senkrechten Strecke ermittelt werden kann, da $\overrightarrow{l}$ hierbei die Länge 0 besäße und somit nur ein beliebiger Vektor in der XZ-Ebene entstehen würde.

2.1.3 Fläche eines Polygons

In der späteren Umsetzung werden Polygone aus dreidimensionalen Zyklen (*Cycle3D*) erzeugt. Für eine weitere Bearbeitung wird aus diesem 3D-Zyklus ein zweidimensionales Polygon (*Shape2D*) erstellt, das der Projektion auf die XZ-Ebene der dreidimensionalen Form des Zyklus entspricht. Dieses Polygon kann verwendet werden, um die Fläche, die es einschließt, zu ermitteln. Die Fläche wird mittels der Gaußschen Trapezformel errechnet.

$$\begin{aligned}P_i &= (x_i, y_i)\\ 2A &= \left|\sum_{i=1}^{n}(y_i + y_{i+1}) \cdot (x_i - x_{i+1})\right| = \left|\sum_{i=1}^{n}(x_i + x_{i+1}) \cdot (y_{i+1} - y_i)\right|\end{aligned}$$

Der Index i kann größer werden als die Anzahl der Punkte, die den Zyklus bilden. Ist dies der Fall, wird der Index auf den Anfang umgebrochen (Modulo).

2.1.4 Punkt im Polygon

Ein wichtiger Bestandteil des Systems ist der Test, ob ein Punkt innerhalb eines Polygons liegt (Haines, 1994). Hierzu gibt es mehrere Möglichkeiten, die sich in ihrer Geschwindigkeit und teilweise in der Unterstützung von Grenzfällen unterscheiden.

Um diesen Test auszuführen, ist anfangs der Winkelsummierungstest verwendet worden. Dieser ist sehr einfach umzusetzen, jedoch vergleichsweise langsam. Bei diesem Test wird der zu prüfende Punkt als Zentrum angenommen und es werden die Winkel, die dieser zu den Kantenanfangspunkten und -endpunkten bildet, summiert. Liegt die Winkelsumme bei 0° bzw. einem Vielfachen davon (Fall a aus Abbildung 2.1.1 auf der nächsten Seite), liegt der Punkt im Polygon. Ist die Winkelsumme eine andere (Fall b), liegt der Punkt außerhalb.

Mit der Verwendung eines Strahlentests („even-odd-test“) kann eine Beschleunigung

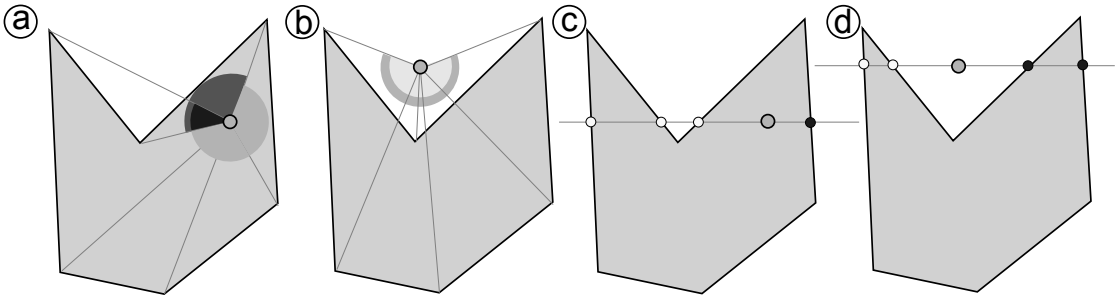


Abb. 2.1.1: Testen ob ein Punkt innerhalb eines Polygons liegt

erreicht werden. Bei dieser Methode wird ein Strahl durch den zu testenden Punkt gelegt. Anschließend werden die Anzahl der Schnittpunkte links und rechts dieses Punktes berechnet und gezählt. Ist die Anzahl auf beiden Seiten ungerade (Fall c), liegt der Punkt innerhalb, ansonsten außerhalb (Fall d).

Teilweise zu beachten sind Grenzfälle, bei denen der zu prüfende Punkt auf dem Umriss liegt oder innerhalb eines Loches im Polygon.

Segmente und Polygone

Neben dem Punkt-im-Polygon-Test und dem allgemeinen Schnitttest ist eine zusätzliche Kombination dieser Algorithmen nötig, denn ein Segment muss nicht zwangsweise in einem Polygon enden oder beginnen. Es sind mehrere Fälle zu unterscheiden, die durch einen der Tests oder beide erkannt werden können.

- Das Segment liegt vollständig außerhalb des Polygons
- Das Segment beginnt oder endet im Polygon
- Das Segment durchquert das Polygon, so dass Start- und Endpunkt außerhalb von diesem liegen
- Das Segment liegt vollständig im Polygon
- Das Segment liegt teilweise im Polygon und verlässt dieses teilweise (konkaves Polygon)

Nur durch beide Tests sind die Fälle eindeutig zuordenbar und es kann so auf diese reagiert werden.

2.2 Grundlagen der Informatik

Die benötigten Grundlagen der Informatik befassen sich im Wesentlichen mit der Manipulation und Erzeugung von Geometrienetzen (Meshes) sowie mit Graphentheorie. Zudem werden einige Begriffe erklärt.

2.2.1 Clipping

Bei der Erzeugung von prozeduralen Inhalten können Polygone entstehen, die durch ihre Form andere Polygone schneiden. Zur Lösung dieses Problems gibt es viele Ansätze, welche jedoch in der Regel auf konvexen Polygonen basieren. Der Algorithmus von Sutherland-Hodgman schneidet ein beliebiges Polygon an einem beliebigen konvexen Polygon (Viewport). Da dieses Verfahren jedoch nicht zwei konkave Polygone miteinander schneiden kann, ist ein anderer Algorithmus nötig.

Als ein geeignetes Verfahren hat sich der Algorithmus von Weiler-Atherton (Agu) herausgestellt, da dieser sowohl mit konvexen als auch mit konkaven Polygonen zurechtkommt. Eine Bibliothek, die u.a. diesen Algorithmus umsetzt, ist die General Polygon Clipper Library (University of Manchester).

Sutherland-Hodgman-Algorithmus

Mit dem Algorithmus von Sutherland-Hodgman (Sutherland/Hodgman, 1974) können beliebige Polygone in der einfachen Variante gegen ein Rechteck und in der erweiterten Variante an einen beliebigen konkaven Polygonen geclippt werden. Im Folgenden wird die erweiterte Variante beschrieben.

Für das Clipping werden die Kanten der zu clippenden Polygone in vier Arten unterschieden.

1. Kanten, die vollständig innerhalb des Clippingfensters (CLIP) liegen
2. Kanten, die vollständig außerhalb von CLIP liegen
3. Kanten, die CLIP verlassen
4. Kanten, die in CLIP eintreten

Der Algorithmus durchläuft alle Kanten des zu clippenden Polygons (POLY) und überprüft für diese jeweils, welches der vier Kriterien zutrifft. Liegt die Kante innerhalb, wird der Endpunkt dieser Kante in das Ergebnis übernommen, liegt sie vollständig außerhalb wird sie nicht übernommen. Sollte die Kante jedoch CLIP durchqueren, dann werden die Schnittpunkte mit CLIP berechnet und übernommen. Tritt der Fall ein, dass eine Kante in CLIP ein- bzw. austritt, dann wird der Schnittpunkt mit CLIP berechnet und in das Ergebnis übernommen.

Schwächen des Algorithmus sind zum Einen die Voraussetzung eines konkaven Clipfensters und dass das Clipping von konvexen Polygonen zu dünnen entarteten Randkanten (siehe Abbildung 2.2.1 auf der following Seite) führen kann.

Weiler-Atherton-Algorithmus

Der Vorteil dieses Algorithmus (Weiler/Atherton, 1977) gegenüber dem von Sutherland-Hodgman besteht in der Unterstützung eines konkaven Clipfensters und eines konkaven

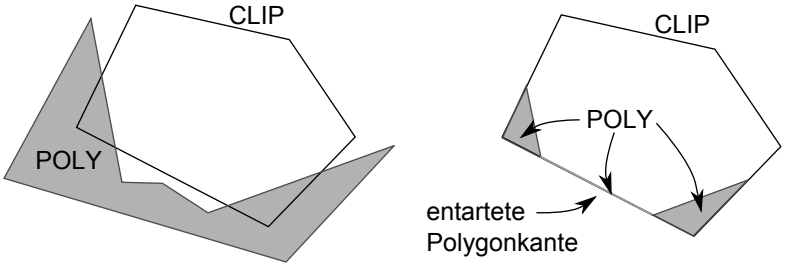


Abb. 2.2.1: Sutherland-Hodgman-Algorithmus mit entarteter Randkante

Clippolygons. Zudem erstellt der Algorithmus, falls nötig, mehrere unverbundene Polygone. Das geclippte Polygon wird im Folgenden mit POLY und das clippende Polygon als CLIP bezeichnet.

Der Algorithmus arbeitet mit zwei Listen, jeweils bestehend aus den Ecken von POLY bzw. CLIP. Jede Ecke wird markiert, ob diese in Abhängigkeit des anderen Polygons innerhalb oder außerhalb liegt. Anschließend werden alle Schnittpunkte der Polygone untereinander berechnet und in die Listen einsortiert. Die Schnittpunkte in den Listen, die dem gleichen Schnitt entsprechen, verketten die beiden Listen miteinander. Aus den nach innen verlaufenden Schnitten der POLY-Liste wird eine weitere Liste (ENTERING) erzeugt. Für jeden Schnittpunkt S in ENTERING wird nun das geclippte Ergebnispolygon gesucht. Hierfür wird jeweils S als Startpunkt gewählt. Der erste Folgeknoten von S liegt auf POLY. Es wird POLY solange durchlaufen, bis ein weiterer Schnittpunkt gefunden worden ist. An diesem Schnittpunkt wird auf das andere Polygon übergewechselt (POLY bzw. CLIP) und die Folgeecke vom Schnittpunkt aus gewählt, die sich im Uhrzeigersinn auf dem aktiven Polygon befindet. In der aktiven Liste wird solange der Folgeknoten dem Ergebnispolygon angefügt bis wieder ein Schnittpunkt gefunden worden ist und der Prozess auf der anderen Liste von vorne beginnt. Ist der gefundene Schnittpunkt der Startpunkt, kann das Polygon geschlossen werden (siehe Beispiel in Abbildung 2.2.2 auf der nächsten Seite sowie Tabelle 2.2.1 auf der nächsten Seite). Die ENTERING-Liste wird von allen passierten Schnittpunkten bereinigt. Im Fall von mindestens einem konkaven Polygon kann mehr als ein geclipptes Polygon entstehen. In diesem Fall enthält ENTERING nach der Bereinigung noch weitere Schnittpunkte, die verfolgt werden müssen.

Wird anfangs kein Schnittpunkt gefunden, liegt POLY in CLIP oder umgekehrt. Es wird entsprechend POLY unverändert zurückgegeben oder POLY auf CLIP reduziert.

Eine unkritische Einschränkung des Weiler-Atherton-Algorithmus besteht darin, dass keine sich selbstschneidenden Polygone unterstützt werden.

Beispiel In Abbildung 2.2.2 auf der nächsten Seite werden zwei konkave Polygone gegeneinander geschnitten. Im ersten Teilbild ist der Ausgangszustand zu erkennen, der in eine Tabelle überführt werden kann (siehe Tabelle 2.2.1 auf der nächsten Seite). Im zweiten Teilbild sind die Schnittpunkte berechnet und diese verketten nun die beiden Polygone. Im letzten Teilbild ist der gefundene Pfad eingezeichnet und somit das geclippte

Polygon gefunden worden.

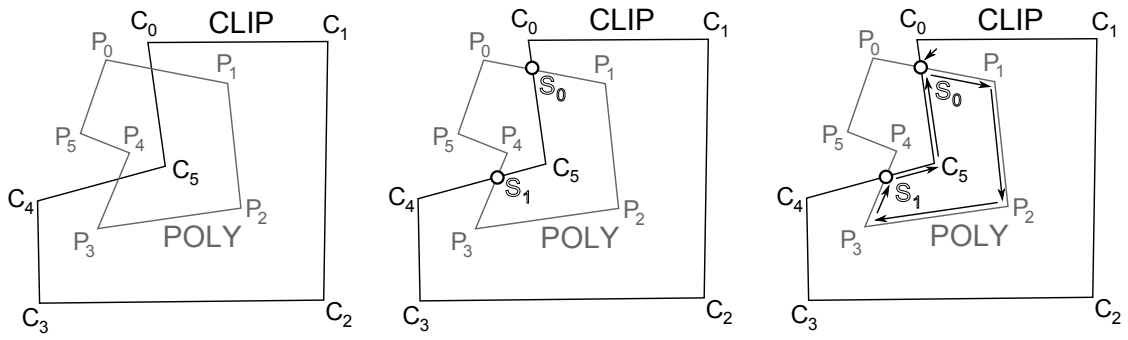


Abb. 2.2.2: Weiler-Atherton-Algorithmus am Beispiel konkaver Polygone

CLIP	POLY
C_0	P_0
C_1	P_1
C_2	P_2
C_3	P_3
C_4	P_4
C_5	P_5

CLIP		POLY
	S_0	
C_0		P_1
C_1		$\vdots$
C_2		P_2
C_3		$\vdots$
C_4		P_3
	S_1	
$\vdots$		P_4
C_5		P_5
$\vdots$		P_0

ENTERING
S_0

Suche (S_0)
S_0
P_1
P_2
P_3
S_1
C_5
S_0

Tabelle 2.2.1: Beispielhafte Auswertung mit Weiler-Atherton

2.2.2 Triangulation

Bei der Triangulation wird aus einer gegebenen Punktwolke ein Netz aus Dreiecken erzeugt. Bei der prozeduralen Erstellung von beispielsweise Hausdächern und Straßenkreuzungen ist es erforderlich aus einer generierten Kontur eine triangulierte Fläche zu erzeugen. Die Delaunay-Triangulation (Delaunay, 1934; Wikipedia, 2010b) ist hierfür ein einfaches und bewährtes Verfahren, zu dem es eine Vielzahl an Optimierungen gibt.

Delaunay-Triangulation

Bei diesem Verfahren wird aus einer gegebenen 2D-Punktwolke ein Dreiecksnetz mit speziellen Eigenschaften erzeugt.

Definition. Eine Delaunay-Triangulation einer Punktmenge P in einer Ebene ist gegeben, wenn es keinen Punkt aus P gibt, der innerhalb eines Umkreises um die Dreiecke der Triangulation liegt. (Umkreiskriterium)

Aus dieser Bedingung folgt, dass in den entstehenden Dreiecken der kleinste „Innenwinkel über alle Dreiecke maximiert“ wird. Dies ist für das spätere Rendern von Bedeutung, da der Rundungsfehler minimiert wird und so das Flackern zu spitzer Dreiecke reduziert wird.

Für die Umsetzung der Delaunay-Triangulation gibt es verschiedene Verfahren (u.a. Flip-Verfahren, Divide & Conquer, Sweep-Algorithmus), die sich in ihrer Laufzeit von $O\left(n^2\right)$ bis $O\left(n \log n\right)$ unterscheiden.

Bei dem Flip-Verfahren wird auf einem bereits trianguliertem Netz gearbeitet, das die Delaunay-Bedingung nicht erfüllen muss. Der Algorithmus überprüft anschließend für eine Kante, die zwei Dreiecke besitzt, ob das Umkreiskriterium erfüllt ist. Ist dieses nicht erfüllt, wird die Diagonale des betrachteten Vierecks, das von den zwei Dreiecken gebildet wird, getauscht („flip“). Anschließend werden die Nachbarkanten des zuvor betrachteten Vierecks überprüft. Nach dem Flippen der jeweiligen Diagonale ist gegeben, dass das Umkreiskriterium erfüllt ist (siehe z. Du/Hwang (1992)). Dieses Vorgehen ist jedoch nicht optimal, da zuerst ein Netz erzeugt werden muss und im schlimmsten Fall alle Kanten getauscht werden müssen.

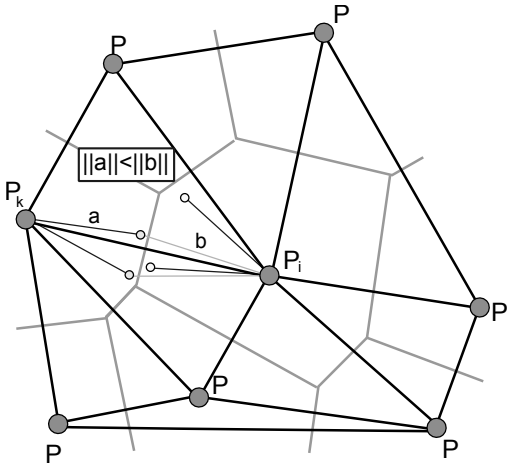


Abb. 2.2.3: Dualität des Voronoidiagramms und der Delaunaytriangulation

Ein schnelleres Vorgehen ist beispielsweise das Sweep-Line-Verfahren, bei dem ein Voronoidiagramm in einer Laufzeit von $O\left(n \log n\right)$ ermittelt werden kann. Ein Voronoidiagramm ist der duale Graph einer Delaunay-Triangulation und kann somit problemlos in die duale Form überführt werden.

Definition. Ein Voronoidiagramm besteht aus Regionen mit je einem Punkt P_i. Jede Region enthält alle Punkte, die zu P_i näher sind als zu allen anderen Punkten P_k.

Bei dem Sweep-Algorithmus wird eine Gerade über das gesamte Punktfeld bewegt. Wird diese Gerade bewegt, werden nach und nach die Punkte berührt und in eine Voronoi-Datenstruktur eingefügt. Um hierbei nicht permanent das bereits erstellte Voronoidiagramm auf Laufzeitkosten verändern zu müssen, kann ein Transformationsverfahren verwendet werden (Fortune, 1986; Lehner, 2002).

Um anschließend aus einem Voronoidiagramm eine Delaunay-Triangulation zu erhalten, werden die Zentren der Regionen so verbunden, dass zu jeder Regionskante eine orthogonale Gerade gebildet wird, die die entsprechenden benachbarten zwei Zentren miteinander verbindet.

Der Nachteil der Delaunay-Triangulation besteht darin, dass immer die konvexe Hülle erzeugt wird, was für den gegebenen Anwendungsfall nicht erwünscht ist. Da vor der Triangulation jedoch die gewünschte Hülle bekannt ist, ist es möglich die Triangulation anzupassen.

Constrained-Delaunay-Triangulation

Eine Erweiterung stellt die Constrained-Delaunay-Triangulation (CDT) da. Bei dieser werden Punkte oder Kanten als fest angenommen. Es entsteht bei der CDT keine im Gesamten gültige Delaunaytriangulation, jedoch mehrere an den festen Segmenten unterteilte Teildelaunaytriangulationen. Für unseren Anwendungsfall wird die konkave Hülle als „Constrained“ gesetzt, da diese vor der Triangulation bekannt ist. Durch die festen Kanten erweitert sich das Umkreiskriterium.

Definition. Es sei eine Punktmenge P und eine sich nicht schneidende Kantenmenge L gegeben, die aus Kanten zwischen Punkten aus P besteht. Eine CDT ist gegeben, wenn alle Dreiecke Teil der normalen Delaunay-Triangulation sind oder Punkte der Kanten aus L enthalten.

Die CDT wird gebildet, indem alle Kanten, die von den festen Kanten aus L geschnitten werden, aus der Triangulation entfernt werden. Anschließend werden die nicht vollständig vernetzten Bereiche links und rechts der festen Kante erneut unter Einhaltung der angepassten Definition trianguliert (Lehner, 2002).

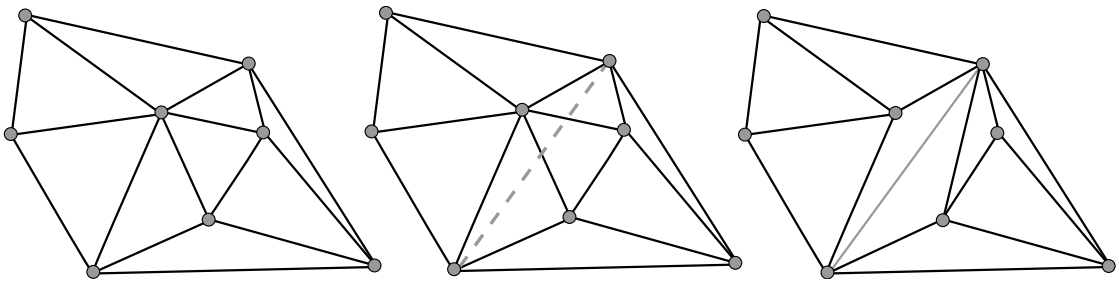


Abb. 2.2.4: Constrained Delaunay Triangulation

Um konkave Flächen mittels CDT zu triangulieren, wird eine angepasste Version der Bibliothek Triangle von Jonathan Richard Shewchuk (Shewchuk) verwendet.

2.2.3 Graph

Ein Graph G besteht aus Kanten (E) und Knoten (V). Die Knoten eines Graphen werden von den Kanten verbunden. Jeder Knoten kann mehrere Kanten zu anderen Knoten besitzen. Es ist zu unterscheiden zwischen gerichteten und ungerichteten Graphen. Bei gerichteten Graphen ist ein Übergang zwischen Knoten nur in vorgegebener Kantenrichtung möglich. Für die Darstellung des Straßennetzes wird ein ungerichteter Graph verwendet. Deswegen ist eine Bewegung entlang der Kanten in beide Richtungen möglich.

Zyklen

Während der Generierung einer Stadt müssen die bebaubaren Bereiche erkannt werden. Diese Bereiche, Parzellen, sind dabei die kleinsten Zyklen eines Graphen. Ein Zyklus besteht aus Knoten des Graphen und den Kanten, die diese Knoten untereinander verbinden. Keine Kante wird hierbei mehrmals passiert oder ausgelassen. Das gleiche gilt für die Knoten. Zudem muss der Startknoten gleich dem Endknoten sein. Zur Erkennung dieser Zyklen gibt es eine Reihe von Ansätzen, die in Abschnitt 4.3 auf Seite 59 beschrieben werden.

2.2.4 Back- und Frontend

Unter einem Backend versteht man ein abgeschlossenes System, das über eine definierte Schnittstelle Funktionalitäten einer vorgeschalteten Anwendung, dem Fronten, zur Verfügung stellt. Das Frontend greift über die Schnittstelle auf das Backend zu und nutzt dessen Funktionalitäten. Beide Systeme arbeiten vollständig unabhängig voneinander und können sich nur über die Schnittstelle beeinflussen.

In dieser Arbeit ist bei der Verwendung des Begriffes „Backend“ die Einheit gemeint, die die gesamte Logik des Erstellungsprozesses einer Stadt beinhaltet. Der Begriff „Frontend“ wird für die Grafikschnittstelle verwendet, die die Daten des Backends darstellt. Das Frontend stellt exemplarisch die Verwendung des Backends unter Verwendung einer Grafikengine dar, die die Behandlung der Grafikhardware abnimmt.

2.2.5 Rendering

Mit Rendering ist im weiteren Sinne das Zeichnen von grafischen Inhalten gemeint. Im engeren Sinne versteht man darunter die Rasterisierung von geometrischen Inhalten in der Computergrafik. Dieses Verfahren wird auch Scankonvertierung genannt. Diese Aufgabe wird von der Grafikhardware übernommen, nachdem diese über eine Programmierschnittstelle mit Daten befüllt wurde.

Zwei bekannte Programmierschnittstellen für die 2D- und 3D-Computergrafikentwicklung sind das plattformunabhängige OpenGL und plattformabhängige DirectX®. Mit diesen Schnittstellen kann die Grafikhardware direkt programmiert werden. Für wiederkehrende Aufgaben und andere hardwarenahe Funktionen wäre jedoch die aufwendige Entwicklung eines Frameworks nötig, um beispielweise Maus und Tastatureingaben zu erhalten. SDL (Simple DirectMedia Layer) und GLUT (OpenGL Utility Toolkit) sind zwei Bibliotheken, die auf OpenGL, DirectX oder ähnliche Programmierschnittstellen aufsetzen und diese Aufgabe umsetzen. GLUT ist jedoch aufgrund seines Alters nicht objektorientiert und daher nur bedingt brauchbar. SDL ist hingegen weniger auf die Erstellung dreidimensionaler Anwendungen ausgelegt und bietet hierzu nur Grundfunktionen an.

Um dennoch hardwarenah entwickeln zu können und einen einfachen Zugriff auf Eingabegeräte zu erhalten, empfiehlt sich die Verwendung einer Grafikengine.

Grafikengine

Eine Grafikengine ist eine Schnittstelle zwischen Anwendung und einer Programmierschnittstelle zur Grafikhardware sowie anderer Hardware. Es gibt eine Vielzahl solcher Engines, die sich meist in Programmiersprache, unterstützten Grafikhardwareprogrammierschnittstellen, Portabilität und Umfang unterscheiden. Aus diesem Grund sind einige solcher Engines auf ihre Einsatzmöglichkeiten geprüft worden.

OpenSG ist eine auf einem Szenegraphen basierende Engine. Ihre Stärken liegen neben der hierarischen Organisation in der Unterstützung von Clustersystemen und 3D-Anwendungen. Die Unterstützung von Clusteranwendungen führt jedoch eine sehr komplexe Programmierweise nach sich, die zuerst erlernt werden muss. Die nötigen Programmierkonstrukte erschließen sich meist nur durch eine Dokumentation, die jedoch nahezu vollständig fehlt. Aus diesem Grunde kommt OpenSG (*Open Scene Graph*) nicht in Frage.

CryEngine ist eine Spieleengine der Crytek GmbH, mit der z. B. der Kassenschlager „Crysis“ entwickelt worden ist. Die Engine zeichnet sich durch besonders hochwertige Grafikeffekte aus, insbesondere im Szenario von Inselwelten. Die CryEngine in Version 3 enthält eine Sandbox, die es ermöglicht, für mehrere Plattformen (PC, Playstation 3 und Xbox 360) zu entwickeln. Das Entwickeln in der Sandbox beschränkt sich jedoch ausschließlich auf das Skripten von Szenarien und das statische Designen einer Welt. Um von der vorgegebenen Physik abzuweichen, bietet die Engine eine in C++ geschriebene GameDLL an, die die Spiellogik beschreibt. Diese Logik kann durch Verändern der GameDLL angepasst werden. Der große Nachteil der Engine besteht jedoch darin, dass es nicht bzw. nur sehr umständlich möglich ist, Änderungen vorzunehmen, die nicht die Spiellogik betreffen. Das dynamische Generieren von Inhalten ist eine Aufgabe, die davon betroffen ist. Zudem schließen sich eine Inselwelt mit fester Größe und eine sich unendlich erweiternde Großstadt gegenseitig aus bzw. ließen sich diese Konzepte nur schwer miteinander vereinen.

OGRE 3D ist eine *O*pen*S*ource *G*rafik-*E*ngine für *3D*-Inhalte (Johnstone et al.). Sie vereint mehrere Programmierschnittstellen der Grafikhardware wie beispielsweise OpenGL und DirectX, der Audiohardware und anderer hardwarenaher Systeme. Sie ist für kein festes Szenario optimiert bzw. für jedes geeignet. Die Engine ist zudem an keine spezielle Plattform gebunden. Mit OGRE ist somit ein einfacher Zugriff auf Eingabehardware und Grafikhardware möglich. Die Entwicklung von Inhalten erfordert keine komplizierten Konstrukte wie in OpenSG und somit ist die Entwicklung mit dieser schnell zu erlernen. Ein weiterer Vorteil besteht darin, dass die Engine nicht so stark abgeschirmt und somit weniger eingeschränkt ist als die CryEngine. Zudem ist die Bibliothek durch den offenen Quellcode mittlerweile sehr stark optimiert und u.a. auch auf aktuelle Hardware zugeschnitten. OGRE besitzt eine gute Dokumentation, die zumeist sogar multilingual ist.

Somit ist die Entscheidung auf die OGRE 3D Engine gefallen, um mit dieser das Frontend umzusetzen.

2.2.6 Reguläre Ausdrücke

Für das Einlesen der Regelsätze zur späteren Straßengenerierung werden reguläre Ausdrücke verwendet. Mit ihnen ist es einfach möglich einen Ausdruck auf korrekte Syntax zu überprüfen und diesen in seine Bestandteile zu zerlegen. Diese Ausdrücke müssen jedoch zuerst konzipiert werden.

Symbol	Bedeutung
a	Das Zeichen „a“.
.	Ein beliebiges einzelnes Zeichen.
a+	Erfordert eine Zeichenkette mit einem oder beliebig vielen Vorkommen von „a“. (z.B. „aaa“)
a?	Wie „+“, jedoch 0 bis 1 Wiederholungen von „a“.
a*	Wie „+“, jedoch 0 bis beliebig viele Vorkommen von „a“.
[a-z0-9F]	Ein Zeichen aus dem Bereich „a“ bis „z“, „0“ bis „9“ oder das Zeichen „F“.
[^a-z]	Ein Zeichen, das nicht im Bereich „a“ bis „z“ liegt.
(...)	Gruppierung. Nach einer Erkennung kann auf sie per Index zugegriffen werden.
(?: ...)	Nicht zu indizierende Gruppierung.
a|b	Auswahl zwischen „a“ und „b“. Kann sowohl für Zeichen als auch Regelteile verwendet werden.
\.	„\“ ermöglicht das Erkennen von Symbolen als Zeichen. (Escape-Sequenz)

Tabelle 2.2.2: Symbole eines regulären Ausdrucks

Mit dem regulären Ausdruck „*([a-zA-Z]+) ([0-9.]+)*“ werden beispielsweise alle Wörter mit den Buchstaben „a“ bis „z“ in Groß- und Kleinschreibung gefolgt von einer positiven Dezimalzahl erkannt. Eine Erkennung einer Zahl mit mehreren Kommas wäre auch möglich, wird jedoch in diesem Ausdruck nicht behandelt. Eine Dezimalzahl kann mit dem etwas länglicheren Ausdruck „*[+-]?[0-9]+(\.[0-9]+)?*“ erkannt werden.

Die Verwendung der Ausdrücke zum Laden der Regelsätze wird im Detail in Abschnitt 4.2.4 auf Seite 45 beschrieben.

Die Anwendung der Wiederholungsoperatoren „+“ und „*“ auf Gruppierungen führen zu einem deutlich erhöhten Rechenaufwand bei der Erkennung und werden daher nicht zur Wiederholung von Gruppen verwendet.

2.3 Interdisziplinäre Grundlagen

Für die Erstellung glaubhafter Stadtbilder sind Grundkenntnisse von Städten und deren Architektur erforderlich gewesen. Für den in der Umsetzung gewählten Lösungsansatz müssen zudem Grundkenntnisse über das aus der Biologie stammende Lindenmayersystem vorhanden sein.

2.3.1 Der Stadtbegriff

Heineberg (2006) hat sich im Detail mit dem Stadtbegriff auseinandergesetzt und eine umfangreiche Gliederung von diesem erarbeitet. Es wird angemerkt, dass der Stadtbegriff abhängig von der Domäne eine andere Bedeutung besitzt.

Im Gebiet der *räumlichen Aufteilung* wird zwischen drei Siedlungsstrukturen unterschieden: der Desurbanisierung, der Reurbanisierung und einer nachhaltigen Stadtlandschaft. Von einer Desurbanisierung ist die Rede, wenn die Siedlung aus einem großen Zentrum besteht, um das sich die Vororte bilden. Eine Reurbanisierung ist gegeben, wenn mehrere kleinere Stadtzentren um ein größeres Zentrum herum existieren. Um die Stadtzentren bilden sich wiederum Vororte. Eine nachhaltige Stadtlandschaft bezeichnet zumeist eine gegebene Siedlungsstruktur. Sie ist gegenüber der anderen Strukturen weniger zentrenorientiert. Die nachhaltige Stadtlandschaft wird durch Abhängigkeiten zwischen den Regionen bestimmt.

Der Stadtbegriff in der *Umgangssprache* besitzt wiederum eine andere Bedeutung. So ist mit „in die Stadt fahren“ eigentlich das Stadtzentrum gemeint, das sich formal definieren lässt (siehe Abschnitt 2.3.2 auf Seite 21).

In der *statistisch-administrativen* Domäne wird der Stadtbegriff wiederum durch die Einwohnerzahl definiert.

Einwohnerzahl	Größenordnung
2.000 - 4.999	Landstadt
5.000 - 19.999	Kleinstadt
20.000 - 99.999	Mittelstadt
ab 100.000	Großstadt

Tabelle 2.3.1: Abhängigkeit zwischen Einwohnerzahl und Stadtgrößenordnung

Es existieren einige weitere Gliederungsmöglichkeiten, wie die Aufgliederung nach Stadtlage, wie Hanglage, Tallage und Meerlage. Eine Gliederung nach regionalen Besonderheiten, nach Funktion (politisch, kulturell, Wirtschaft, Verkehr), nach historischem Hintergrund oder nach Kulturraum ist zudem denkbar.

Städtesysteme

Für den Aufbau und die räumliche Gliederung von Städten gibt es zwei sich ähnelnde Theorien. Die „Theorie der zentralen Orte“ von Walter Christallers und die „Theorie

der Marktnetze“ von August Lösch (siehe Heineberg) beschreiben beide die Faktoren, die die Stadtdichte eines Gebietes bestimmen.

Christallers Theorie geht von zentralen Orten aus, die desto zentraler sind je mehr zentrale Güter sie besitzen. Als Kriterium für die Zentralität wird von Christaller beispielhaft die Anzahl der Telefonanschlüsse eines zentralen Ortes genommen[3]. Über eine Differenz zwischen durchschnittlichem und vorhandenem Telefonanschlussaufkommen bildet er zehn Zentralitätsstufen für alle Städte. Jedes zentrale Gut besitzt eine obere und untere Reichweitengrenze, von der die Stadtgröße abhängt. Je mehr Güter somit in der Reichweite einer Stadt liegen, desto größer wird eine Stadt. Durch diesen Einfluss bilden sich Städte in gleichem Abstand zueinander, die eine Abhängigkeit zueinander bilden. Diese Abhängigkeit kann hierarisch sein, so dass gewisse Güter nur in großen, andere Güter nur in kleineren Zentren existieren. Diese Theorie beachtet jedoch nur ökonomische Faktoren und keine geographischen Eigenheiten oder geschichtlich bedingte Einflüsse (Kinder, 2009).

Die „Theorie der Marktnetze“ erweitert das System auf güterspezifische Marktgebiete und kann auf geographisch beeinflusste Gebiete angewandt werden. Die Theorie erklärt die Größen der Zentren umliegender Städte mit der Entfernung der Städte zueinander. Eine steigende Entfernung zum Zentrum einer großen Stadt bedeutet eine Steigerung der Größe der umliegenden Zentren.

Im Umkehrschluss können diese Theorien bei der Erzeugung virtueller Städte über die Position bestehender Zentren neue Stadtzentren generieren.

2.3.2 Stadtgebiete

Für die Erstellung von virtuellen Städten sind Kenntnisse über die Aufteilung einer Stadt und ihrer speziellen Gebiete vonnöten. Städte lassen sich in Gebiete der Nutzungsform unterteilen. Neben allgemeinen Übergruppen lassen sich die Nutzungsformen in Untergruppen aufteilen.

Bauliche Nutzung Die wichtigste Form einer Stadt. Sie kann in drei Untergruppen aufgeschlüsselt werden: Wohnungsbau, Gewerbe und Sonderbau.

- **Wohngebiete** können sich in ihrem Aussehen verändern. Je nach Wohlstand sehen Armenviertel, Mittelschichtgebiete und Wohlstandsviertel unterschiedlich aus. Die Entstehung des letzteren Gebiets ist nur bei einer guten Infrastruktur denkbar.
- **Gewerbegebiete** bestehen aus Geschäften, Fabriken und Unternehmen, wie z.B. Supermärkte, Tankstellen, Baumärkte und Büros. Sie unterscheiden sich in ihrem Aussehen grundlegend von Wohngebieten. Die Areale sind bedingt durch nötige Abstellflächen um einiges größer.
- **Sonderbau** bezeichnet Gebäude mit außergewöhnlichen Abmessungen oder Nutzungsbereichen, wie Krankenhäusern oder Garagenanbauten. Die Einordnung

[3] Dies könnte in der heutigen Zeit durch die Anzahl der Hochgeschwindigkeits-DSL-Anschlüsse realisiert werden, die selbst 2010 in vielen kleinen Orten nicht existieren.

eines Gebäudes zum Sonderbau erfolgt in Deutschland z.B. durch die Bundesländer.

Gemeinbedarf Bezeichnet Gebiete mit für die Infrastruktur elementarer Bebauung. Einrichtungen der öffentlichen Verwaltung wie Bürgeramt und Arbeitsamt zählen zum Gemeinbedarf, aber auch Schulen, Polizeireviere, Büchereien, Kirchen und Friedhöfe. Die Zuordnung zu dieser Gruppe kann sich mit der Bedeutung der Sonderbauten vermischen. Um Gebiete bzw. Gebäude des Gemeinbedarfs bilden sich Wohngebiete.

Verkehr Ein nicht unerheblicher Flächenanteil wird für die Verkehrsmittel genutzt. Autobahnen, Bahnschienen, Fußgängerzonen oder Parkplätze zählen zu dieser Gruppe.

Versorgung Eine weitere Nutzungsgruppe besteht in der Stadtversorgung, die sich mit der Entsorgung und Endverwertung beispielsweise von Abwasser oder Abfall beschäftigt. Elektrizitätswerke oder Flächen zur ökologischen Energiegewinnung, wie Windkraftanlagen, nehmen ebenfalls Gebiete der Versorgungskategorie ein.

Sonstige Diese Gruppe beinhaltet u.a. „naturbezogene“ Gebiete, wie Grünflächen oder Seen innerhalb einer Stadt. Solche u.a. Erholungsgebiete wie Parks und Schwimmbäder fügen sich lückenlos in das Stadtbild ein und liegen meistens am Rand von Wohngebieten. Auch Rohstoffminen und landwirtschaftlich genutzte Felder sind zu dieser Gruppe zu zählen.

Stadtkern

Neben der Flächenverteilung innerhalb einer Stadt ist es von Interesse, den Begriff des Stadtkerns zu klären, um hierdurch glaubwürdige Städte erzeugen zu können.

Der Stadtkern einer Stadt, sozusagen die Innenstadt, kann mittels einer Formel von R.E Murphy und J.E. Vance (aus Heineberg) ermittelt werden. Als Grundlage dient der CBDHI[4] und CBDII[5]. Der CBDHI gibt ein Verhältnis der gewerblich genutzten Geschossfläche eine Baublocks zur gesamten Gebäudegrundfläche an.

$$\text{CBDHI} = \frac{\text{Gesamte Geschossflächen mit CBD-typischen Nutzungen}}{\text{Gesamte Gebäude-Grundflächen}}$$

Der CBDII beschreibt den prozentualen Anteil aller gewerblich genutzten Flächen eines Baublocks.

$$\text{CBDII} = \frac{\text{Gesamte Geschossflächen mit CBD-typischen Nutzungen}}{\text{Gesamte Geschossflächen}} \cdot 100$$

[4]CBDHI: Central Business District Höhenindex
[5]CDHII: Central Business District Intensitätsindex

Der eigentliche Stadtkern wird über das CBD-Abgrenzungskriterium beschrieben. Ist der $CBDHI > 1$ und der $CBDII \geq 50\%$ zählt der Baublock zum Stadtkern. Auf diese Weise lässt sich eine Grenze zwischen der Innenstadt und dem Rest der Stadt berechnen. Es ist möglich, dass hierdurch mehrere Stadtkerne entstehen.

2.3.3 Gebäudetypen

Abhängig vom Stadtgebiet existieren verschiedene Gebäudetypen. So gibt es an den Stadträndern in den Vororten meist Ein- und Mehrfamilienhäuser. Im Bereich des Stadtzentrums sind öfter Geschosswohnungsbauten wie Hochhäuser und Vielwohnungshäuser anzutreffen. In speziellen Gebieten können Sonderformen entstehen, wie Terrassenhäuser am Hang.

Dachformen

Ein authentisches Merkmal eines Hauses ist das Dach. Dächer gibt es in vielen Farben und Variationen, wovon die wichtigsten vorgestellt werden.

Satteldach Das Satteldach besteht aus einem Dachfirst entlang der längeren Gebäudeseite und zwei zu diesem parallele Gebäudekanten, die ein Dreieck aufspannen. Es entstehen zwei geneigte Dachseiten. (2.3.1a)

Walmdach Das Walmdach besitzt zusätzlich zum Satteldach nicht nur zwei Schrägen, die sich im Dachfirst treffen, sondern auch zwei an den kurzen Seiten des Hauses. Diese Walme kommen durch einen verkürzten First zustande (2.3.1b). Ein Zeltdach entsteht, wenn der Dachfirst wegfällt und sich alle Dachseiten in einer Spitze treffen.

Krüppelwalmdach Ein Krüppelwalmdach entspricht einem Satteldach mit nach oben verschobenem Walm. Der Walm schließt somit nicht mit dem unteren Dachende, der Traufe, ab. (2.3.1c)

Mansarddach Das Mansarddach ähnelt dem Satteldach. Die Dachschrägen sind im unteren Bereich zusätzlich abgeknickt. Die unteren Dachbereiche sind steiler als die oberen. (2.3.1d)

Pultdächer besitzen nur eine Dachschräge. Die verläuft vom Dachfirst an der längeren Wand zur Dachtraufe an der kürzeren Wand. Sie eignet sich neben Flachdächern auch für Hochhäuser. (2.3.1e)

Sheddächer werden durch das Hintereinanderreihen von Pult- oder Satteldächern erzeugt. Sie finden meist Einsatz bei großflächigen Gebäuden, wie Fabriken oder Lagerhallen. (2.3.1f)

Flachdach Ein weiterer Dachtyp ist das Flachdach. Es besitzt keine oder kaum eine Steigung und findet oft bei Hochhäusern Verwendung.

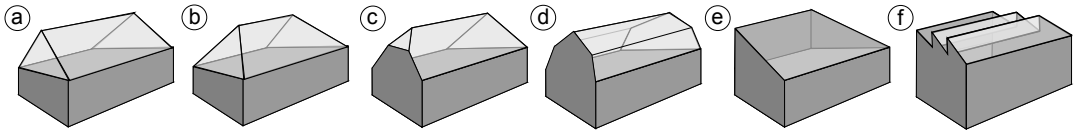


Abb. 2.3.1: Verschiedene Dachformen von Gebäuden

2.3.4 Lindenmayer-Systeme (L-Systeme)

Lindenmayersysteme (Prusinkiewicz/Lindenmayer (1990)) sind um 1990 von Astrid Lindenmayer und Przemyslaw Prusinkiewicz entwickelt worden und dienen zur Beschreibung von organischen Strukturen wie beispielsweise Pflanzen. Mit diesen ist es möglich, mit Hilfe einer einfachen Sprache die Definition einer Struktur anzugeben.

L-Systeme haben den Vorteil, dass diese ein abwechslungsreiches und organisches Bild schaffen können. Sie werden nicht nur bei der Beschreibung von Pflanzenwachstum eingesetzt, sondern können auch zur Erstellung eines organisches Stadtbildes verwendet werden.

In dieser Arbeit werden L-Systeme eingesetzt, um ein Straßennetz, ähnlich wie einen Baum, wachsen zu lassen. L-Systeme lassen sich am besten mit der Schildkröten-Metapher erklären. Dies bedeutet, dass ein L-System einen Pfad beschreibt, der von einer Schildkröte zurückgelegt wurde. Diese kann nur vorwärts gehen und ihre Ausrichtung ändern und sich entlang dieser wiederum vorwärts bewegen.

Einfache L-Systeme

Ein einfaches L-System besteht aus Befehlen, Variablen, Regeln und einem Startzustand (ω). Mit dem nachstehenden L-System kann eine einfache Treppe gezeichnet werden.

$$\begin{aligned} \omega &\rightarrow A \\ A &\rightarrow F + F - A \end{aligned}$$

Die Befehle eines einfachen L-Systems sind folgendermaßen zu interpretieren:

Symbol	Bedeutung
F	Zeichnet eine Linie der Länge 1 nach vorne
f	Bewegung um 1 nach vorne ohne zu zeichnen
+	Dreht die Orientierung um 90° nach links
-	Dreht die Orientierung um 90° nach rechts
[	Setze eine Markierung
]	Kehre zur entsprechenden [-Markierung zurück

Tabelle 2.3.2: Symbole eines Lindenmayersystems

Die Auswertung eines L-Systems ist ein iteratives Verfahren. In jedem Auswertungs-

schritt, werden alle Vorkommen von Platzhaltern durch deren Entsprechung ersetzt (siehe Abbildung 2.3.2). Im Folgenden stehen die ersten fünf Folgeiterationen:

$$
\begin{aligned}
I_{(n=0)} &: A \\
I_{(n=1)} &: F + F - A \\
I_{(n=2)} &: F + F - F + F - A \\
I_{(n=3)} &: F + F - F + F - F + F - A \\
I_{(n=4)} &: F + F - F + F - F + F - F + F - A \\
I_{(n=5)} &: F + F - F + F - F + F - F + F - F + F - A \\
&\ldots
\end{aligned}
$$

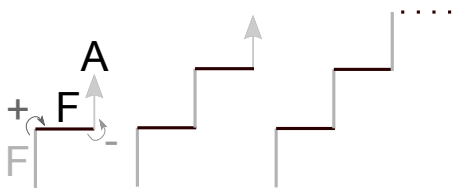


Abb. 2.3.2: Beispiel eines einfachen L-Systems

Eine weitere einfache Erweiterung ist die stochastische Regelwahl. Bei dieser gibt es mehr als eine mögliche Regelersetzung. Die Regel wird dann zufällig mit der entsprechenden Wahrscheinlichkeit ausgewählt. Im untenstehenden Beispiel wird eine Treppe erzeugt die zu 30% doppelt so lange Stufen besitzt.

$$
\begin{aligned}
\omega &\rightarrow A \\
A &\rightarrow_{0.7} F + F - A \\
A &\rightarrow_{0.3} F + FF - A
\end{aligned}
\tag{2.3.1}
$$

Durch stochastische Regeln können verschiedene Ergebnisse erzeugt werden, was für Alternativen wünschenswert ist. Abbildung 2.3.3 auf der nächsten Seite zeigt drei mögliche Ergebnisse eines Baumes, der mit einem stochastischen Regelwerk erzeugt worden ist.

Erweiterte L-Systeme

Eine Erweiterung des L-Systems besteht darin, dieses um Parameter zu erweitern, wie es z. B. in Chen et al. (2002) eingesetzt wird, um realistische Federn zu erzeugen. Hierbei werden statt Variablen Funktionen benutzt, die beliebig viele Parameter besitzen können. Auch der Befehlssatz wird durch Funktionen ersetzt.

Mit dem folgenden System lässt sich das gleiche Ergebnis erzielen, wie mit der nicht parametrischen Variante (Formel 2.3.1).

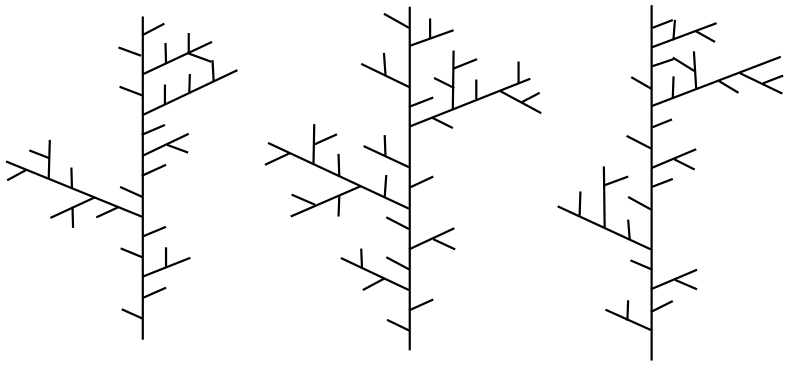


Abb. 2.3.3: Baum mit zufälliger Regelwahl

Funktion	Bedeutung
F(len)	Zeichnet eine Linie der Länge *len* nach vorne
f(len)	Bewegung um *len* nach vorne ohne zu zeichnen
$+(\alpha)$	Dreht die Orientierung um $\alpha°$ nach links. Negative Werte drehen nach rechts.
[,]	Einstellige Funktion. Setzt eine Markierung bzw. kehrt zur entsprechenden zurück.

Tabelle 2.3.3: Parametrische Symboldefinition des L-Systems

$$\begin{aligned} \omega &\rightarrow R \\ A(X) &\rightarrow F(X) + (90)F(X) + (-90) + R \\ R &\rightarrow_{0.3} A(1) \\ R &\rightarrow_{0.7} A(2) \end{aligned}$$

Die parametrische Variante hat den Vorteil, dass durch Veränderung der Regel A diese in den Regeln von R sofort Anwendung finden.

Eine weitere Ergänzung besteht darin den einzelnen Regeln des Systems Bedingungen zuzuordnen. Diese können arithmetisch oder kontextsensitiv sein. Im letzteren Fall ist noch zu unterscheiden zwischen einer Selbstbeeinflussung und einer Beeinflussung durch die Umgebung (Open-L-System).

Das nachstehende System verwendet einfache arithmetische Regeln, um eine Blume zu beschreiben (Auszug aus Prusinkiewicz/Lindenmayer (1990)), sowie eine Erweiterung für die dritte Dimension (siehe Abschnitt 2.3.4 auf Seite 27).

$$
\begin{aligned}
o &\rightarrow +(-90)I(9)a(13)\\
a(t)_{t>0} &\rightarrow [\&(70)L]/(137.5)I(10)a(t-1)\\
a(t)_{t=0} &\rightarrow [\&(70)L]/(137.5)I(10)A\\
A &\rightarrow [\&(18)u(4)FFI(10)I(5)KKKK]/(137.5)I(8)A\\
I(t)_{t>0} &\rightarrow FI(t-1)\\
I(t)_{t=0} &\rightarrow F\\
u(t)_{t>0} &\rightarrow \&(9)u(t-1)\\
u(t)_{t=0} &\rightarrow \&(9)\\
L &\rightarrow [+(-18)FI(7)+(18)FI(7)+(18)FI(7)]\\
&\quad\ [+(18)FI(7)+(-18)FI(7)+(-18)FI(7)]\\
K &\rightarrow [\&(18)+(18)FI(2)+(-18)+(-18)FI(2)]\\
&\quad\ [\&(18)+(-18)FI(2)+(18)+(18)FI(2)]/(90)
\end{aligned}
$$

Bei kontextsensitiven Regelwerken ist die Ausführung der Regel vom Umfeld abhängig, in dem eine Einfügung durchgeführt werden soll. Im folgenden Regelsatz wird ein solches System verwendet, um eine Signalweitergabe zu simulieren. Das Zeichen „b“ wird hierbei von links nach rechts durchgereicht.

$$
\begin{array}{ccccccc}
 & & \omega & & & \rightarrow & baaa\\
b & < & a & & & \rightarrow & b\\
 & & b & > & b & \rightarrow & a
\end{array}
$$

Alle Vorkommen von „a“, die einem „b“ folgen werden zunächst durch „b“ ersetzt und anschließend werden alle Vorkommen von „b“ gefolgt von „b“ durch „a“ ersetzt bzw. der Ausdruck „ba“ wird durch „ab“ ersetzt.

$$
\begin{aligned}
I_{(n=0)} &: baaa\\
I_{(n=1)} &: abaa\\
I_{(n=2)} &: aaba\\
I_{(n=3)} &: aaab
\end{aligned}
$$

Open-L-Systeme verwenden zur Simulation von Kontext Umgebungsinformationen, die z.B. eine maximale Ausbreitung oder einen Bereich angeben, in dem das System wachsen darf (Wurzelwachstum in einem Blumentopf). Eine weitere Eigenschaft eines solchen Systems ist die Möglichkeit auf den eigenen Zustand zugreifen zu können, wodurch beispielsweise Kollisionen von Pfaden verhindert werden können.

Funktion im 3D

Um ein L-System in 3D beschreiben zu können, wird dieses um zwei weitere Drehungsoperatoren erweitert.

Die Vektoren H, L und U sind Einheitsvektoren und sind alle orthogonal zueinander, was mit der Gleichung $\overrightarrow{H} \times \overrightarrow{L} = \overrightarrow{U}$ ausgedrückt werden kann. Rotationen werden durch eine 3×3-Matrix ausgedrückt, mit der Form $M' = M \cdot R \Rightarrow \left[\overrightarrow{H'}\,\overrightarrow{L'}\,\overrightarrow{U'}\right] = \left[\overrightarrow{H}\,\overrightarrow{L}\,\overrightarrow{U}\right] \cdot R$. Wird beim Zeichnen im Ursprung bei $(0, 0, 0)$ begonnen, so wird die Folgeposition mit $x' = x + a \cdot H_x$, $y' = y + a \cdot H_y$ und $z' = z + a \cdot H_z$ bestimmt. Es ist zu beachten, dass jede Rotation in einem lokalen Koordinatensystem ausgeführt wird und somit in Abhängigkeit der vorherigen Orientierung der Schildkröte steht. Die zusätzlichen Befehle des 3D-L-Systems sind in Tabelle 2.3.4 im Überblick zu sehen.

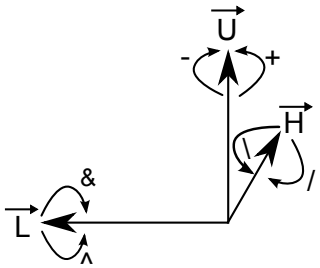


Abb. 2.3.4: Koordinatensystem eines 3D-L-Systems

Funktion	Bedeutung
+ (α)	Drehung um den lokalen U-Vektor um $\alpha°$. (gieren/yaw)
& (α)	Drehung um den lokalen L-Vektor um $\alpha°$. (nicken/pitch)
/ (α)	Drehung um den lokalen H-Vektor um $\alpha°$. (rollen/roll)
F(a)	Bewegung entlang des H-Vektors mit Länge a. $P_{neu} = P_{alt} + a \cdot H$

Table 2.3.4: Erweiterter Symbolsatz eines 3D-L-Systems

Ein Problem bei der Auswertung des Systems besteht darin, dass die Zeichenoperationen entlang eines lokalen Koordinatensystems ausgeführt werden. Um jedoch Kollisionstests durchführen zu können, müssen die Koordinaten zusätzlich in Weltkoordinaten mitgeführt werden und aus den jeweils vorfolgenden Transformationsmatrizen errechnet werden.

Optimierung

Wie festzustellen ist, gibt es für die Auswertung eines Lindenmayersystems eine feste Reihenfolge. Diese ist iterativ und von links nach rechts. Dies führt dazu, dass die Ausführung rein linear stattfindet und nicht optimiert werden kann. Lipp/Wonka/Wimmer (2009) hat sich mit der parallelen Ausführung von L-Systemen beschäftigt und ein Verfahren entwickelt, mit dem sich die Ausführung des Systems in mehrere Teile zerlegen lässt. Die Zerlegungen können gleichzeitig ausgewertet werden und in einem späteren Schritt wieder zusammengeführt werden.

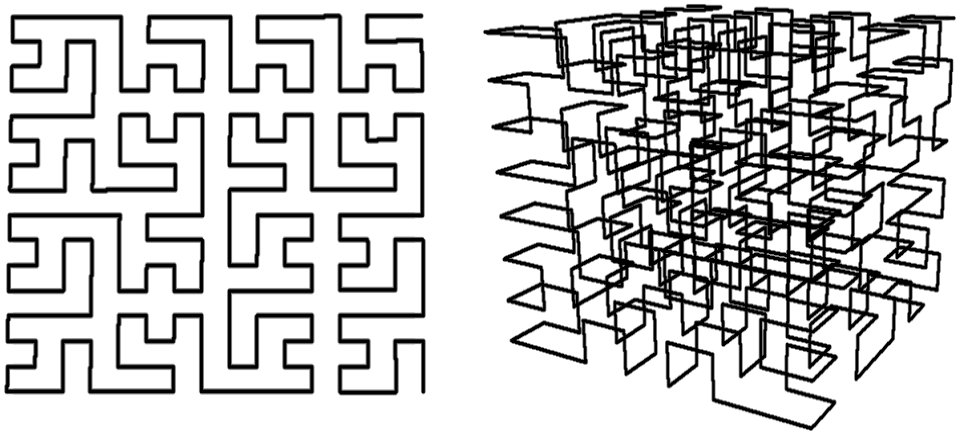


Abb. 2.3.5: Hilbertmuster und Hilbertwürfel mit 3D-Lindenmayersystem erzeugt

Bei dieser Parallelisierung wird sichergestellt, dass das gleiche Ergebnis erzielt wird, wie bei einer linearen Ausführung. Da L-Systeme in dieser Arbeit verwendet werden um ein organisches und zufälliges Bild zu erreichen, ist ein konsistentes Ergebnis nicht zwingend nötig. Aus diesem Grund und der Anforderung, dass nur ein lokales Teilergebnis benötigt wird, kann die Auswertung des L-Systems auf eine spezielle Art und Weise durchgeführt werden (siehe 4.2.5).

Kapitel 3

3 State of the Art

Zur Schaffung virtueller Städte gibt es eine Reihe von Ansatzpunkten. Einer besteht in der virtuellen Rekonstruktion bestehender Städte bzw. Gebiete. Ein anderer Ansatz, der nicht auf die vollständige Rekonstruktion hinausläuft, besteht darin, eine Stadt aus äußerlichen Gegebenheiten durch Wachstum zu rekonstruieren. Werden viele Umgebungsparameter behandelt, kann dadurch ein nah am Original liegendes Stadtbild geschaffen werden. Sollen vollständig neue Städte aus wenigen bis gar keinen Eingabedaten geschaffen werden, ist ein regelbasierter Ansatz nötig, der die Beschaffenheit der zu erstellenden Stadt beschreibt.

Bei allen Ansätzen ist eine Beschreibungssprache zu wählen, die computerverständlich ist und somit automatisiert ausgewertet werden kann. Alexander/Ishikawa/Silverstein (1977) haben bereits das Aussehen von Städten im Detail in vielen natürlichsprachlichen Regeln beschrieben, diese jedoch nicht schematisch ausgedrückt. Eine Übertragung dieser Beschreibungen auf Formeln ist somit im Detail nicht möglich oder würde ein sehr umfangreiches Auswertungssystem benötigen.

3.1 Rekonstruktion

Der Ansatz der Rekonstruktion kann entweder manuell durch Modellierung existierender Gebäude und Objekte oder automatisiert geschehen. GoogleMaps, GoogleEarth und BingMaps sind Anwendungen, in denen auf dreidimensionale Gebäude zurückgegriffen wird, um beispielsweise eine Stadt nicht nur virtuell aus Fußgängersicht entlang eines vorgegebenen Pfades, sondern auch frei betrachten zu können. Bei diesen Anwendungen wird hauptsächlich auf Rekonstruktion durch Modellierung zurückgegriffen, jedoch werden zunehmend automatisierte Verfahren verwendet.

Weil die manuelle Modellierung sehr aufwändig ist, wird mittlerweile auf andere Verfahren zurückgegriffen. Luftaufnahmen, Höhendaten und Straßenaufnahmen dienen dabei als Grundlage zur Generierung dreidimensionaler Objekte.

In Henricsson/Streilein/Gruen (1996) wird ein Verfahren zur automatischen schematischen Rekonstruktion von Gebäuden aus Höhendaten vorgestellt. Der Schwerpunkt dieser Arbeit liegt auf der Rekonstruktion der Dachform. Es wird versucht die Dachform

über verschiedene Höhenmessungen aus einem Katalog modellierter Dächer zu ermitteln. Dies bringt einige Probleme mit sich, wenn die Dachform unbekannt ist oder bedingt durch falsche oder fehlende Höhendaten nicht ermittelt werden kann. Dem Problem wird durch die Trennung von Gebäudeerkennung und Rekonstruktion entgegengewirkt. Dies geschieht indem mehrerer überlappende unterschiedlich ausgeleuchtete Farbbilder verwendet werden und eine möglichst frühe Überführung in den 3D-Raum erfolgt. Aus diesen Bilddaten werden Features extrahiert und mittels einer Stereorekonstruktion in die dritte Dimension übertragen. Die vorgestellte Software TOBAGO benötigt jedoch das Eingreifen eines Anwenders zur Bestätigung oder Korrektur erkannter Dach- und Gebäudeformen, die über die Dächer erschlossen wird. Der Nachteil dieses Verfahrens ist, dass von der Bodenansicht aus keine fotorealistischen Ergebnisse erzeugt werden können.

In Xiao et al. (2009) wird diesem Problem durch einen bildbasierten Ansatz zur Rekonstruktion von Straßenblöcken entgegengewirkt. Anders als in Henricsson/Streilein/Gruen werden keine Höhendaten verwendet, sondern auf Fotografien von Gebäudefassaden bzw. die gesamten Fassaden eines Straßenzugs zurückgegriffen. Als Unterstützung wird auf Tiefendaten von Laserscans zurückgegriffen. Der Rekonstruktionsprozess erstellt zuerst aus Bild- und Tiefendaten eine in Regionen segmentierte Ansicht, die aus Straße, Gebäude und Himmel besteht. Im nächsten Schritt wird der Gebäudeteil des Straßenzugs entlang von senkrecht verlaufenden dominanten Kanten in Gebäude unterteilt. Die Gebäude an sich werden in einzelne Merkmale wie Fenster und Türen zerlegt und somit analysiert. Nach abgeschlossener Analyse wird eine dreidimensionale Gebäudefront generiert, die nach hinten erweitert wird. Dieses Verfahren eignet sich hervorragend um interaktiv passierbare 3D-Straßenzüge zu rekonstruieren, jedoch nicht für Luftansichten. Zudem können durch dieses Verfahren leicht Texturen für Teile von Gebäuden erstellt werden, die in anderen Anwendungsgebieten eingesetzt werden können.

Eine Kombination der beiden Verfahren liefert eine relativ zuverlässige und nahezu automatische Rekonstruktion von Gebäuden. Das Verfahren lässt sich zudem auf andere Szenarien übertragen, in denen 3D-Modelle aus 2D-Daten erzeugt werden müssen.

3.2 Aufbau durch Entstehung

Möchte man eine Stadt in einem bestehenden Gebiet wachsen lassen, so ist ein umfangreiches Verständnis von Stadtwachstum erforderlich. In Lechner/Watson/Wilensky; Watson wird der Ansatz verfolgt über die Simulation von Stadtzonen und elementaren Einheiten, die für eine funktionierende Infrastruktur nötig sind, Städte zu erzeugen. Mögliche Zonen bzw. Einheiten sind in Abschnitt 2.3.2 auf Seite 20 behandelt worden.

Neben dieser Unterteilung ist zu beachten, dass äußere Umstände wie Krieg, Naturkatastrophen, Erdbeben oder Tsunamis das Stadtbild stark beeinflussen, da neue Strukturen auf alten entstehen.

Auf diese Weise erschaffene Städte erfordern Transportsysteme. Bewohner müssen sich zwischen Zonen bewegen können, was durch Systeme mit unterschiedlichem Transportvolumen geschehen kann (Überlandstraßen, Autobahnen, Busse, Straßenbahnen, U-Bahn).

Güter müssen für die Produktion anderer Güter möglichst störungsfrei transportiert werden, wofür sich ein Schienennetz eignet (Güterzüge, etc.).

Durch die Verbindung von Transportsystemen mit Stadtzonen ergeben sich zwei Konstruktionsmöglichkeiten. Entweder es werden Gebäude und Zonen angelegt und durch diese ein Transportnetz gelegt, oder es wird zuerst ein Transportnetz angelegt und anschließend die Zonen um dieses herum geschaffen.

3.3 Konstruktion durch Regeln

Der regelbasierte Konstruktionsansatz ist sehr ähnlich zum Ansatz aus „Aufbau durch Entstehung“, nur dass das Vorgehen teilweise invers ist. Elementare Strukturen, die sich aus dem Stadtwachstum ergeben, werden hier als Grundlage verwendet, um aus diesen den Stadtwachstum nachzubilden. Mit dieser elementaren Struktur ist das Transportnetz einer Stadt gemeint. Die folgenden Ansätze spezialisieren sich daher hauptsächlich auf die Konstruktion eines Transportnetzes, wodurch die geometrische Struktur nachgebildet wird.

3.3.1 Geometrische Verfahren

Mehrere Verfahren zur Erzeugung verschiedener geometrischer Muster werden ausführlich von Pascal Müller (Müller, 2001; Pascal Müller, 2001; Müller et al., 2006; Müller et al., 2007; Müller, 1999; Müller, 2006 und Müller Pascal Mueller's Wiki; Müller Procedural Inc. - 3D Modeling Software for Urban Environments) in seinen Arbeiten behandelt.

Zu diesen Mustern zählt u.a. das Rastermuster, das in geplanten Städten wie New York vorzufinden ist. In einer Semesterarbeit von Pascal Müller (1999) wird ein L-System (siehe Abschnitt 2.3.4 auf Seite 23) verwendet, um eine Manhattan ähnelnde Schachbrettmusterform zu generieren. In der Arbeit wird im Detail auf die Erweiterungsmöglichkeiten eines Lindenmayersystems eingegangen. Das Open-L-System wird außerdem verwendet um mit der Umgebung zu interagieren. Diese Interaktion betrifft zum Einen das Terrain und zum Anderen die bereits erstellten Straßen. Die erzeugten Straßennetze der Arbeit liefern ein realistisches Ergebnisbild, jedoch wird angemerkt, dass eine lokale Modifikation der Ergebnisse nicht möglich sei. Eine lokale Modifikation führt bei L-Systemen dazu, dass nachfolgende Segmente mittransformiert werden würden und somit das Ergebnis in großen Teilen verändert werden würde. Eine weitere Schwierigkeit besteht in der Anpassung für andere Stadttypen, da ein völlig neues Regelwerk erschaffen werden müsste und das Regelwerk nicht ohne Weiteres auf das Ergebnisbild schließen lässt.

In Pascal Müllers Abschlussarbeit (2001) wird das bestehende System zu einer vollständigen Pipeline erweitert. Als Eingabedaten dienen, neben dem L-System, geographische und sozialstatistische Karten. Auf diesen Daten basierend werden die Schritte der Straßennetzerzeugung, Parzellenunterteilung, Gebäudeerstellung und Geometrieüberführung durchgeführt. Die Straßennetzerzeugung ist um die Unterstützung radialer Straßenmuster erweitert worden, wie es beispielsweise in Paris vorzufinden ist. Die

Parzellenunterteilung wird mit einem einfachen Subdivisionsalgorithmus realisiert (siehe 3.3.2). Für die Gebäudeerstellung wird erstmals die Shape-Grammar eingeführt. Sie vereint drei Techniken. Zu denen zählt erstens die Formengrammatik, die bestehende Geometrieobjekte mit verschiedenen Transformationen kombiniert. Die zweite Technik besteht aus Definitionen von Formelementen, die in beliebiger Art kombiniert werden können. Die letzte Technik, die in der Shape Grammar umgesetzt wird, ist die Substitution von Blöcken eines Blockgebäudes durch andere Formen. Diese Grammatik basiert wiederum auf L-Systemen, wovon es für jede Gebäudeart ein anderes Regelwerk gibt. Die Geometrieausgabe erfolgt über die Konvertierung der Ergebnisse in ein Format, das von einer Renderingsoftware gelesen werden kann (OBJ-Datei in Maya).

Im Anschluss (2001) ist das System um die prozedurale Texturierung erweitert worden. Zusätzlich ist die prozedurale Gebäudeerzeugung so erweitert worden, dass die verschiedenen Ersetzungstiefen des L-Systems für verschiedene LODs[1] verwendet werden können, die je nach Entfernung zum Betrachter mit Details versehen werden. Die Texturierung erfolgt mittels aus Basistexturen prozedural erstellten Texturen. Die Grenze des Systems besteht darin, dass Basistexturen manuell in ihre Teile, wie Fenster und Türen, zerlegt werden müssen.

In einer Präsentation aus 2006 wird neben dem aktuellen Fortschritt erstmals die prozedurale Modellierung von Straßen und Kreuzungen aus Straßenzentrumslinien (ähnlich zu 4.4.2) erwähnt.

In Müller et al. (2007) wird eine Technik vorgestellt, die Abhilfe für die fehlende lokale Modifizierbarkeit von L-Systemen schafft. Für die Straßennetzerstellung wird nun ein Tensorfeld[2] verwendet, das die bestehende Technik mit L-Systemen ablöst. Das Tensorfeld kann durch Vorgabe von Richtungslinien, Mustern oder verrauschten Bereichen, die z.B. mittels Perlin Noise erstellt werden können, beeinflusst werden. Es wird in drei Schritten ein Straßennetz erzeugt. Zuerst wird ein Tensorfeld vom Benutzer erzeugt, dann daraus ein Straßengraph erzeugt und dieser abschließend in Geometrie überführt. Der Straßengraph wird mittels gleichmäßig verteilten Strömungslinien erzeugt. Haupt- und Nebenstraßen werden durch Haupt- und untergeordnete Strömungslinien erzeugt. Der Schwerpunkt liegt auf der Erstellung eines verbundenen Gesamtnetzes, das vom Benutzer beeinflusst werden kann.

Bei den Arbeiten von Pascal Müller et. al. ist festzustellen, dass eine feste Pipeline gegeben ist, die aus Eingabe, Verarbeitung und Ausgabe besteht. Ein Eingriff während der Verarbeitung scheint nicht möglich zu sein. Zudem scheint das Framework durch den hohen Detailgrad der Ergebnisse nicht echtzeitfähig zu sein. Auf die Echtzeitfähigkeit und lokal modifizierbare L-Systeme wird später eingegangen.

Sun et al. (2002) haben ein Verfahren zur Erstellung von Städten entwickelt, das nicht mit einem Regelsatz, wie bei L-Systemen angegeben wird, sondern einen Highway erzeugt und um diesen herum, basierend auf der Bevölkerungsdichte, Straßenbausteine einfügt.

Eine zusätzlich in der Literatur erwähnte Form (Sun et al. (2002); O'Rourke (1998)) ist das Wabenmuster. Es entsteht mittels Voronoidiagramm (siehe 2.2.2). Als Inselzentren

[1] Level of Detail

[2] Ein Spannungsfeld, das die Richtungen der Straßen beeinflusst.

werden zentrale Gebäude der Stadtstruktur gewählt. Das Ergebnis bei diesem Verfahren wirkt jedoch nicht sehr glaubwürdig, da teilweise keine Straßenzugänge zu Gebäuden erzeugt werden.

Ein in u.a. Deutschland vertretener Stadtstil, wird in Arbeiten von Pascal Müller häufig als westlicher Stil bezeichnet. Jacobs (1995) hat sich mit den Kriterien westlichen Straßenbaus auseinandergesetzt. Er bemerkt, dass Hauptverkehrsstraßen meist entlang von Höhenlinien (Isohypsen) verlaufen. Isohypsen sind auf Landkarten benachbarte Punkte mit nahezu gleicher Höhe, wie beispielsweise die 100-Meter-Höhenlinie eines Hügels.

Eine echtzeitfähig inspirierte Arbeit stammt von Lechner/Watson/Wilensky (2003). In dieser Arbeit werden westliche Städte mittels zwei Agenten für Erweiterung und Verbindung von Straßen verwendet. Es werden jedoch nur tertiäre Straßen[3] auf vorgegebenem Terrain geschaffen. Das gesamte Land wird mit dem Erweiterungs-Agent so mit Straßen zu verknüpfen versucht, dass die Steigung einer Straße möglichst gering bleibt, die Dichte der Straßen ausgewogen ist und nicht zu viele Straßenschnitte entstehen. Der Verbindungs-Agent erweitert Straßenteile, die einen zu langen minimalen Zyklus bilden, so dass dieser Zyklus kleiner wird.

In einer Arbeit von Bruneton (2005) wird eine gesamte Stadt in einem hohlen zylindrischen Raumschiff mit dem Namen „Rama“ nachmodelliert. Für die Modellierung werden u.a. Algorithmen eingesetzt, die für den Stadtbau interessante Aspekte beinhalten. So werden beispielsweise Gebäude prozedural aus Prismen erstellt, Parzellen gesucht und Straßen korrekt texturiert. Leider gehen aus dieser Arbeit nur wenige Details hervor.

3.3.2 Parzellen

Der wichtigste Teil bei der Generierung einer prozeduralen Stadt besteht im Finden der Parzellen. Parzellen sind im genaueren Sinne die Flächen, die vollständig von einem Straßenzug eingeschlossen sind und keine weiteren Straßen bis auf Sackgassen beinhalten. Werden die Straßen als Kanten eines Graphen betrachtet, sind Parzellen jene Zyklen, die geometrisch den kürzesten Weg vom Startknoten zu diesem zurück besitzen. Sind diese Zyklen alle gefunden (Details siehe 2.2.3 auf Seite 16), können die Parzellen in Grundstücke unterteilt werden.

Hierfür gibt es mehrere Möglichkeiten:

- Flächenfüllend
- Subdivision
- Verfahren nach Schrader (2007)

Die einfachste Lösung zur Bebauung der Fläche besteht darin, ein vollständig füllendes Gebäude einzuzeichnen, was jedoch kein sonderlich zufriedenstellendes Ergebnis liefert.

Eine deutlich bessere Lösung ist die iterative Unterteilung der Fläche (siehe Abbildung 3.3.1 auf der nächsten Seite). Hierbei wird die Fläche solange entlang der längeren

[3] Tertiäre Straßen sind Straßen der untersten Ebene. Sie verlaufen in Wohnungssiedlungen und Industrieparks und sind schmäler als Zubringerstraßen.

Flächenhalbierenden unterteilt, bis die gewünschte Seitenlänge erreicht ist (Bruneton (2005); Rudzicz (2008)). Als Grundflächen werden jene verwendet die eine Kante mit dem Parzellenumriss teilen. Alle anderen Grundrisse werden verworfen. Der Nachteil dieses Verfahrens besteht in der Notwendigkeit einer gesonderten Behandlung konkaver Flächen und Flächen mit ungerader Seitenzahl.

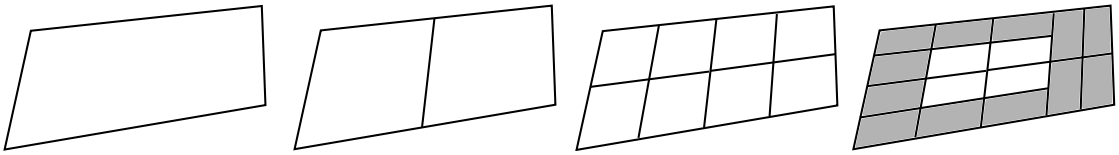


Abb. 3.3.1: Klassische Subdivision einer Fläche

Ein weiteres Verfahren wurde von Schrader entwickelt. Mit diesem Verfahren lassen sich sehr einfach glaubhafte Flächeneinteilungen erzielen. Das Verfahren arbeitet mit beliebig entarteten Flächenformen und wird in einer vereinfachten Variante angewandt (siehe Abbildung 3.3.2 auf der nächsten Seite).

1. Zuerst wird der Umfang (U) einer Parzelle ermittelt und aus diesem eine Anzahl der Gassen ($|G| = \lfloor U/D \rfloor$) mit einer Dichte ($D$) zwischen den Gebäuden ermittelt.

2. Es werden nun der Anzahl $|G|$ entsprechend viele Markierungen auf der jeweils längsten Kante der Parzelle verteilt.

3. Jede Kante wird nun in n gleichgroße Segmente unterteilt, wobei n durch die Anzahl der zugeteilten Markierungen bestimmt wird.

4. Von jeder Unterteilungsstelle wird nun ein orthogonaler Strahl durch das Flächeninnere geschickt und alle Schnittpunkte zwischen diesen Strahlen gespeichert.

5. Für alle Strahlen wird nun der vorderste Schnittpunkt bestimmt und jeweils eine Verbindung (k) zwischen den vorderen Schnittpunkten zweier Nachbarstrahlen erstellt.

6. Eine Baufläche entsteht jeweils über die Außenkante zwischen zwei Markierungen, den Strecken zwischen diesen Markierungen und dem entsprechenden vorderen Schnittpunkt sowie der Verbindung k zwischen den Schnittpunkten.

Bei diesem vereinfachten Vorgehen gibt es jedoch einige mathematische und optische Sonderfälle bei der Verwendung der benachbarten Strahlen zu beachten:

- Mathematisch:

  M1 Der vorderste Schnittpunkt liegt zu nah an einer Außenkante.

  M2 Zwei Strahlen sind koinzident.

  M3 Ein Strahl wird nicht im definierten Bereich geschnitten.

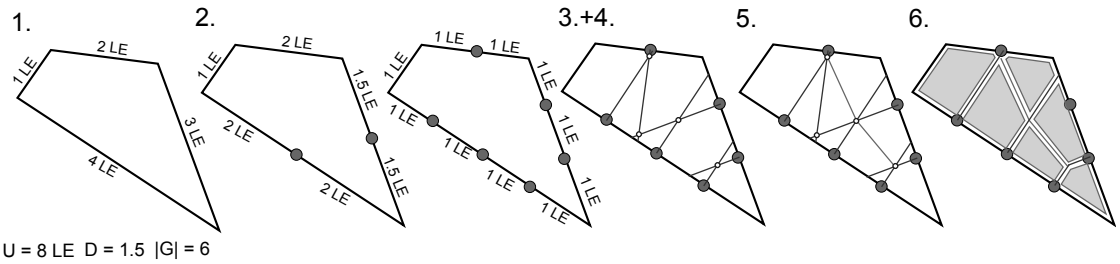


Abb. 3.3.2: Beispiel einer Parzellenunterteilung

- Optisch:

  O1 Sich schneidende Nachbarstrahlen besitzen den gleichen Schnittpunkt.

  O2 Ecke bildet einen Innenwinkel nahe 180°.

  O3 Die innere Wand ist nicht parallel zur äußeren Wand.

  O4 Konkave Gebäudeumrisse können sich selbst überschneiden.

  O5 Gebäude überschneiden sich.

Die mathematischen Sonderfälle lassen sich wie folgt lösen. Liegt die Position eines Schnittpunktes auf einem Strahl unter einem Schwellwert und somit zu nah am Rand der Parzelle, wird dieser Schnittpunkt nicht gespeichert. Ist hierdurch kein weiterer Schnittpunkt gegeben, wird mit dem Strahl wie in M1 verfahren. Sind zwei Strahlen koinzident, schneiden sich somit in jedem Punkt, werden die Schnittpunkte der betroffenen Strahlen über einen Mindestabstand vom jeweiligen Parzellenrand bestimmt (siehe Abbildung 3.3.3). Wird ein Strahl an keiner definierten[4] Stelle geschnitten, wird der Schnittpunkt über die Schnittdistanz des benachbarten Strahls berechnet. Sollte der Fall eintreten, dass keiner der Strahlen geschnitten wird, wird der Grundriss verworfen.

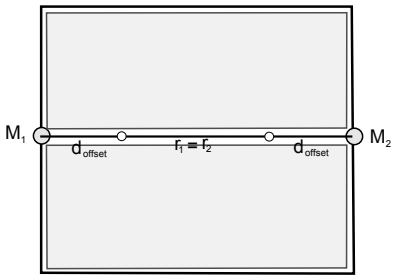


Abb. 3.3.3: Zwei koinzidente Strahlen

Neben den mathematischen Sonderfällen sind einige optische Eigenheiten zu beachten. Besitzen zwei Nachbarstrahlen den gleichen Schnittpunkt, was bei Eckparzellen der Fall ist, muss einer der identischen Eckpunkte verworfen werden (siehe Abbildung 3.3.4), da dies sonst beim späteren Schrumpfen der Fläche zu Fehlern führen würde.

[4] Ein Strahl ist nur innerhalb der von der Kontur umschlossenen Fläche gültig bzw. bis zum ersten Schnitt mit einer Außenkante.

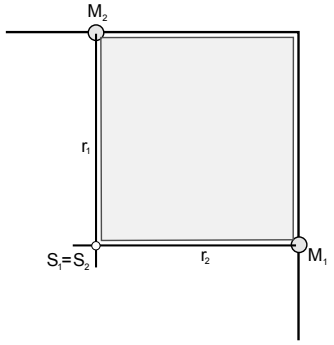


Abb. 3.3.4: Identischer Schnittpunkt

Beinhaltet eine Parzelle einen Eckpunkt, dessen Kanten einen Winkel nahe 180° aufspannen, wird dieser aus der Kontur entfernt. Der Eckpunkt würde somit auf bzw. nahe einer Geraden zwischen den Nachbareckpunkten liegen. Dies würde bei der späteren Grundrissverkleinerung zu einer doppelten Gewichtung an diesem Eckpunkt führen, was einen Knick in der Grundrissfront verursachen würde.

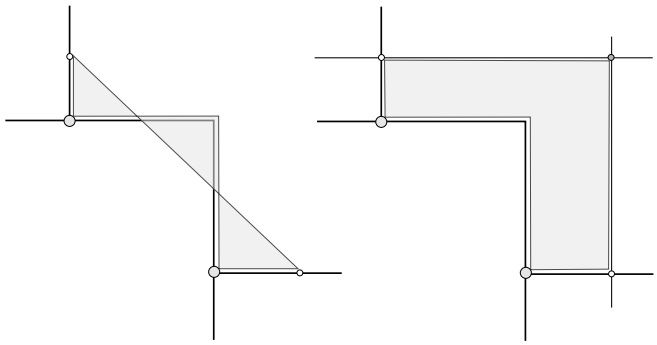


Abb. 3.3.5: Korrektur einer konkaven Fläche

Eine weitere Anpassung besteht in der Korrektur der Ausrichtung der in der Parzelle innenliegenden Außenwand. Soll dies eine parallel zur Straße orientierte Wand sein, muss die Schnittdistanz beider Strahlen auf den kleineren beider Distanzwerte gesetzt werden.

Ein weiterer Sonderfall besteht in konkaven Grundflächen (siehe Abbildung 3.3.5), die sich an Ecken befinden, da sich die ermittelte Kontur überschneiden kann. Dieses Problem lässt sich lösen, indem durch die vorderen Schnittpunkte Parallelen zur jeweils dazugehörigen Außenkante gelegt werden. Werden die Schnittpunkte dieser Parallelen in die Konturliste eingefügt, entsteht ein korrekter Gebäudeumriss.

Durch diese Behandlungen kann es, bedingt durch zu geringe Winkel an den Kanten oder zu schmale Parzellen, weiterhin zu Überschneidungen zwischen den Grundrissen innerhalb einer Parzelle kommen. Dieses Problem wird in letzter Instanz durch das gegenseitige Clippen der betroffenen Fläche mittels dem Weiler-Atherton-Algorithmus (Abschnitt 2.2.1 auf Seite 11) gelöst.

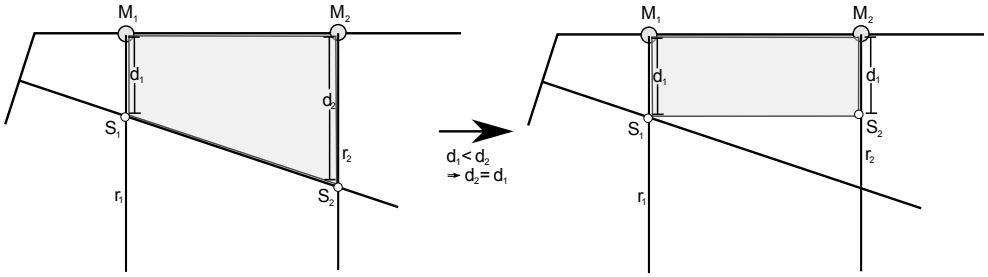


Abb. 3.3.6: Anpassung der Gebäuderückwand

Sind alle Fälle behandelt, können die so entstandenen Grundrisse verkleinert werden, so dass Gassen zwischen den Gebäuden entstehen. Hierzu werden die Eckpunkte entlang der Normalen ihrer Kanten verschoben. Dabei ist zu beachten, dass der erste bzw. letzte Eckpunkt zuvor zwischengespeichert werden muss, da sonst bei der letzten Kante mit einer bereits teils verschobenen Position gerechnet werden würde.

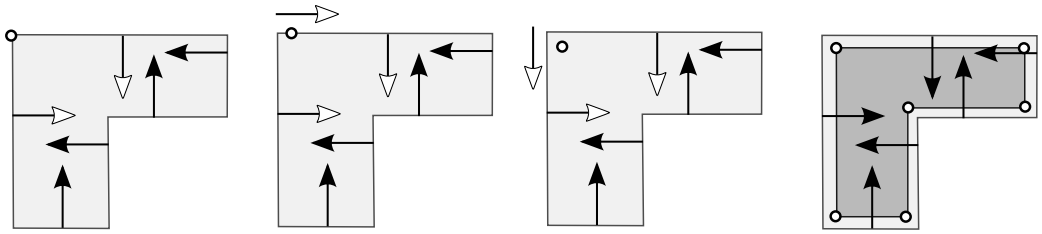


Abb. 3.3.7: Verkleinerung des Grundrisses

3.3.3 Gebäude

Für die prozedurale Gebäudeerstellung gibt es im Wesentlichen vier verschiedene Techniken: Shape-Grammars, L-Systeme, Split-Grammars und CGAs.

Shape-Grammars sind von Stiny (1980) entwickelt worden und werden u.a. in Lipp (2007) auf Verwertbarkeit für Gebäudearchitekturen untersucht. Ein Shape ist eine begrenzte Anordnung von Linien wie beispielsweise ein Dreieck oder Viereck. Diese Formen werden durch Produktionsregeln ineinander kombiniert, so dass ein Subshape innerhalb eines Shapes liegt. Eine Erweiterung für beliebige Formen stammt von Stiny aus 1982 und wird Set-Grammars genannt. Das Ergebnis der Shape-Grammars wirkt jedoch sehr künstlich und somit nicht sehr realistisch.

Extrusionsverfahren von Greuter et al. (2003) besitzt Ähnlichkeiten zu den Shape-Grammars. Es wird mit einer Grundform eines regulären Vielecks für das Dach begonnen, dessen Wände nach unten gezogen werden. Die aktuelle Grundform wird nun mit einem weiteren kleineren Vieleck verschmolzen und somit ein Zwischendach erschaffen. Die Wände werden nun erneut nach unten gezogen. Das Vorgehen wird so lange wiederholt, bis genügend Abstufungen oder die gewünschte Höhe erreicht

ist (siehe Abbildung 3.3.8). Das Ergebnis wirkt um einiges glaubwürdiger als das der Shape-Grammars.

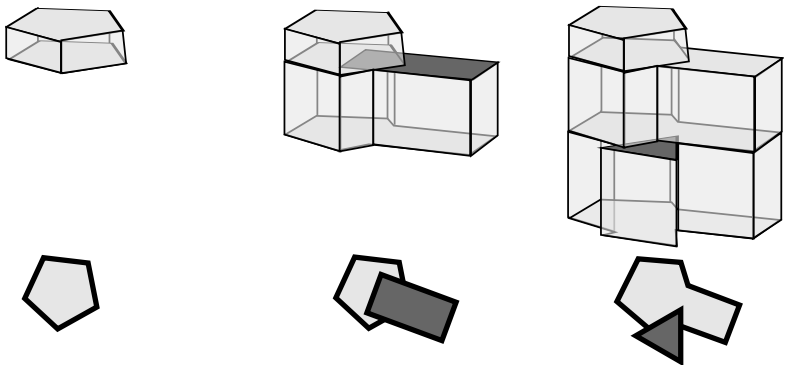


Abb. 3.3.8: Extrusionsverfahren nach Greuter zur Erschaffung von Gebäuden

Lindenmayersysteme von Prusinkiewicz/Lindenmayer finden auch Verwendung für die Gebäudegenerierung. Mit dem L-System wird die Hülle des Gebäudes erstellt, die als Zeichenkette ausgedrückt wird. Die Hülle wird in der einfachsten Variante achsensymmetrisch erzeugt, indem eine Verjüngung entlang der senkrechten Gebäudeachse nach oben hin durchgeführt wird. Der Nachteil bei dieser Technik ist die zusätzlich notwendige nachträgliche Texturierung der Gebäude.

Split-Grammars sind Kompositionen aus Basisformen und wurden von Wonka et al. (2003) eingeführt. Eine Split-Grammar ist eine Grammatik, die aus einem Vokabular von Basisformen (Zylinder, Quader, Fenster, Fensterrahmen, Türen, etc.) besteht. Sie besitzt zwei Regeltypen, die kontextsensitiv sein können: Teilungsregeln und Konvertierungsregeln.

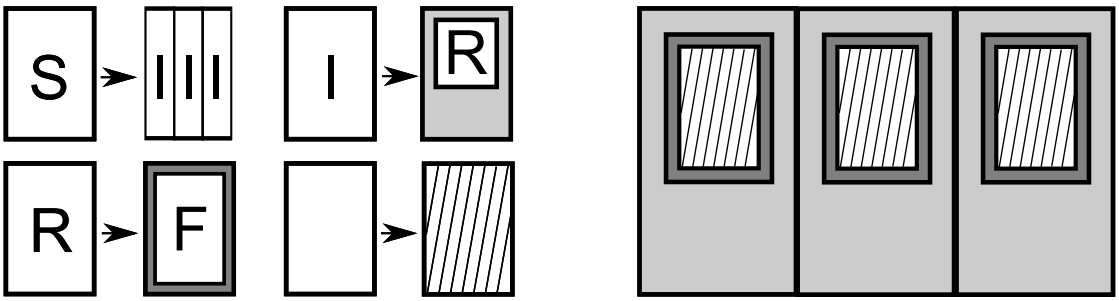


Abb. 3.3.9: Beispiel einer Shape Grammar nach Wonka et al. (2003)

Definition. Eine Grammatik $G = (N, T, R, I)$ besteht aus dem nichtterminalen Vokabular $N \subseteq U$, dem terminalen Vokabular $T \subseteq U$, einem (einer Menge von) initialen Objekt(en) $I \subseteq N$ und einer Menge von Pruduktionsregeln $R \subseteq U \times U$.

Regeln überführen einen Zustand von a nach b. Teilungsregeln ersetzen exakt ein nichtterminales Symbol aus a durch mehrere andere Symbole. Konvertierungsregeln ersetzen ein nichtterminales Symbol aus a durch ein anderes Symbol. Vorteil dieser Grammatik ist die automatische Texturerstellung und die Möglichkeit kontextspezifische Einschränkungen festzulegen.

Computer Generated Architectures (CGA) sind von Müller et al. (2006) vorgestellt worden. Der Schwerpunkt wird auf das konsistente Aussehen der Fassaden und Gebäudedächer gelegt. Es handelt sich um Verfahren zur „Massengenerierung" von Gebäuden. Die entstehenden Gebäude besitzen einen extrem hohen Detailgrad. CGAs verwenden einfache geometrische Körper wie Würfel. Diese Körper werden durch eine textuelle Regelbeschreibung der nachstehenden Form zu einem Gesamtobjekt kombiniert.

$$predecessor\,(localParameters) : conditions \rightarrow successor : probability;$$

Der Vorteil liegt in sehr realistisch aussehenden Ergebnissen, ähnlich den Split-Grammars. Der Nachteil liegt in der fehlenden lokalen Kontrolle. Um dem entgegenzuwirken, hat Lipp einen Editor entwickelt, um die Ergebnisse nachträglich zu manipulieren. Weiterhin sind CGAs bei zu hohem Detailgrad nicht echtzeitfähig.

3.3.4 Gebäudefassaden

Für die Gebäudetexturierung gibt es zwei Vorgehensweisen, die sich gegenüberstehen und kombiniert werden können.

- Eines der Vorgehen ist die vollständig manuelle Texturierung mit Bildern von realen Gebäudefassaden. Der Nachteil ist jedoch, dass das zeitintensive Eingreifen eines Modellierers nötig wird und somit keine Echtzeitfähigkeit bei zufälligen Inhalten erreicht werden kann.

- Das andere Vorgehen ist die vollständig prozedurale Texturgenerierung durch formale Spezifikation von Texturmustern. Glaubhaft wirkende prozedural erzeugte Materialien sind jedoch sehr schwierig zu erzeugen und mit sehr viel Aufwand verbunden.

Um die Nachteile beider Techniken zu reduzieren, werden die Stärken beider Techniken kombiniert. Für eine vollautomatisierte Texturierung, die realistische Ergebnisse liefert, werden semiautomatische Algorithmen verwendet. Die Eingangsbilder werden manuell oder durch Bildverarbeitungsalgorithmen untersucht und in eine Basistextur überführt. In der Basistextur gibt es feste Bereiche, beispielsweise für Fassade und Fenster (Müller, 2001). Aus diesen fest definierten Basistexturen können z.B. mit Split-Grammars (Wonka et al., 2003) automatisierte Texturierungen durchgeführt werden.

Um Variationen aus einer geringen Anzahl von Basistexturen zu erhalten, werden Modifikatoren für die Farben der Elemente verwendet, so dass z.B. anders kolorierte, hellere oder dunklere Fassaden generiert werden können.

3.4 Zwischenfazit

Die prozedurale Inhaltserzeugung im Bereich von Städten jeglicher Art ist ein bereits sehr gut untersuchtes Forschungsgebiet. Viele Verfahren zur Städtegenerierung sind bereits untersucht worden, obgleich es sich um den Nachbau existierender Städte handelt oder die Erstellung völlig neuer Städte. Methoden zur Erstellung von Straßennetzen, Parzellenunterteilung, Generierung von Gebäuden (Hochhäuser, Wohnhäuser, etc.) und deren Texturierung sind ausführlich in der Literatur behandelt worden.

Der Schwerpunkt der gefundenen Literatur liegt jedoch meist auf der prozeduralen Generierung möglichst realistisch wirkender Ergebnisse. Der Realismus der Ergebnisse führt jedoch dazu, dass die Ergebnisse nicht echtzeitfähig bleiben und somit für Computerspiele oder Trainingssimulationen für die virtuelle oder augmentierte Realität nicht zu gebrauchen sind. Zudem kommen in der Literatur meistens Algorithmen zum Einsatz, die wie zuvor erwähnt, einem strengen EVA-Prinzip[5] folgen. So ist während der Generierung kein Eingriff möglich.

Der Schwerpunkt dieser Arbeit wird daher auf die noch nicht sehr detailliert erforschte Echtzeitfähigkeit von PCG-Algorithmen gelegt. Für die Straßenerzeugung wird eine Modifikation eines Open-L-Systems verwendet, da sich das Verhalten dieses Systems leicht anpassen lässt. Eine Realisierung eines KI-Akteurs[6] zur Erzeugung von Straßenmustern bringt den Nachteil mit sich, dass für die elementaren Veränderungen des KI-Verhaltens, der Programmcode angepasst werden muss. Bei einer regelbasierten Beschreibungssprache, wie dem L-System, ist eine Verhaltensveränderung allein über das Regelwerk zu erreichen. Zudem stellen sich bei L-Systemen einige Herausforderungen. Generatorergebnisse sollen lokal beeinflussbar sein und zusätzlich ausschließlich in fest definierten Bereichen lokal erstellt werden können.

Neben dem Schwerpunkt der Straßennetzerzeugung soll eine gesamte Stadt in Echtzeit erzeugt werden, während diese durchquert wird. Hierzu werden weitere Algorithmen benötigt, die sowohl effektiv als auch schnell arbeiten. Für die Parzellenerkennung wird in Abschnitt 4.3 auf Seite 59 ein solches Verfahren vorgestellt. Die Parzellenunterteilung und Gebäudeerstellung werden mittels bestehender Techniken umgesetzt.

[5] Eingabe-Verarbeitung-Ausgabe
[6] KI: künstliche Inteligenz

Kapitel 4

4 Umsetzung

Wie zuvor erwähnt wird ein erweitertes Open-L-System umgesetzt. Eine der Herausforderungen, die virtuelle Welt lokal beeinflussen zu können, erfordert ein System, das in der Lage ist, in festen Bereichen ein Straßennetz wachsen, entfernen und neu schaffen zu können. Die Umsetzung dieser Fähigkeiten wird in Abschnitt 4.2 auf der nächsten Seite im Detail beschrieben. Die Umsetzung der Techniken zur restlichen Stadtgenerierung werden in Abschnitt 4.3 auf Seite 59 und Abschnitt 4.4 auf Seite 64 erläutert.

Für das Szenario einer futuristischen Großstadt werden in diesem Kapitel zusätzlich Möglichkeiten für normalerweise physikalisch unmögliche Städte vorgestellt. Futuristische Städte besitzen Charakteristiken, die im Weiteren untersucht und behandelt werden. Solche sind beispielsweise schwebende Transportplattformen, Hochhäuser und Straßen.

4.1 Softwaretechnischer Entwurf

Für die Planung eines komplexen Systems mit zwei umfangreichen Komponenten, dem Front- und Backend, ist ein softwaretechnischer Entwurf erstellt worden.

Das Frontend (4.1.2) wurde so konzipiert, dass es über die Schnittstelle des Backends (CityDll) Daten anfordern kann und nur für die Geometrieaufbereitung und -darstellung sowie Grundlagen zuständig ist. Im Diagramm sind importierte Programmteile u.a. der Grafikengine OGRE dunkel eingefärbt. Es ist zu sehen, dass das Hauptprogramm (City-Interface) das Rendering und die Steuerung über Maus und Tastatur an OGRE abgibt. Die erwähnte Geometrieaufbereitung erfolgt über die „MeshFactory“, sie überführt die Geometriebeschreibungen in Geometriemodelle, die von OGRE dargestellt werden können.

Die Spiellogik wird in der Klasse „Player“ abgelegt, die über den „TrackController“ eine Navigation über das Straßennetz ermöglicht. Für andere dynamische Inhalte, neben der Spielerbewegung, wird der „AnimationController“ eingesetzt. Er ist eine Schnittstelle zu OGRE um Animation, wie z.B. die bewegbaren Plattformen zu aktivieren und zu deaktivieren.

Im Backend findet eine Unterteilung in mehrere Aufgabenbereiche statt. Jeder Aufgabenbereich besitzt eine eigene Verwaltungsklasse (Factory). Für die Erstellung von

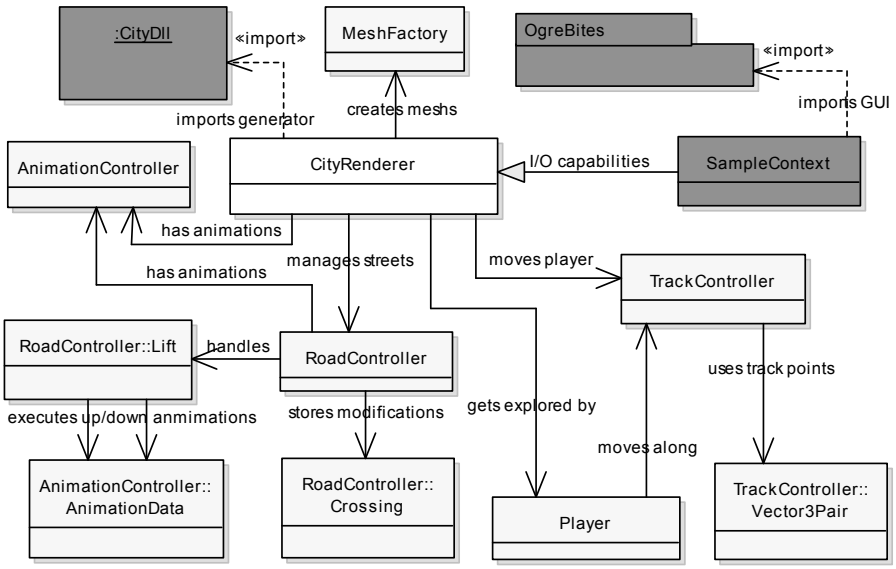


Abb. 4.1.1: Vereinfachtes Klassendiagramm des Frontends

Straßen (StreetFactory), Parzellen (LotFactory) und Gebäuden (BuildingFactory) gibt es je eine solche Klasse. Die StreetFactory verwendet als Regelsystem das L-System, das wiederum die Veränderungen an die Datenstruktur weitergibt (Graph). Über die Datenstruktur können Auswertungen, wie die Parzellenerkennung durch die LotFactory, ausgeführt werden. Die StreetFactory kann u.a. über die Triangulationsklasse Geometrienetze der Straßenkreuzungen erstellen. Die Triangulation kann jedoch auch extern durch das Frontend verwendet werden, um einfache Geometrievorschläge zu erhalten. Die rot hinterlegten Klassen Config und Tracer sind Systemklassen, auf die aus jeder Klasse zugegriffen werden kann. Sie ermöglichen globale Funktionalitäten wie das Logging des Programmablaufs oder das Laden globaler Parameterwerte.

Alle berechneten Ergebnisse werden in einer vereinfachten Datenrepräsentation zurückgegeben, so dass keine komplexen Austauschformate nötig sind. Alle benötigten Formate werden in einer einzigen kompakten Headerdatei (PublicDataTypes.h) spezifiziert.

4.2 Programmablauf

Dieser Abschnitt der Arbeit beschäftigt sich im Wesentlichen mit der Implementierung des Backends und dessen Schnittstelle. Die Umsetzung des Frontends wird im Detail in Kapitel 6 auf Seite 89 erläutert. Auf eine detaillierte Beschreibung des Quellcodes wird absichtlich verzichtet, um stattdessen näher auf die verwendeten Verfahren, Techniken

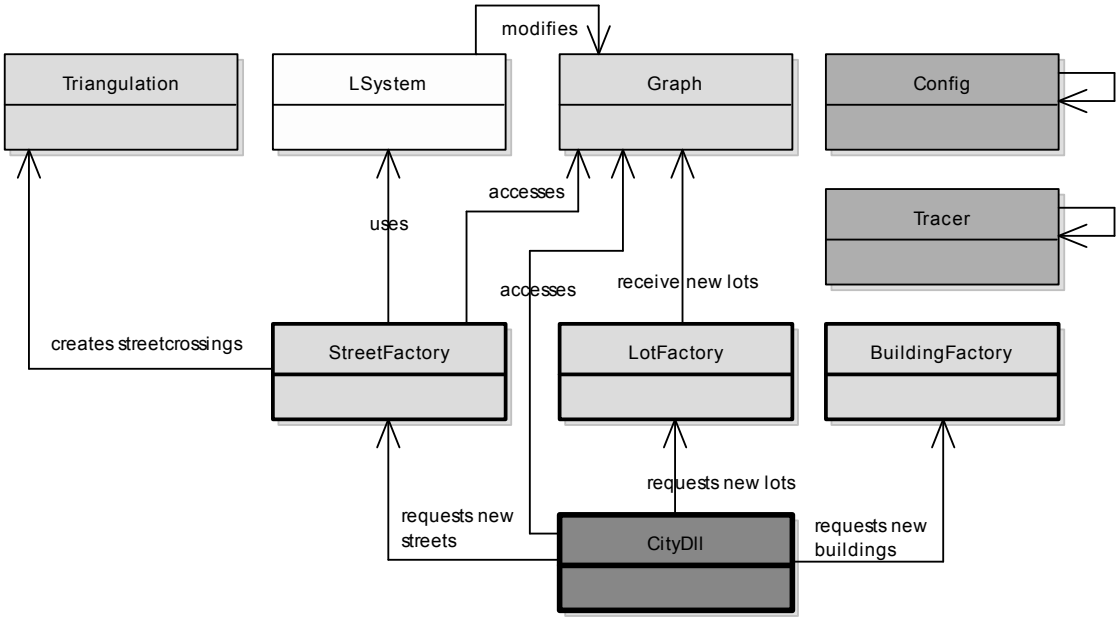


Abb. 4.1.2: Vereinfachtes Klassendiagramm des Backends

und daraus entstandenen Schwierigkeiten eingehen zu können. Der dokumentierte Quellcode sowie die kompilierte Endanwendung einschließlich der Bibliothek CityDll sind auf der beigelegten CD zu finden.

4.2.1 Verschiedene Regelsysteme

In Kapitel 3 auf Seite 29 sind existierende Ansätze beschrieben worden und die Mächtigkeit des Lindenmayersystems hervorgehoben worden. Das Backend ist so konzipiert, dass die für das Regelsystem zuständige Klasse einfach durch eine andere ausgetauscht werden kann. Für die Arbeit wurde das L-System umgesetzt, da dies die größte Vielfalt bietet.

4.2.2 Programmablauf mit L-System

Der Ablauf des Generierungsprozesses wird in den nächsten Abschnitten im Detail beschrieben. Als Überblick soll der nachfolgende Pseudocode dienen.

Listing 4.1: Pseudocode des Programmablaufs des Generators

```
1  Erzeuge eine StreetFactory
2    Wähle ein Regelsystem: LSystem
3    Lade LSystem
4      Auswertung der Regeldatei mit regulären Ausdrücken
5      Erstelle ein LSystem mit Regeln
6    Verbinde Regelsystem mit Datenstruktur
7  Aktualisiere Straßennetz mittels StreetFactory
8    Erweitere Regelzustand (LSystem)
9      Ermittlung umliegender Knoten (U)
10     for alle u in U do
11       if u ist ersetzbar then
12         Erweiterungsregel suchen
```

```
13          for alle r in Regeln do
14            if Regelkopf von r ist passend then
15              if Bedingungen von r gültig then
16                if Bedingung einfach then
17                  Variablen ersetzen
18                  Arithmetik auswerten
19                else if Bedingung sensitiv then
20                  Zuweisungen ausführen
21                  Kollisionstest durchführen
22                  Ergebnis des Tests auswerten
23                  Korrekturen gegebenfalls speichern
24                end if
25              end if
26            end if
27          end for
28          if Regelanzahl == 1 then
29            wähle diese Regel
30          else
31            wähle eine Regel über deren Wahrscheinlichkeit
32          end if
33          Gewählte Regel auswerten
34            for alle w in Regelworte do
35              for alle p in Paramterliste von w do
36                Funktionen im Parameter auswerten
37                Variablen ersetzen
38                Arithmetik auswerten
39              end for
40              Regelwort nun als Regelfunktion speichern
41            end for
42          Platzhalter durch Regelfunktionskette ersetzen
43        end if
```

Der Pseudocode enthält und beschreibt Techniken aus verschiedenen Klassen (siehe Abbildung 4.2.1) am Beispiel des Regelmoduls für L-Systeme.

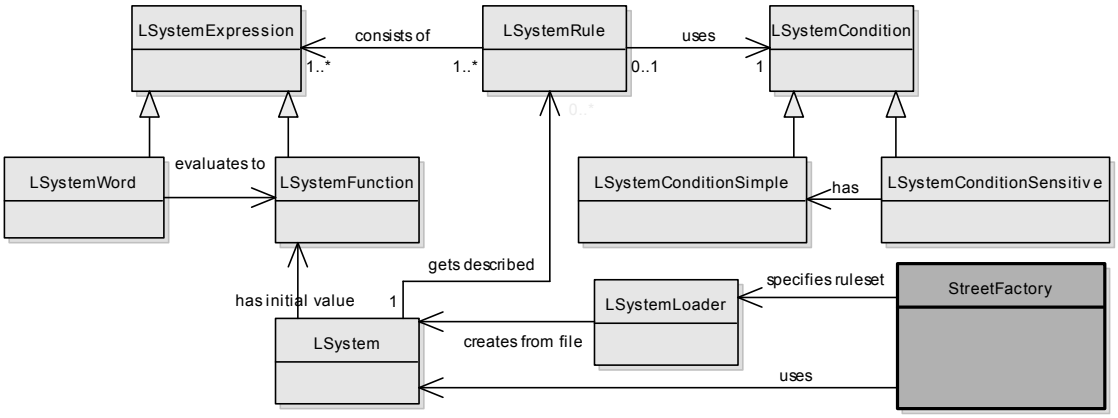


Abb. 4.2.1: Vereinfachtes Klassendiagramm des L-Systems

StreetFactory Basisschnittstelle zum Erzeugen neuer Inhalte (Wahl des Regelmoduls, Ausführen des Regelsatzes, ...)

LSystem Das verwendete Regelsystem (Ausführen des System, Regelsatzersetzungen, Graphmanipulationen, ...)

LSystemRule Eine einzelne Regel des L-Systems

LSystemExpression Ein einzelnes unausgewertetes (*LSystemWord*) bzw. ausgewertetes (*LSystemFunction*) Wort einer Regel.

LSystemCondition Bedingung einer Regel, damit diese zutrifft. Kann ein einfacher arithmetischer Ausdruck (*LSystemConditionSimple*) oder eine komplexe umgebungssensitive Bedingung (*LSystemConditionSensitive*) sein.

LSystemLoader Liest eine Regeldatei ein und erstellt ein L-System mit Regeln, Startzustand und globalen Variablen.

4.2.3 Parameter/Initialdaten

Vor der Implementierung ist festgelegt worden, wie das Generatorergebnis beeinflusst werden kann. Dies kann über den Parser des Regelsystems (L-System-Parser) und globale Eigenschaften geschehen, die sich nur umständlich im Regelsystem ausdrücken lassen würden. Diese globalen Werte werden in einer Konfigurationsdatei bestimmt. Zusätzlich können Parameter, die mit dem Frontend verändert werden können, über die Schnittstelle des Backend angegeben werden.

Das während der Entwicklung verwendete L-System erstellt beispielsweise Straßen immer nach einer einfachen Regel, bei der an jedes Straßensegment jeweils ein umgedrehtes T-Stück (siehe Abbildung 4.2.2) angefügt wird. Die Länge der Segmente dieses T-Stücks und die Winkel zwischen diesen können dabei um im Regelwerk angegebene Intervalle variieren.

Übergreifend wird die Konfiguration verwendet, um z. B. einen Minimalwinkel zwischen den Straßen anzugeben oder die Minimal- und Maximalhöhe von Gebäuden anzugeben.

Das Frontend gibt an die Schnittstelle in regelmäßigen Abständen die Anwenderposition und einen Generierungsradius weiter, in der die Stadt aktualisiert werden soll. Der Radius kann vom Frontend beispielsweise über die Rechengeschwindigkeit des verwendeten PCs bestimmt werden.

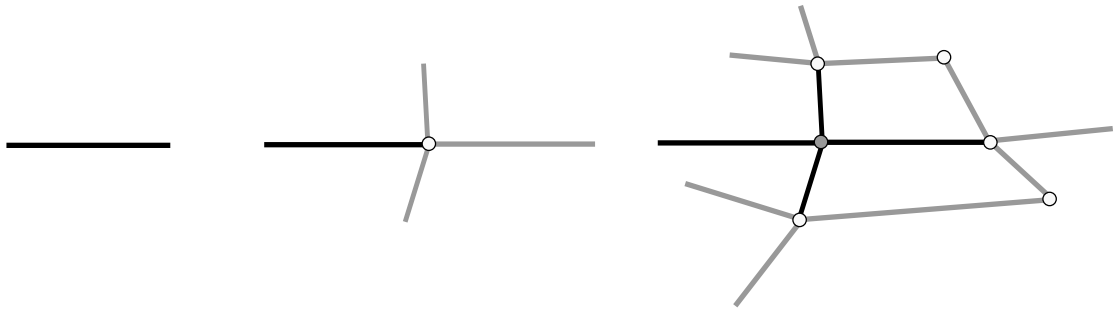


Abb. 4.2.2: Mögliches Ergebnis eines L-Systems durch iteratives Anfügen von T-Stücken

4.2.4 Einlesen eines Regelsatzes

Die Grundlage für die Erstellung von Straßen ist das Regelwerk. Das Lindenmayersystem wird aus einer einfachen Textdatei eingelesen. Das System wird in einer festen Syntax an-

gegeben und ist an ein parametrisches L-Systems angelehnt. Das folgende Beispiel ist ein modifizierter Auszug aus Pascal Müller (2001) zur Erstellung eines Manhattanmusters.

Listing 4.2: Regelsatz eines L-System zur Straßenerzeugung

```
1 #define ProbChangeStreetAngle 0.20
2 #define DevStreetAngle 10
3 #define ProbChangeStreetLength 0.20
4
5 o: S(10)
6 p1: S(len) > ?I(a,len,insertion) : insertion == NORMAL {a = ProbChangeStreetAngle ? U[-
      DevStreetAngle,DevStreetAngle] : 0.0; len = ProbChangeStreetLength ? U[0.5*len,1.5*len] :
      len; } -> &(a)F(len)[+(80)P][&(90)S(len)][&(-90)S(len)]&(a)S(len)
7 p2: S(len) > ?I(a,len,insertion) : insertion == NN || insertion == INTSCT || insertion ==
      FWD_INTSCT -> &(a)F(len)
```

Der Regelsatz ist in drei Teile unterteilt: Variablendefinition, Startzustand und Regeln. Zum Einlesen werden reguläre Ausdrücke verwendet, die es ermöglichen, Zeichenketten komfortabel zu unterteilen.

Um den ersten Teil zu erkennen, wird ein regulärer Ausdruck verwendet, der nach Zeilen sucht, die mit „#define“ beginnen und anschließend ein alphanumerisches und numerisches Wort enthalten. Der jeweilige Variablenname und -wert wird über den Gruppenindex eins und zwei abgefragt, der durch die Klammerung zustande kommt.

```
1 #define ([a-zA-Z0-9]+) ([0-9.]+)
```

Der nullte Index entspricht immer dem gesamten erkannten Ausdruck.

Sobald eine Zeile nicht mehr diesem Ausdruck entspricht, wird nach dem zweiten Teil gesucht. Im zweiten Teil wird mit einem regulären Ausdruck der Startzustand ermittelt. Dieser kann beispielsweise die Form „o: S(10)“ besitzen. Passend sind alle Belegungen, die mit einem alphanumerischen Wort beginnen, das gefolgt wird von einem Doppelpunkt und abschließend ein Funktionswort enthält. Ist dieser Zustand gefunden, wird auf den letzten Zustand umgeschaltet.

Der dritte und letzte Teil enthält die Regeln. Es wird zuerst versucht die Regeln im Ganzen zu erkennen und anschließend diese in Einzelteile zu zerlegen. Mit einem regulären Ausdruck können optionale Regelteile definiert werden. So sind die Regeln „p1“ und „p2“ aus Listing 4.2 zutreffend, jedoch auch die minimale Regel „p3: A -> F(10)“, die keine optionalen Regelbereiche beinhaltet.

Ein Regelsatz besteht bei Angabe aller Informationen aus fünf Feldern: Kopf, Kopfparameter, Bedingung, Regel und Wahrscheinlichkeit. Die Bedingung und Regel lassen sich weiter unterteilen. Die Unterteilung erfolgt dann innerhalb der jeweiligen Programmklassen.

Zuordnen einer Regelbedingung

Eine Regelbedingung kann eine einfache arithmetische oder eine kontextsensitive Bedingung sein. Eine arithmetische Bedingung der Form *„A>10 && B<20“* wird in seine Bestandteile zerlegt und im späteren Verlauf ausgewertet.

Eine kontextsensitive Bedingung ist aufwendiger auszuwerten, da diese eine Reihe optionaler Angaben besitzen kann. Die kontextsensitive Bedingung der Regel *„p1“* aus Listing 4.2 entspricht dem nachstehenden Ausdruck.

```
1 ?I(a,len,insertion) :
2   insertion == NORMAL {
3     a = ProbChangeStreetAngle ? U[-DevStreetAngle,DevStreetAngle] : 0.0;
4     len = ProbChangeStreetLength ? U[0.5*len,1.5*len] : len;
5   }
```

Dieser Ausdruck besteht aus einer sensitiven Funktion, sie beginnt mit „*?*“. Auf die Funktion folgt eine Bedingung, die für die Regelausführung eingehalten werden muss. Die zwischen „{“ und „}“ stehenden Zuweisungen werden vor der sensitiven Funktion ausgeführt.

Diese Zuweisungen müssen separat ausgewertet werden, da sie optional sind. Es findet eine Unterteilung der einzelnen Zuweisungen am Semikolon statt. Die Zuweisungen können wiederum in einfacher Form (*„a = 10“*) erfolgen oder binäre Bedingungsoperatoren („Bedingung ? Abschnitt wenn wahr : Abschnitt wenn falsch“) verwenden. Die Zuweisung kann eine Funktion sein, die wiederum arithmetische Ausdrücke enthalten kann. Die Funktion *„U[a,b]“* generiert eine uniform verteilte Zufallszahl innerhalb des Intervalls [a..b], womit bei jeder Regelausführung ein anderes Verhalten erreicht werden kann.

Zerlegung einer Regel

Bevor eine Regel verwendet werden kann, muss diese in Regelworte und anschließend jedes dieser Worte in Name und Parameter zerlegt werden. Die Parameter können wiederum terminale Funktionen sein wie z.B. *„random(a,b)“*. Jede Regel wird intern als Liste von *LSystemWord*s gespeichert. Dieses *LSystemWord* kann im späteren Verlauf ausgewertet werden und wird zu einer *LSystemFunction* konvertiert.

4.2.5 Erweiterung des Regelzustandes

Da das Regelwerk eines Lindenmayer-Systems unendlich sein kann, findet eine Auswertung ausschließlich iterativ statt. Normalerweise wird die Ersetzung von Platzhaltern im Zustand eines L-Systems von links nach rechts ausgeführt. Da dies jedoch ohne geometrischen Zusammenhang geschieht, wird dieses Vorgehen erweitert.

Die Erweiterung besteht darin, dass nur Platzhalter innerhalb der vorgegebenen Weltgrenze ersetzt werden. Bei der Verschiebung der Weltgrenze rücken bereits erzeugte Bereiche aus dieser hinaus und neue Bereiche in diese hinein. Findet eine Ersetzung unabhängig der Iterationstiefe statt, können in Bereichen, die länger innerhalb der Weltgrenze gelegen haben, detailliertere Bereiche entstehen als in anderen Bereichen. Diesem Problem wird entgegengewirkt, indem jedes Symbol zusätzlich eine Information über dessen Iterationstiefe erhält. Aufgrund diesem Wert werden Platzhalter mit geringerer Tiefe bevorzugt behandelt.

Füllen der Weltgrenze

Damit steht fest, in welcher Reihenfolge die Platzhalter ersetzt werden, jedoch nicht wie das Abbruchkriterium aussieht. Das Kriterium besteht aus mehreren Unterkriterien.

Es werden solange die Platzhalter innerhalb der Grenze mit der geringsten Iterationstiefe ausgewählt, bis die Ersetzung entweder zu kleine Kantensegmente oder gar keine mehr erzeugen würde. Zusätzlich wird die Anzahl der durch die Ersetzungen kreierten Knoten innerhalb der Weltgrenze gezählt. Unterschreitet die Anzahl der Knoten einen Schwellwert, wird die Befüllung der Welt ebenso abgebrochen.

4.2.6 Erweitern eines Platzhalters

In einem L-System kann jedes Wort durch eine entsprechende Ersetzungsregel als ein Platzhalter fungieren. Um festzustellen, ob es sich um einen Platzhalter handelt, wird versucht im Regelsatz eine passende Ersetzung zu finden. Ein zu ersetzendes Wort könnte die Form „*S(10)*“ haben. Um einen schnellen Abgleich zu gewährleisten, wird in einer vorgegebenen Reihenfolge eine Prüfung durchgeführt, wie es in Listing 4.3 beschrieben wird.

Listing 4.3: Pseudocode für den Regelabgleich

```
1  LSystemRule* getMatchingRule(Name, Parameterwerte) {
2    for (alle r in Regeln) {
3      if (Kopf von r != Name) weiter;
4      if (Parameteranzahl von r != |Paramterwerte|) weiter;
5      if (r hat keine Bedingung) {
6        Regel merken;
7      } else { // Auswertung der Bedingung
8        Erstelle eine Parameternamewert-Liste kvl;
9        bool result = Ergebnis der Bedingung von r(kvl);
10       if (result)
11         Regel merken;
12       else
13         weiter;
14     }
15   }
16   Summe(Wahrscheinlichkeit aller gefundenen Regeln) == 1;
17   if (Anzahl gemerkter Regeln == 1) {
18     return Gefundene Regel;
19   } else { // zufällige Auswahl einer Regel
20     Erzeuge eine Zufallszahl n in [0..1];
21     Definiere Gültigkeitsintervalle (I) der Regeln [0..r1, r1..r2, r2..1];
22     return Regel an der Stelle n aus I;
23   }
24   return Keine Regel;
25 }
```

Das Wort „*S(10)*“ würde somit nicht auf die Regel „*X()*“, „*X(a)*“, „*S()*“ oder „*S(a,b)*“ zutreffen, sondern nur auf eine Regel der Form „*S(a)*“. Sollte die Regel „*S(a)*“ zusätzlich eine Bedingung besitzen, muss diese ebenfalls wahr werden, damit die Regel in die Vorauswahl übernommen wird.

4.2.7 Einfache arithmetische Bedingungen

Besitzt eine Regel eine arithmetische Bedingung, ist eine Auswertung dieser nötig. Eine solche Regel könnte beispielweise „*S(a)>a>5 && a<20 → ...*“ sein. Sie verlangt, dass ihr Parameter zwischen 5 und 20 liegt. Für diese Aufgabe wurde ein Parser für einfache arithmetische Ausdrücke geschaffen. Unterstützt werden die Vergleichsoperatoren „>“, „>=“, „<“, „<=“ und „==“, sowie eine ODER- („||“) und UND-Verkettung („&&“). Bei der Regelerstellung ist der Ausdruck bereits in die Form „<linker Operand><Operator><rechter Operand>[<Verkettung>]“ gebracht worden. Somit können

Operanden, die globale Variablen oder ein Eingabeparameter sind, einfach ersetzt werden. Anschließend wird der logische Ausdruck auf den Programmcode abgebildet und das Ergebnis zurückgegeben.

Die Regel würde nach der Auswertung aus Tabelle 4.2.1 ausgeführt werden, da der Aufruf von „*S(10)*“ somit wahr wäre.

	linker Operand	Operator	rechter Operand	Verkettung
1.	a	>	5	&&
2.	a	<	20	NULL

↓

	linker Op.	Operator	rechter Op.	*Ergebnis*	Verkettung
1.	10	>	5	*true*	&&
2.	10	<	20	*true*	NULL

Tabelle 4.2.1: Auswertung eines arithmetischen Ausdrucks

4.2.8 Selbstsensitive Bedingungen

Soll eine Regel nur bei Eintreten eines Umgebungskriteriums ausgeführt werden, muss die geometrische Umgebung des Platzhalters, der die aktuelle Bedingung enthält, entsprechend untersucht werden. Das umgesetzte System kennt zwei solcher Umgebungskriterien: „*?I(inOut Winkel, inOut Länge, out Detailergebnis)*“, überprüft auf Kollisionen mit der eigenen L-Systemausgabe und „*?E(inOut Winkel, inOut Länge, out Detailergebnis)*“ prüft auf Kollisionen, wie Wasser auf der Umgebungskarte. Die Prüfung, ob ein Segmentende in einem gesperrten Bereich liegt, erfolgt durch einfaches Abgleichen des Segmentverlaufs mit den Pixelwerten einer Textur, die die Weltkarte repräsentiert. Die eigentliche Komplexität besteht in der Prüfung des Endsegmentes mit dem aktuellen L-Systemergebnis. Hierfür ist speziell die Architektur des L-System-Programmteils angepasst worden. Ein L-System wird normalerweise erst vollständig ausgewertet und somit die Ersetzungen bis zu einer vorgegebenen Iterationstiefe ausgeführt. Anschließend wird der L-Systemstatus in eine geometrische Repräsentation überführt. Bei der gewählten Architektur findet eine direkte Auswertung statt, so dass jederzeit der geometrische Zustand bekannt ist.

Die Umgebungsfunktionen sind ähnlich zu einem normalen L-Systembefehl zu verstehen. Es wird eine Rotation, gefolgt von einer Zeichenoperation ausgeführt, nur dass der Zustand des L-Systems nicht verändert wird. Es wird lediglich geprüft, wie sich die eingefügte Symbolfolge verhalten würde. Den Funktionen werden Winkel und Zeichenlänge übergeben, um mit diesen Werten arbeiten zu können. Die Umgebungsfunktionen können die Funktionswerte, falls nötig, jedoch verändern. Beispielsweise kann durch den Umgebungstest eine Straße, die im Wasser enden würde, verhindert oder so korrigiert werden, dass sie entlang des Ufers verläuft oder kurz vor diesem endet.

Ob die initialen Werte verändert worden sind, liefert die Funktion als Ergebnis. Die Eingabewerte dieser Umgebungsfunktionen werden über die Parameterliste der Regel

oder über eine vorausgehende Zuweisung (siehe Abschnitt 4.2.4 auf Seite 46) gesetzt.

4.2.9 Kollisionsselbsttest

Mit den gesetzten Initialdaten kann nun ein Kollisionstest mit den Kanten des bereits erstellten Graphen des L-Systems durchgeführt werden.

Zur Erkennung geometrischer Zusammenhänge zählt das Auffinden eines bestehenden Knotens am Ende eines einzufügenden Segmentes. Die Erkennung eines Schnitts mit anderen Kanten[1] des Graphs entlang des vorgeschlagenen Segments und die Identifizierung nahegelegener Knoten entlang des gesamten neuen Segments ist ebenfalls zu realisieren.

Um eine einfache und schnelle Suche zu ermöglichen, werden alle geometrischen Tests in einem projizierten 2D-Raum durchgeführt. Dreidimensionale Einschränkungen müssen dadurch zusätzlich behandelt werden, da sich das generierte Straßennetz nicht nur in einer Ebene bewegt, sondern sich auch in die Höhe erstrecken kann.

Erkennung eines nahegelegenen Knotens (nearest neighbour)

Zur Erkennung von Knoten, die nahe an dem vorgeschlagenen Segment liegen, wird um dieses eine Hülle gebildet. Diese Hülle erstreckt sich vom Streckenbeginn bis leicht über das Streckenende hinaus. Die Kontur dieser Hülle besitzt zum enthaltenen Segment jeweils den gleichen Abstand (ausgenommen vom Beginn). Es wird nun nach Knoten gesucht, die innerhalb dieser Hülle liegen. Knoten mit anderen Höhenwerten werden nicht beachtet. Der Vorteil einer Hülle dieser Art gegenüber einer achsenorientierten Boundingbox (AABB) um den Endpunkt besteht darin, dass zum Einen Knoten durch die falsche Ausrichtung ein- bzw. ausgeschlossen werden (Abbildung 4.2.3) und zum Anderen auch Knoten in der Nähe der gesamten Strecke gefunden werden können.

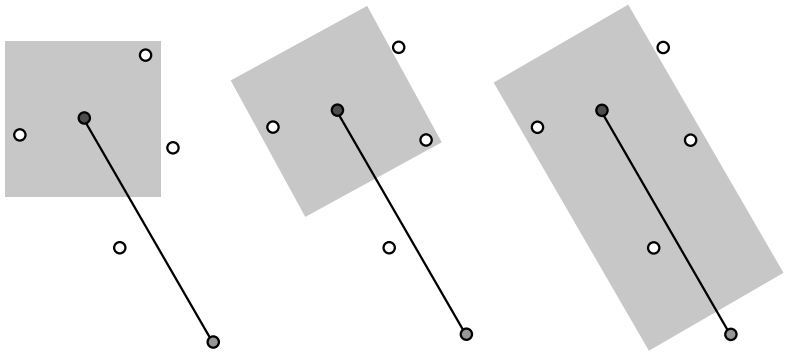


Abb. 4.2.3: Vektororientierte Bounding Box

Ob ein Segment als „nearest neighbour“ (NN) gewählt wird, hängt von einigen Kriterien ab. Das Segment, das zwischen dem NN und dem vorgeschlagenen Kantenbeginn

[1] Ein Segment ist losgelöst vom Graphen. Ein Segment, das in den Graph integriert wird, wird zur Kante.

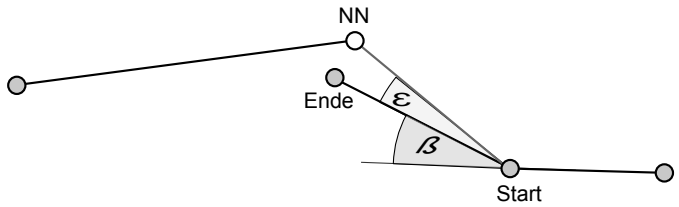


Abb. 4.2.4: Neuberechnung des Transformationsparameters

gebildet werden würde, muss eine Mindestlänge besitzen und es darf an der Anschlussstelle kein zu kleiner Winkel entstehen.

Durch die Wahl eines nahegelegenen Knotens verändert sich nicht nur die Endposition des vorgeschlagenen Segmentes, sondern auch die Parameter, die zu dieser geführt haben. Diese Parameter bestehen aus Transformationswinkel (β) und -länge und werden erneut berechnet, indem die Länge auf den Abstand zwischen Startknoten und NN gesetzt wird (Abbildung 4.2.4). Der Winkel wird auf $\beta + \epsilon$ korrigiert durch Berechnung des Winkels (ϵ), der vom vorgeschlagenen Segment und dem nun gefundenen Segment zwischen Start und NN aufgespannt wird.

Erkennung eines Schnitts/Vorwärtsschnitts

Hierbei wird zwischen normalen Schnittpunkten mit anderen Segmenten und jenen unterschieden, die über das Segmentende hinaus zustande kommen würden. Hierfür wird das vorgeschlagene Segment um einen relativen Wert verlängert. Würde dies nicht geschehen, könnten Straßenkreuzungen in unmittelbarer Nähe bestehender Straßen zustande kommen. Anschließend werden alle Geraden ermittelt, die das verlängerte Segment schneiden würden und deren Schnittpunkte errechnet. Über diese Schnittpunkte wird errechnet, wie weit diese vom Startpunkt des verlängerten Segments entfernt sind. Anschließend wird über die geringste Distanz der vorderste Schnittpunkt gewählt. Ob ein Schnittpunkt zulässig ist, wird nun über eine Reihe von Kriterien entschieden.

- Der Schnitt darf das geschnittene Segment in nicht zu kleine Segmente unterteilen.
- Die Länge bis zum Schnittpunkt darf nicht zu kurz sein.
- Die Winkel, die zwischen dem geschnittenen und schneidenden Segment entstehen, dürfen nicht zu klein werden.

Sind all diese Kriterien erfüllt, wird der Schnittpunkt sowie die Kanten, auf der er sich befindet, gespeichert. Ob es sich um einen Vorwärtsschnitt handelt, wird ermittelt, indem geprüft wird, ob die Schnittdistanz oberhalb der initialen Segmentlänge liegt.

Zudem ist zu beachten, dass im projizierten Raum Schnitte zustande kommen können, die eigentlich nicht existieren. In Abbildung 4.2.5 auf der nächsten Seite ist ein Spezialfall an zwei Beispielen zu sehen, bei denen eigentlich kein Schnittpunkt existiert. Im linken Bild verläuft ein steiles Segment über ein darunterliegendes Segment hinweg. Um in der Projektion keinen Schnitt zu erhalten, ist vorher zu überprüfen, ob die Start- und

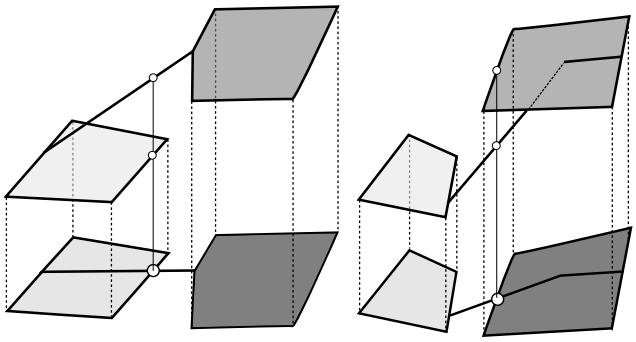


Abb. 4.2.5: Projizierte Schnittpunkte, die in 3D nicht existieren

Endpunkte der Segmente den gleichen Höhenwert besitzen. Sollten die Höhenwerte sich unterscheiden, wird ein erkannter Schnitt verworfen. Im rechten Bild findet der Schnitt mit dem darüberliegenden Segment statt und kann auch über die Höhenwerte abgefangen werden.

Auswertung des Kollisionsergebnisses

Das Kollisionsergebnis enthält Daten über Eingangsdaten und manipulierte Ausgangsdaten. Je nach Kollisionsergebnis können sich diese unterscheiden. Im Fall eines „nearest neighbours“ wird eine Referenz zu diesem gespeichert. Im Fall eines Schnitts wird die Kante, auf der der Schnitt liegt, und die Schnittposition gespeichert. In allen anderen Fällen wird lediglich gespeichert, dass eine Einfügung möglich bzw. nicht möglich ist. Je nach Regelbedingung wird das Schnittergebnis („nearest neighbour“, Schnittpunkt, keine Kollision oder keine Einfügung möglich) erneut als arithmtischer Ausdruck ausgewertet.

Sei „$?I(a, l, insertion)$“ mit $a = 0, l = 10$ gegeben, das Ergebnis von $insertion = NN$ und die Bedingung:

```
1 insertion == NN || insertion == INTSCT || insertion == FWD_INTSCT
```

Mit folgender Auswertung wird die Bedingung dann erfüllt:

	linker Op.	Operator	rechter Op.	*Ergebnis*	Verkettung
1.	insertion = NN	==	NN	*true*	||
2.	insertion = NN	==	INTSCT	*false*	||
3.	insertion = NN	==	FWD_INTSCT	*false*	NULL

Tabelle 4.2.2: Auswertung eines Kollisionsergebnisses

4.2.10 Auswahl einer Regel

Ist eine Regel dadurch zutreffend, dass keine Bedingung existiert oder diese erfüllt wird, ist zu überprüfen, ob mehr als eine passende Regel gefunden worden ist. Die Regelsätze

sind so konzipiert, dass bei mehr als einer zutreffenden Regel ein Wahrscheinlichkeitswert zwischen null und eins angegeben ist. Sind mehrere Regeln gefunden worden, werden diese auf einem Intervall zwischen null und eins verteilt. Sollte die Summe der Wahrscheinlichkeiten nicht eins sein, wird das Ergebnis skaliert und eine Warnung ausgegeben. Anschließend wird eine Zufallszahl (siehe Abschnitt 4.2.10) im Intervall [0..1] generiert und die entsprechende Regel gewählt.

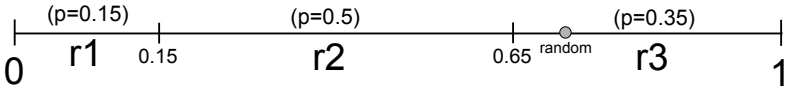


Abb. 4.2.6: Zufällige Auswahl einer Regel

Pseudozufallszahlen (Seeds)

An einigen Stellen werden uniform verteilte Zufallszahlen benötigt, um ein glaubwürdiges und zufällig wirkendes Stadtbild zu erzeugen. Über den Zufallszahlengenerator des Systems werden jedoch keine echten Zufallszahlen erzeugt. Es wird abhängig vom Seed[2] eine Zahlenreihe erzeugt, die aus voneinander unabhängigen Zahlen besteht. Um bei jedem Erstellungsprozess ein anderes Ergebnis zu erhalten, wird als Seed die Systemzeit verwendet, da sich diese jede Sekunde verändert.

Ein Problem entsteht jedoch bei der Verwendung von Threads. Da die Funktionen zur Initialisierung des Generators (*srand(seed)*) und zur Erzeugung der Zahlen (*rand()*) nicht threadsicher sind, ist das Verhalten dieser systemabhängig und nicht sichergestellt. Um ein zuverlässiges Verhalten zu erreichen, ist es nötig im Hauptprogramm die Seeds zu bestimmen und an die Threads weiterzugeben, so dass diese den Generator für jede Verwendung korrekt initialisieren können. Würde dies nicht geschehen, kann dies dazu führen, dass in Threads ausgelagerte Funktionen, die auf Zufallszahlen basieren, immer das gleiche Ergebnis liefern.

Durch die Abhängigkeit der Zahlenreihen von ihrem Seed ist es möglich, generierte Reihen erneut zu erzeugen. Dies ist zwar kritisch in kryptographischen Anwendungen, da hierdurch auf die Verschlüsselung zurückgeschlossen werden kann, ist jedoch für die Erzeugung von Stadtmustern optimal geeignet. Die Möglichkeit der Wiederherstellbarkeit von Zahlenreihen ermöglicht es, ältere Bereiche des generierten Ergebnisses zu verwerfen und diese, wenn nötig, über den Seed zu einem späteren Zeitpunkt wieder herzustellen. Die Performanz des Stadtgenerators kann dadurch enorm gesteigert werden, da die Menge an Kanten und Knoten so für Berechnungen stark reduziert werden kann.

4.2.11 Auswerten einer Regel

Eine Regel besteht aus beliebig vielen Regelworten wie beispielsweise „$F(len \cdot 1.5)$“ oder „$+(0.5 \cdot random(0.0, 1.0))$“. In diesen Beispielen ist zu erkennen, dass ein unausgewertetes Regelwort Variablen, Funktionen und Konstanten beinhalten kann, die ausgewertet

[2]Seed: engl. Keim, Samen; eine Zahl, mit der der Zahlengenerator initialisiert wird.

werden müssen. Als Grundlage für dieses Kapitel soll die Regel „$S(len) = S(10) > ?I(a, len, res) : res\,! = NN\,\{a = 0; c = 3;\} \rightarrow +(0.5 \cdot c \cdot random(0.0, 1.0))\,F(len \cdot 1.5)$“ dienen. Die Regel wird ausgeführt, wenn kein „nearest neighbour“ gefunden worden ist. Sie rotiert um einen zufällig beeinflussten Wert und zeichnet ein Segment mit anderthalbfacher Länge des Eingabeparameters.

Im ersten Schritt werden Variablen, die den Aufrufparametern entsprechen, mit deren Belegung substituiert. Es wird für diese Aufgabe eine Map[3] erstellt, die die Variablen und Werte dieser beinhaltet. Diese Map wird in die Regelworte eingearbeitet, so dass im obigen Beispiel daraus „$+(0.5 \cdot c \cdot random(0.0, 1.0))\,F(10 \cdot 1.5)$“ folgt.

Im nächsten Schritt werden Funktionen, wie *random(min,max)* oder *max(a,b)*, ausgewertet und in den Regelworten ersetzt. Ein mögliches Zwischenergebnis wäre „$+(0.5 \cdot c \cdot 0.312)\,F(10 \cdot 1.5)$“.

Im folgenden Schritt werden optionale Aufrufparameter ersetzt. Diese Parameter sind nicht die Parameter des Regelkopfes, sondern die Rückgabeparameter und Zuweisungen einer selbstsensitiven Bedingung. Über diese Zuweisungen können neue Variablen definiert und initialisiert werden, ohne dass diese im Regelkopf bekannt sind. Dies führt zur Ersetzung von c durch „3“ und dem Zwischenergebnis „$+(0.5 \cdot 3 \cdot 0.312)\,F(10 \cdot 1.5)$“.

Im vorletzten Schritt muss nun der mathematische Ausdruck berechnet werden. Bei der Berechnung werden die Grundrechenarten plus, minus, mal und geteilt unterstützt, sowie die korrekte Punkt-vor-Strich-Rechnung und die Klammerung von Ausdrücken. Für die Auswertung wird der Ausdruck zuerst in die Umgekehrte Polnische Notation (UPN) überführt. Bei dieser Notation werden immer zuerst die Operanden und dann der Operator angegeben. Implementiert ist dieser Rechner mit zwei Stacks, die für einen rekursiv absteigenden Parser benötigt werden. Einer für die Operatoren und die Klammerung und einer für die Operanden. Hieraus folgt das Zwischenergebnis „$+(0.468)\,F(15)$“.

Im letzten Schritt werden die Worte in Funktionen konvertiert. Der Unterschied besteht lediglich im Datentyp der Parameter, die nun Fließkommazahlen sind. Als Ergebnis liefert die Regel nun eine Liste von ausgewerteten L-Systemwörter, die wiederum im nächsten Erweiterungschritt einzeln ausgewertet werden können.

4.2.12 Verarbeitung der Ersetzungen

Konnte das L-Systemwort nicht ersetzt werden, wird dieses belassen und nicht weiter behandelt. Hat eine Ersetzung stattgefunden wird das L-Systemwort mit der Ersetzung überschrieben. Da die Ersetzung aus mehreren Worten bestehen kann, wird jedes einzeln verarbeitet und im Graphen angehängt. Wie das Einhängen stattfindet, ist davon abhängig, ob das Wort normal eingefügt werden kann oder ob es zu einer Kollision gekommen ist. Die Graphenoperationen werden im Detail in Abschnitt 4.2.13 auf der nächsten Seite beschrieben.

Verarbeitung eines normalen Knotens Wird weder ein Schnitt verursacht noch ein nahegelegener Nachbar gefunden, so wird der neue Knoten eingefügt und mit dem Aus-

[3] Map: Liste bestehend aus Schlüsseln und Werten.

gangsknoten bzw. dem vorherigen verbunden.

Verarbeitung eines nahegelegenen Knotens Wird wie in Abschnitt 4.2.9 auf Seite 50 beschrieben ein „nearest neighbour“ (n_{NN}) erkannt, kann dieser verkettet werden. Es ist zu beachten, dass die Verbindung direkt zwischen dem Ausgangsknoten n_a und n_{NN} erfolgen muss. Die Erstellung eines Zielknotens n_b von n_a aus, der anschließend mit n_{NN} verbunden wird, ist nicht nötig, da $n_b$und n_{NN} identisch wären und somit an gleicher Stelle liegen würden. Zudem würde eine Kante ohne Länge zwischen n_b und n_{NN} erstellt werden. Der Teilgraph, der ohne NN nach n_{NN} erstellt werden würde, wird verworfen.

Verarbeitung eines Schnittknotens Der in Abschnitt 4.2.9 auf Seite 51 errechnete Schnittpunkt kann als Knoten mittels Einfügung, wie in Abschnitt 4.2.13 auf Seite 58 beschrieben, integriert werden. Dieser neue Knoten kann anschließend mit dem Ausgangsknoten verbunden werden (wie in Abschnitt 4.2.12). Der Teilgraph, der über den Schnittpunkt hinaus verlaufen würde, wird verworfen.

Umlenkung von Schnitten Ein Spezialfall bei der Schnitterkennung sind Schnittpunkte, die sehr dicht am Anfang oder Ende einer geschnittenen Kante liegen. In diesem Fall wird die Kante nicht unterdrückt, sondern auf den Knoten umgelenkt, in dessen Nähe sich der Schnittpunkt befindet. Hierbei müssen wieder Längen- und Winkelkriterien eingehalten werden. Zu beachten ist hierbei die Kantenlänge nach der Umlenkung, da diese zu gering werden könnte (siehe Abbildung 4.2.7), falls der Startknoten des vorgeschlagenen Segments zu dicht am Knoten liegt, zu dem der Startknoten verbunden werden soll.

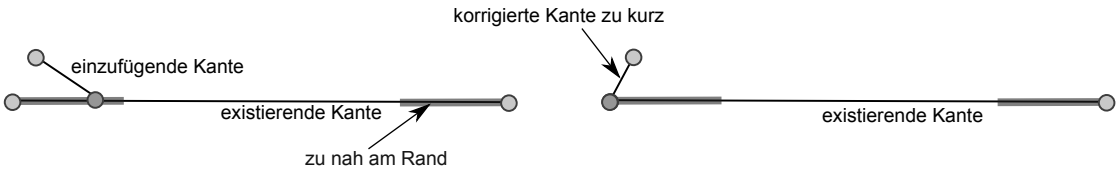


Abb. 4.2.7: Startknoten zu nah an Kantenknoten

4.2.13 Manipulation des Graphen

In einem Graph wird zu jeder Zeit der Zustand des L-Systems gespeichert. Dieser enthält zusätzlich das geometrische Aussehen des Straßennetzes. Hierfür ist es nötig, dass dieser in Echtzeit manipuliert werden kann, ohne dass die geometrischen Bedingungen verletzt werden.

Intern wird der Graph von Knoten repräsentiert, die jeweils einem Wort des L-Systems entsprechen. Diese Knoten sind untereinander wie eine Liste verkettet, so dass damit der Zustand des L-Systems ausgedrückt wird. Die Verschachtelungsparameter („[“ und „]“) des L-Systems führen zu Abzweigungen im Graph und somit zu einer Baumstruktur des

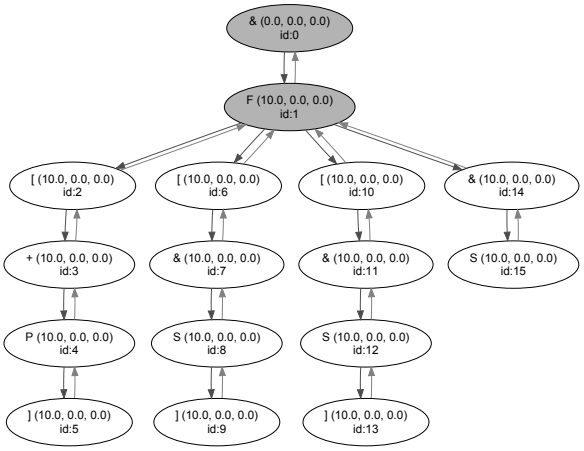


Abb. 4.2.8: Initialer Graph mit drei Platzhaltern

Graphen. Zyklen entstehen schließlich durch die regelbasierte Verbindung von bestehenden Knoten mit neuen Knoten.

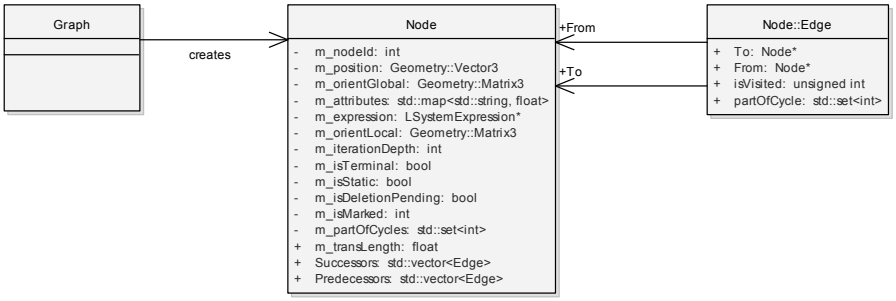


Abb. 4.2.9: Attribute und Abhängigkeiten eines Knotens

Erstellung eines Knotens Wird ein neuer Knoten erstellt, wird eine fortlaufende Identifikationsnummer (ID) erstellt, die der neue Knoten zwecks Optimierung (4.6) erhält. Bei der Erstellung sind einige Pflichtangaben nötig, wie z. B. Vaterknoten, Befehlsname des L-Systems, lokale Orientierung und Transformationslänge. Zusätzlich kann ein Nachfolgerknoten angegeben werden.[4]

Anfügen von Knoten Während der eigentlichen Integration in den Graphen mittels doppelter Verkettung mit Vater- (Predecessor) und Nachfolgerknoten (Successor), insofern vorhanden, wird eine Berechnung der neuen globalen Position ausgelöst.

[4] Abzweigungen kommen durch mehrmaliges Verketten mit einem NN zustande.

Zusätzlich ist bei Knoten, die einen Zeichenbefehl (F) des L-Systems enthalten, eine Auswertung der dadurch entstehenden geometrischen Kante auszuführen. Hierbei muss von diesem neuen Knoten aus der letzte vorherige Zeichenknoten im Graphen ermittelt werden. Dieser Pfad wird zur Beschleunigung mit den Daten der Geometriekante gespeichert. Über dessen globale Position und die dazwischen liegenden Transformationen kann anschließend die globale Position (P) und abgeleitete Orientierung (O) des eingefügten Zeichenknotens bestimmt werden.

$$\begin{aligned} O^{3\times3}_{Global} = \begin{pmatrix} H & L & U \end{pmatrix} &= O^{3\times3}_{Parent\,Global} \cdot O^{3\times3}_{Local} \\ P &= P_{Parent} + H \cdot length \end{aligned}$$

Da die Bewegung durch das L-System immer entlang der H-Achse der Orientierungsmatrix stattfindet, kann über diese die neue Position mittels Verschiebung entlang dieser Achse bestimmt werden. Konnte ein Knoten zwischen zwei anderen Knoten eingefügt werden, wird eine Zyklenerkennung (4.3) ausgelöst.

Aushängen von Knoten Für einige Graphenoperationen ist es nötig, Knoten auszuhängen und dabei die Verkettungen zu korrigieren. Hierzu werden Vor- und Nachfolgerknoten (n_i^Pund n_i^S) des auszuhängenden Knotens (n_{unhook}) gespeichert. Verbindungen zum Knoten n_{unhook} werden aufgelöst. An dieser Stelle ist zu beachten, dass das Aushängen des Wurzelsknotens zu einer Neuwahl von diesem führt. Bevorzugt wird dann der erste Nachfolger von n_{unhook} gewählt, alternativ der erste Vorfolger von n_{unhook}.[5] Handelt es sich bei n_{unhook} um einen Zeichenknoten (F) werden alle Zeichenkanten von oder zu diesem Knoten als zu löschen markiert. Abschließend werden alle Knoten aus n_i^Pund n_i^S miteinander verkettet. Im Normalfall wird eine einfache Verkettung von zwei Knoten ausgeführt. Während dieser Verkettung wird für den Fall, dass n_{unhook} ein Zeichenknoten gewesen ist, eine Neuberechnung der neuen Kante(n) zwischen n_i^Pund n_i^S ausgeführt. Zuletzt wird n_{unhook} als zu löschen markiert. Das eigentliche Löschen findet aus speichertechnischen Gründen erst nach Beendigung eines gesamten Evaluationsschritts statt.

Aushängen von Teilbäumen Das Aushängen von Teilbäumen im Graph wird meistens nötig, wenn eine neue Kante an einen NN angefügt wird. Es wird so verhindert, dass sowohl die alten Platzhalter am NN und die des neuen Segmentes erhalten bleiben. Hierbei wird ein Teilgraph verworfen, der sich unterhalb des letzten gültigen Knotens n_{valid} befindet. Um dies umzusetzen werden mittels einer Tiefensuche alle Knoten ermittelt, die sich unterhalb n_{valid} befinden und, wie zuvor beschrieben, jeweils ausgehängt.

Ersetzen von Knoten durch Listen Wird durch das Regelsystem eine Ersetzung eines Knotens ausgelöst, so müssen Vor- und Nachfolger des zu ersetzenden Knotens gespeichert werden. Anschließend wird der zu ersetzende Knoten ausgehängt und gelöscht. An die/den Vorfolgerknoten (A_n) wird nun der erste Ersetzungsknoten (I_n) angehängt.

[5] Es ist gegeben, dass ein Knoten, der ausgehängt wird, mindestens einen Nachbarknoten besitzt.

Weitere Ersetzungsknoten werden nun jeweils an den vorherigen Ersetzungsknoten angehängt. Dem letzten Ersetzungsknoten werden die zuvor gespeicherten Nachfolgerknoten zugewiesen. Zu beachten ist die korrekte Neuverkettung der ehemaligen Vor- und Nachfolger.

Eine Ausnahme während der Einfügung bildet das Auftreten eines selbstsensitiven Schnittereignisses. Dies hat zur Folge, dass nach dem Knoten, der dieses Ereignis ausgelöst hat, zum Einen keine weiteren Ersetzungen angefügt werden dürfen und zum Anderen alle ehemaligen Nachfolgerknoten erneut evaluiert werden müssen, da deren Position in der Regel durch eingefügte Transformationen nicht mehr korrekt sein wird.

Einfügen von Knoten Beim Auftreten eines selbstsensitiven Schnitts ist es nötig, an dieser Stelle einen neuen Knoten einzufügen. Die Schwierigkeit hierbei besteht in der Erhaltung des geometrischen Zusammenhangs. Falls die Kanten, auf der ein Knoten eingefügt wird, zu einem oder mehreren Zyklen gehört, muss diese Zugehörigkeit gespeichert werden. Denn die Kante wird entfernt und neu erstellt. Da eine Geometriekante immer durch Transformationen erzeugt wird, müssen die entsprechenden Knoten aus dem Graph entfernt werden, so dass nur noch Kantenanfangsknoten (n_{start}) und -endknoten (n_{end}) vorhanden sind. Anschließend müssen die neuen Segmentlängen zum Schnittpunkt errechnet werden und für die Schnittstelle ein Knoten n_{inter} erzeugt werden. Dieser Knoten wird mit korrekter Verschiebungslänge an n_{start} angehängt und bildet die erste Teilkante (e_a). Falls nötig wird e_a den entsprechenden Zyklen zugeordnet. Abschließend muss noch eine Kante (e_b) zwischen n_{inter} und n_{end} eingefügt werden. Hierfür muss die Transformationslänge von n_{end} manipuliert werden. Die Kante e_b wird, falls nötig, wieder den entsprechenden Zyklen zugeordnet.

Die letzte nötige Aktualisierung besteht im Entfernen der alten Kante aus der Lookuptabelle und dem Einfügen der zwei neuen Kanten in diese. Durch entsprechende Funktionsaufrufe kann dem Frontend somit automatisch die Veränderung am Graph mitgeteilt werden.

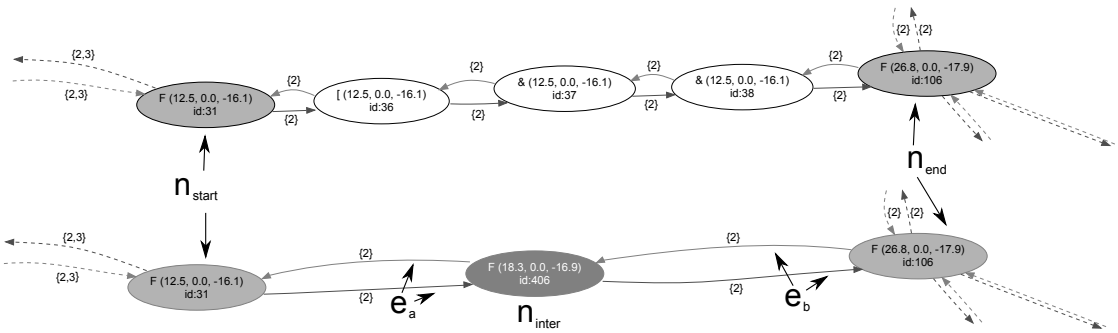


Abb. 4.2.10: Einfügen eines Geometrieknotens

Die hellgraue Kontur der Knoten zeigt an, dass diese nun statisch sind. Dies verhindert eine erneute Evaluation der Transformation beim Anfügen, was durch die nun fehlenden Transformationsknoten zu fehlerhaften Ergebnissen führen würde.

4.3 Verfahren zur Zyklensuche

Um Bauflächen zu bestimmen, müssen die Zyklen des Graphen in diese überführt werden. Dazu gibt es einige Verfahren der Zyklensuche. Ein sehr effektives Verfahren bietet Tarjans Algorithmus für starke Zusammenhangskomponenten[6] (Tarjan/Aspvall/Plass, 1979; Wikipedia, 2010c; Wikipedia, 2010d). Die Strongly Connected Components (SCC) eines gerichteten Graphen sind jene Teilgraphen c_i eines Graphen, dessen Vertices sich untereinander alle erreichen können, ohne rückwärts zu laufen.

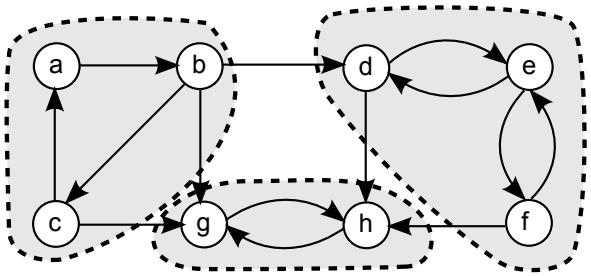


Abb. 4.3.1: SCCs in einem Graphen

Der folgende Pseudocode-Algorithmus aus Wikipedia (2010d) erfüllt diese Aufgabe.

Listing 4.4: Tarjanalgorithmus

```
1  Eingabe: Graph G = (V, E)
2
3  max_dfs := 0                        // Zähler für dfs
4  U := V                              // Menge der unbesuchten Knoten
5  S := {}                             // Stack zu Beginn leer
6  while (es gibt ein v0 in U) do      // Solange es bis jetzt unerreichbare Knoten gibt
7    tarjan(v0)                        // Aufruf arbeitet alle von v0 erreichbaren Knoten ab
8  end while
9
10 procedure tarjan(v)
11 v.dfs := max_dfs;           // Tiefensuchindex setzen
12 v.lowlink := max_dfs;       // v.lowlink <= v.dfs
13 max_dfs := max_dfs + 1;     // Zähler erhöhen
14 S.push(v);                  // v auf Stack setzen
15 U := U \ {v};               // v aus U entfernen
16 for all (v, w) in E do      // benachbarte Knoten betrachten
17   if (w in U)
18     tarjan(w);              // rekursiver Aufruf
19     v.lowlink := min(v.lowlink, w.lowlink);
20   else if (w in S)          // Abfragen, ob w im Stack ist.
21     v.lowlink := min(v.lowlink, w.dfs);
22   end if
23 end for
24 if (v.lowlink = v.dfs)      // Wurzel einer starken Zusammenhangskomponente (SCC)
25   print "Strongly Connected Component:";
26   repeat
27     w := S.pop;
28     print w;
29   until (w = v);
30 end if
```

Dieser und artverwandte Algorithmen arbeiten jedoch nur auf gerichteten Graphen. In einem ungerichteten Graphen ist die Suche von SCCs nicht anwendbar, da hierdurch nicht die geometrisch gewünschten Zyklen gefunden werden würden.

[6] Tarjans strongly connected component algorithm

Eine weitere Möglichkeit bietet die Erstellung von Minimalgerüsten (Minimal Spanning Trees) zur Ermittlung von Zyklen, wie es beim Traveling Salesman Problem (Kazemi et al., 2007) häufig Anwendung findet. Das Problem bei diesem Vorgehen ist, dass alle Zyklen gefunden werden und nicht nur die elementaren Minimalzyklen.

Weitere Algorithmen, die sich mit der Suche von elementaren Zyklen in einem Graph beschäftigen, arbeiten stets auf einem vollendeten Graphen, was den Nachteil besitzt, dass die Suche recht lange dauern kann und somit nicht echtzeitfähig wäre. Hierzu finden aktuell einige Forschungen statt, die sich u.a. mit der Laufzeitreduzierung beschäftigen. (Tiernan, 1970; Wulff-Nilsen, 2009)

Nachdem keine der bekannten Lösungen ansatzweise echtzeitfähig wären und eigentlich nur neue Zyklen erkannt werden müssen, ist ein anderer Ansatz nötig. Es ist festzustellen, dass beim Einfügen einer neuen Kante im Graphen maximal zwei Zyklen entstehen können und zwar links und rechts der neuen Kante. Wird bedacht, dass ein bestehender Zyklus niemals unterteilt wird, kann maximal ein Zyklus entstehen.

4.3.1 Eigenes Verfahren

Unter Berücksichtigung der zuvor genannten Eigenschaft kann von der eingefügten Kante aus der kürzeste Zyklus gesucht werden, hierfür wurde ein Algorithmus entwickelt.

1. Markiere die zyklenschließende Kante als unpassierbar.
2. Speichere Kantenstart und -ende für den Beginn zweier Breitensuchen (BFS) in eine Liste.
3. Wähle die kürzeste Breitensuche und entferne diese aus der Liste.
4. Markiere den vordersten Knoten der aktiven BFS als besucht.
5. Füge alle möglichen Folgeknoten jeweils an den bestehenden Weg an und speichere die daraus resultierenden neuen Wege.
6. Wird ein Knoten angefügt, der bereits als besucht markiert wurde, wurde ein Zyklus gefunden.
   a) Abbruch der Suche
   b) Durchsuche die Liste nach zwei BFSs, die verschiedene Startknoten und identische Endknoten haben.
   c) Kombiniere die Wege dieser zwei BFSs zu einem Zyklus
   d) Entfernung aller gesetzten Markierungen
7. Weiter bei 3.

Dieses Vorgehen kann jedoch nicht erkennen, ob der gefundene Zyklus bereits erkannte Zyklen beinhaltet oder nicht. Dies ist der Fall, wenn der neue Zyklus länger ist als ein Pfad entlang eines bestehenden Zyklus. Außerdem wird nicht erkannt, ob der gefundene Zyklus terminale Segmente (Sackgassen) oder kleinere Zyklen enthält.

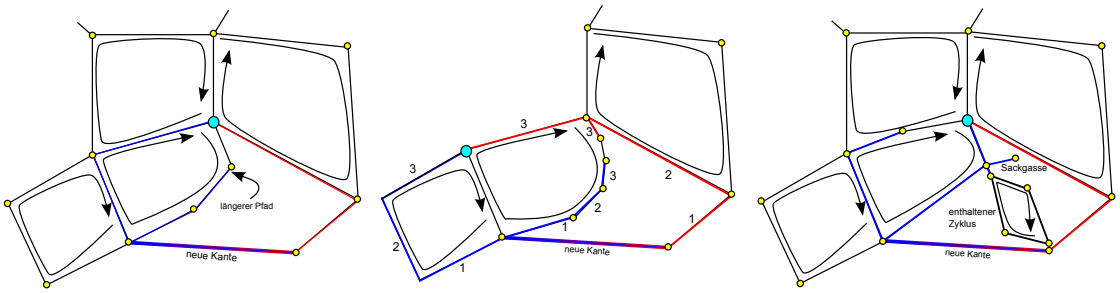


Abb. 4.3.2: Sonderfälle der Zyklensuche mit einfacher BFS

Um den ersten Fall (Abbildung 4.3.2) zu beheben, ist es nötig, für jede Kante zu speichern, welchen Zyklen sie bereits angehört. Somit kann bei der Suche verhindert werden, dass Kanten passiert werden, die bereits zwei Zyklen angehören. Der letztere Fall, in dem sich Sackgassen in einem Zyklus befinden, lässt sich nur durch eine weitere Behandlung lösen. Hierfür müssen alle unmittelbaren Nachbarknoten des gefundenen Zyklus darauf geprüft werden, ob sie geometrisch innerhalb oder außerhalb des neu geschlossenen Zyklus liegen. Ist dies geschehen, können die Sackgassen nun entfernt oder in den Zyklus eingearbeitet werden. Für den weiteren Verlauf der Arbeit hat man sich dazu entschieden Sackgassen zu entfernen (4.2.13). Durch diese Erweiterungen ist jedoch weiterhin nicht der Fall des innenliegenden Zyklus gelöst. Auch der mittlere Fall aus Abbildung 4.3.2, bei dem der kürzere Zyklus über die Außenkanten von Zyklen geschlossen wird, wird so noch nicht gelöst. Hier wird zwar keine Kante mehr passiert, die zwei Zyklen angehört, jedoch ist der Weg dennoch nicht korrekt.

Die Lösung der bestehenden Probleme lässt sich durch einen stark angepassten Algorithmus erreichen.

4.3.2 Optimierter Algorithmus

Die Grundidee dieser Zyklensuche besteht darin, dass jeweils die äußeren Pfade verfolgt werden. Hierfür muss jedoch an jeder Abzweigung die geometrische Ausrichtung der Kanten (e_i), die durch die benachbarten Knoten aufgespannt werden, bekannt sein. Diese Ausrichtungen müssen in Abhängigkeit der Herkunftskante (s) sortiert werden. Bei der Berechnung der Winkel mittels *atan2(y,x)* entsteht nach einer Normierung ein Winkel zwischen $-\pi$ und π. Bei der Verwendung von *atan2* wird im Gegensatz zum Arkustangens, der nur zwischen $\frac{-\pi}{2}$ und $\frac{\pi}{2}$ definiert ist, ein Winkel bestimmt, der eindeutig den Quadranten des Koordinatensystems angibt. Da ein Winkel zwischen 0 und 2π in Abhängigkeit des Herkunftsvektors benötigt wird, kann *atan2* verwendet werden.

$$\begin{aligned} \theta_0 &= atan2(\|s\|_y, \|s\|_x) \\ \theta_{1,i} &= atan2(\|e_i\|_y, \|e_i\|_x) \\ \omega_i &= \theta_{1,i} - \theta_0 \end{aligned}$$

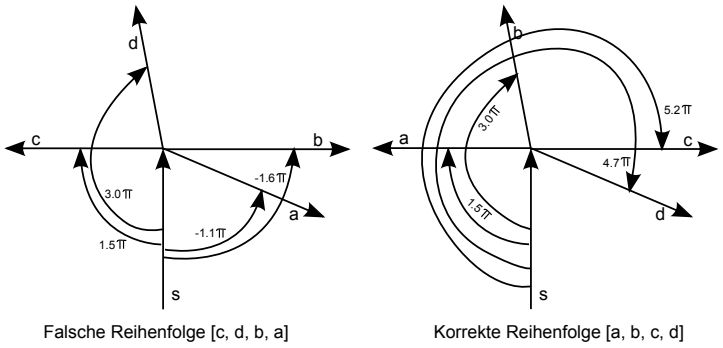


Abb. 4.3.3: Korrektes Sortieren der Nachbarkanten

Würden die negativen Winkel nicht wie im Beispiel aus Abbildung 4.3.3 um $+2\pi$ korrigiert werden, würden die beiden rechten Kanten fälschlicherweise vertauscht werden. Mit der Möglichkeit, die Folgekanten nach Winkeln zu sortieren, kann der Algorithmus unter Einhaltung der Kantenbedingung aus Abschnitt 4.3 auf Seite 59 wie folgt arbeiten.

1. Speichere den Kantenstart in zwei BFSs, $pathLeft$ und $pathRight$.
2. Wähle die BFS aus $pathLeft$ und $pathRight$ aus, die den kürzeren Pfad besitzt.
3. Markiere den vordersten Knoten x der aktiven BFS als von links bzw. rechts besucht.
4. Ermittle für x eine nach Winkeln sortierte Nachbarkantenliste.
5. Enthält diese Liste mindestens einen Pfad und x wurde nicht von links und rechts besucht?

   ja Erweitere den aktiven Pfad um diese Liste und markiere den äußersten Knoten entsprechend der aktiven Liste als aktiv

   a) Enthält die angefügte Liste den Endknoten der Startkante ist der innere Zyklus gefunden worden.

      i. Beenden der Suche

      ii. Der Zyklus kann über die aktiven Knoten des Suchpfads ermittelt werden.

      iii. Entfernung aller gesetzten Markierungen.

   nein Es wurde eine Sackgasse gefunden oder x wurde von beiden Seiten besucht, dies bedeutet, dass ein Pfad über einen bestehenden Zyklus gegangen wurde.

   a) Suche eine Alternative mittels Backtracking

      i. Entferne solange die obersten Pfadlisten, bis eine andere Liste mit mindestens zwei Wegen gefunden worden ist.

ii. Kann keine Liste mit Alternativen gefunden werden, so befindet sich der Zyklus in der anderen Suchrichtung.

iii. Verschiebe den aktiven Pfad der nun obersten Pfadliste um eins weiter vom Rand weg. ($pathLeft$ verschiebt nach rechts und umgekehrt)

6. Weiter bei 2.

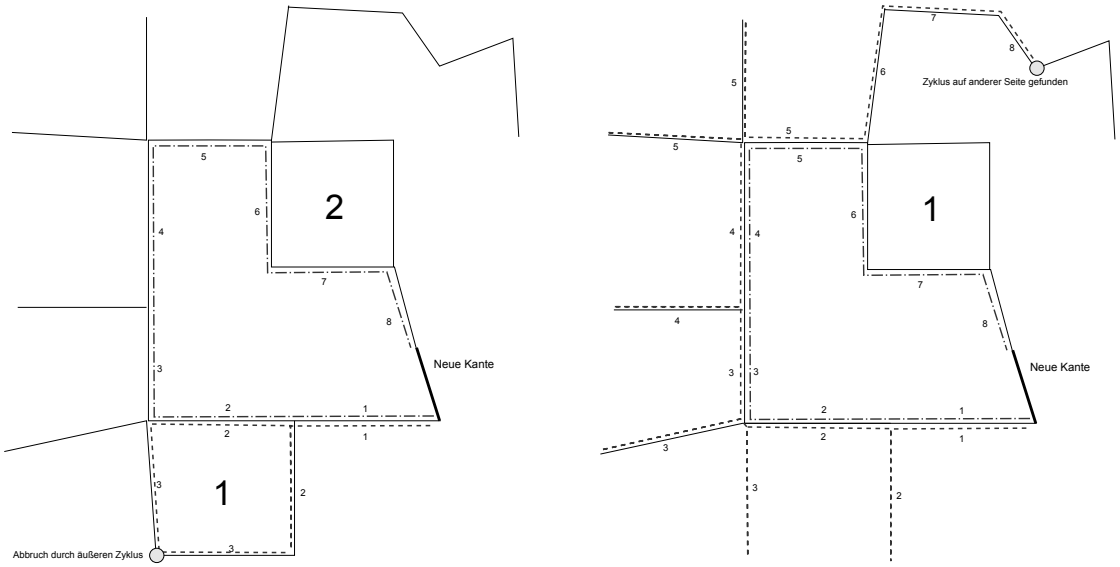


Abb. 4.3.4: Arbeitsweise der Zyklenerkennung

Dieser Algorithmus bietet eine Reihe Vorteile. Zum Einen findet er als Nebenprodukt automatisch innen- und außenliegende terminale Segmente, die für die spätere Nachbearbeitung wichtig sind. Zum Anderen ist es nicht mehr nötig, aus zwei Zyklenhälften auf den Zyklus zu schließen, da der vollständige Pfad gefunden wird. Ein weiterer Vorteil liegt in der korrekten Erkennung des Zyklus selbst bei eingeschlossenen Zyklen wie in Abbildung 4.3.2 auf Seite 61.

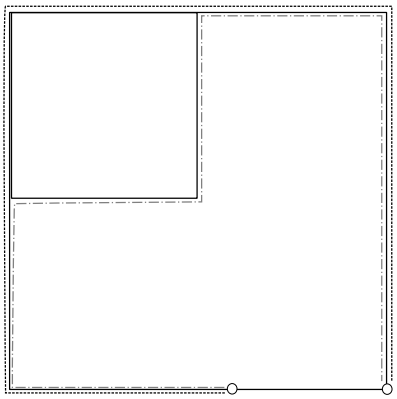


Abb. 4.3.5: Spezialfall eines gleichlangen Zyklus

Ein weiterer Sonderfall bleibt jedoch bestehen. Dieser tritt ein, wenn ein außenliegender Zyklus die gleiche Knotenzahl besitzt wie der innenliegende Zyklus und dabei keinerlei Abzweigungen existieren, die einen Abbruch verursachen könnten, wie in Abbildung 4.3.2 auf Seite 61. Hierbei ist die Wahl des korrekten Ergebnisses von der Ausführungsreihenfolge abhängig. Eine Lösung des Problems mittels Vergleich über die Pfadlänge ist nicht möglich, da sich diese in dem Beispiel nicht unterscheidet. Den korrekten vom falschen Fall zu unterscheiden ist hier

nur mittels Berechnung der Flächeninhalte der Zyklen möglich. Dies setzt jedoch voraus, dass beide Zyklenpfade bis zum Ende verfolgt werden, was jedoch auf Kosten der Laufzeit geht. Eine weitere Möglichkeit besteht in der Ermittlung des korrekten Pfads über die Ausrichtung der eingefügten Kante. Dies wird jedoch problematisch bei der Suche von dreidimensionalen Zyklen.

4.3.3 Zyklenerkennung im 3D

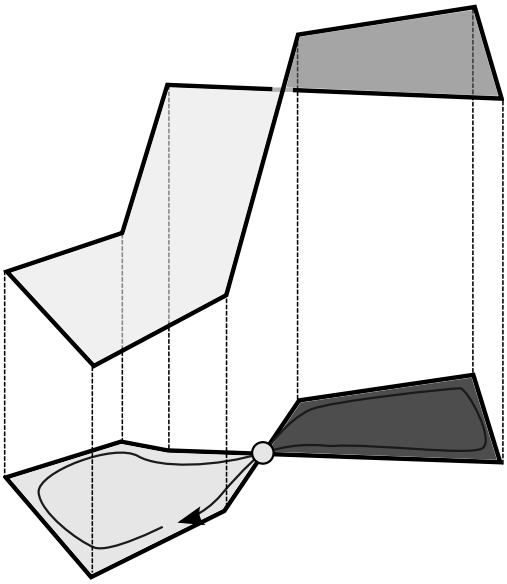


Abb. 4.3.6: Pseudozyklus im 3D

Bei der Suche von Zyklen in einem dreidimensionalen Graph gibt es einige Eigenarten zu beachten und zu behandeln (Sun/Chen, 2000). Zyklen können sich durch senkrechte Kanten in ihrer Projektion überschneiden. Dies ist problematisch beim Erstellen von Parzellen. Zudem können senkrechte Kanten Zyklen auf verschiedener Höhe verbinden, wodurch es zu Überschneidungen von Objekten im unteren Zyklus kommen kann. Die in der Abbildung 4.3.6 dargestellte Problematik kann dadurch gelöst werden, dass keine Parzellen in nichtplanaren Zyklen erstellt werden. Wird ein solcher nichtplanarer Zyklus gefunden, wird dieser gespeichert aber in Zukunft nicht weiter behandelt. Senkrechte Kanten, die planare Zyklen in verschiedener Höhe verbinden, lassen sich durch einen projizierten Schnittest erkennen und wie im ersten Fall behandeln.

4.4 Erzeugung von Geometrie

Das Backend stellt an einigen Stellen die Möglichkeit zur Verfügung, aus Geometriebeschreibungen eine Beispielgeometrie zu erzeugen. Dies ist für Hochhäuser und Straßenkreuzungen möglich.

4.4.1 Erstellung von Gebäuden

Da das Backend lediglich die Beschreibung von Bauflächen übernimmt, kann das Frontend selbst entscheiden, ob es auf die Flächen vormodellierte Gebäude stellt oder anderweitig Gebäude auf den Bauflächen erstellt.

Für die beispielhafte Gebäudeerstellung liefert das Backend eine einfache Extrusion der Gebäudekontur. Intern wird hierzu das Extrusionsverfahren (siehe 3.3.3) verwendet, das

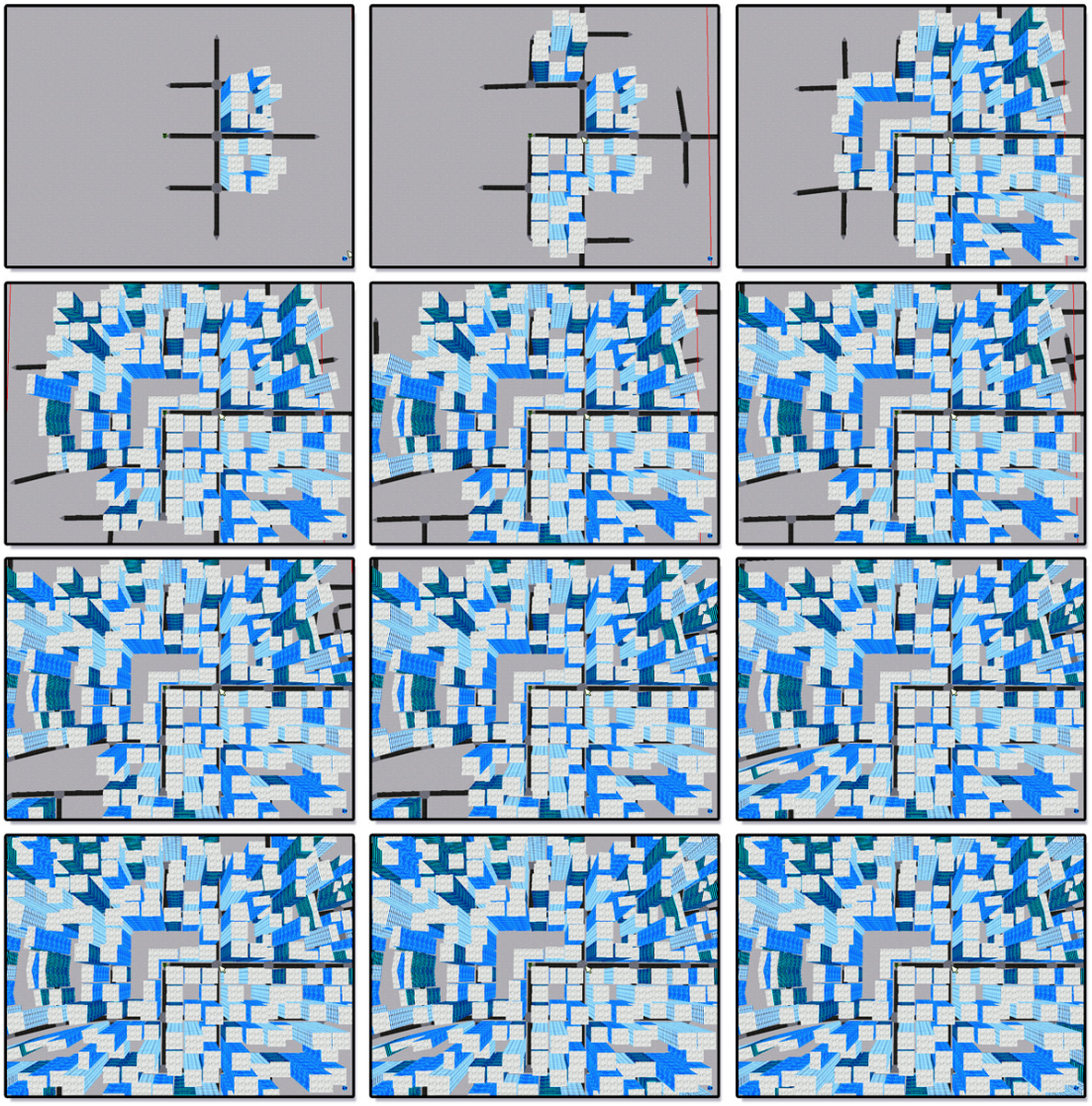


Abb. 4.4.1: Zwischenergebnisse mehrerer Iterationsschritte

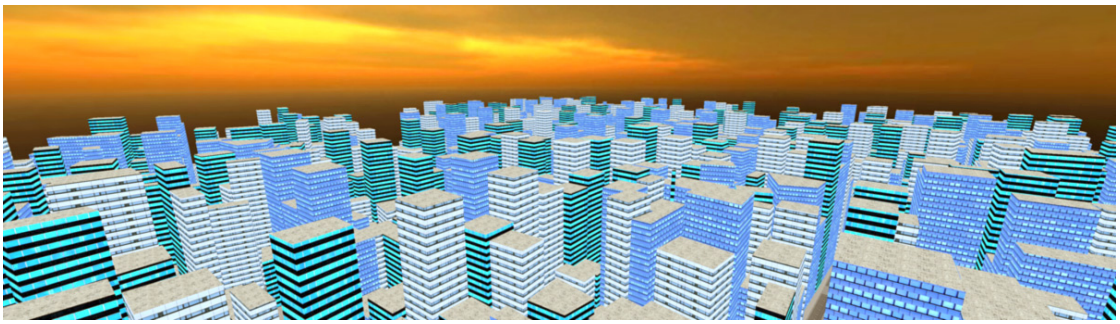


Abb. 4.4.2: Vogelperspektive einer generierten Stadt

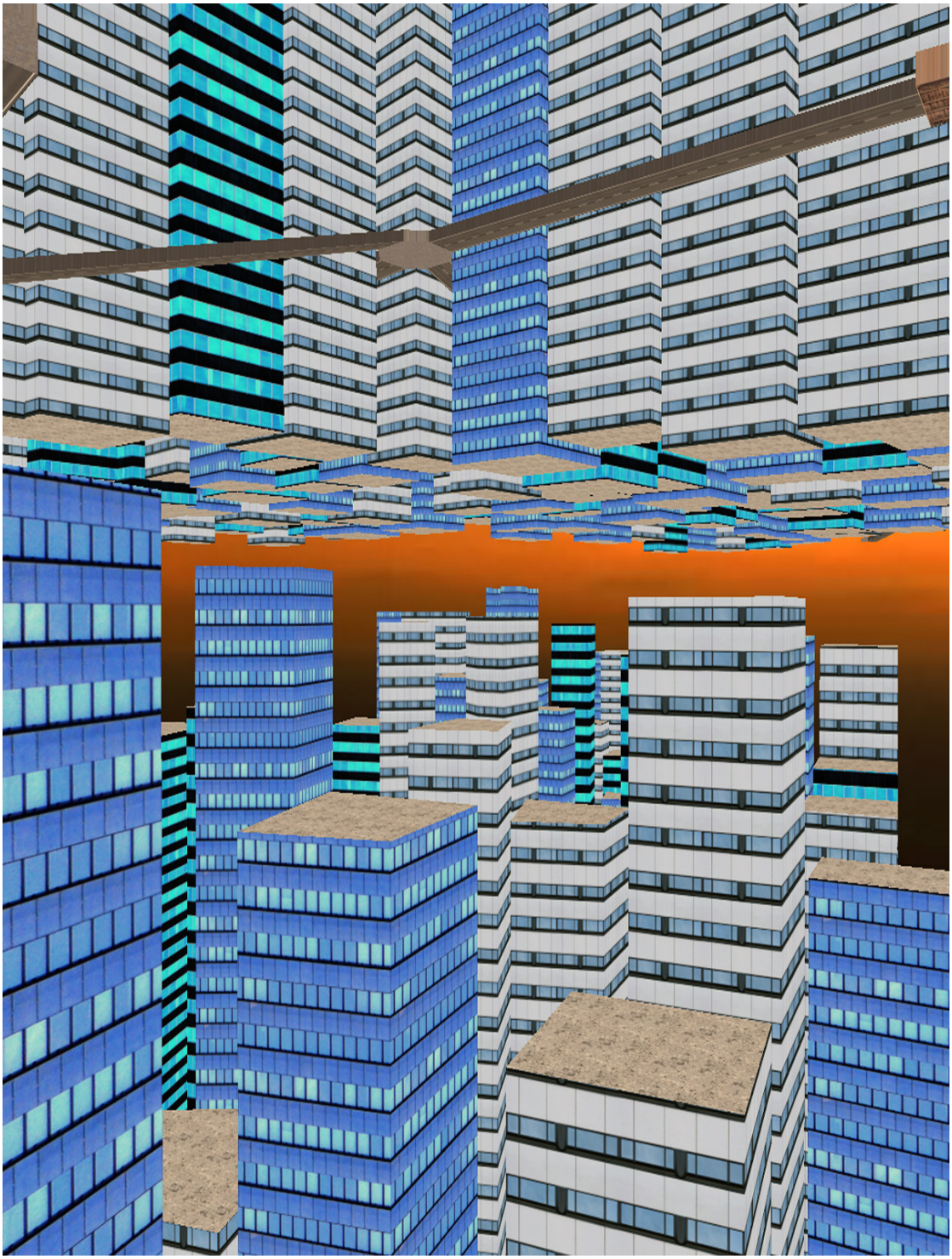


Abb. 4.4.3: 3D-Stadt mit zwei Ebenen

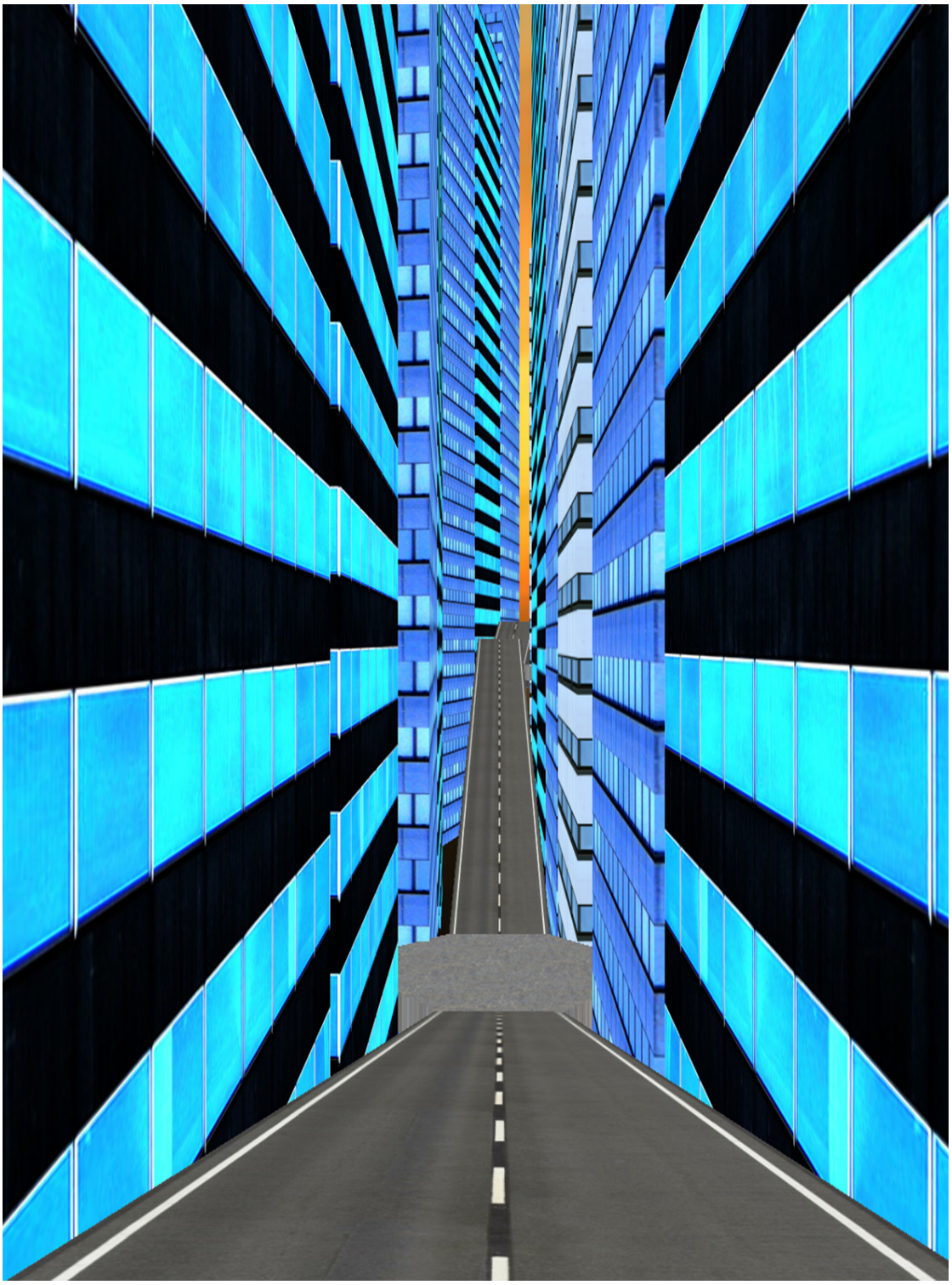


Abb. 4.4.4: Bewegung durch eine 3D-Stadt

das Gebäude von oben nach unten erstellt. Es wird jedoch aus in Abschnitt 6.2 auf Seite 90 erläutertem Grund bereits nach der ersten Iteration abgebrochen. Als Gebäudehöhe wird ein zufälliger Wert aus einem spezifizierbaren Intervall gewählt. Das Gebäudedach wird durch eine einfache Triangulation (siehe Constrained Delaunay 2.2.2) erstellt. Bei schwebenden Gebäuden wird das gleiche Vorgehen für den Boden angewendet.

4.4.2 Erstellung von Straßenkreuzungen

Die Komplexität bei der Erstellung von Kreuzungen besteht darin, dass erst zur Laufzeit bekannt ist, wie viele Straßen sich an einer Kreuzung in welchem Winkel treffen. Hierzu wurde ein einfacher Weg geschaffen, um aus diesen Informationen Kreuzungen zu erzeugen (Abbildung 4.4.5). Als Eingabedaten werden der Kreuzungsmittelpunkt und die Vektoren von diesem zu den anschließenden Straßen verwendet. Für jeden Richtungsvektor wird ein orthogonaler Vektor bestimmt, mit dem in Straßenbreite (k) zwei Punkte der Kreuzungskontur erzeugt werden. Aus Kreuzungsmittelpunkt und den Konturpunkten kann eine Triangulation erzeugt werden.

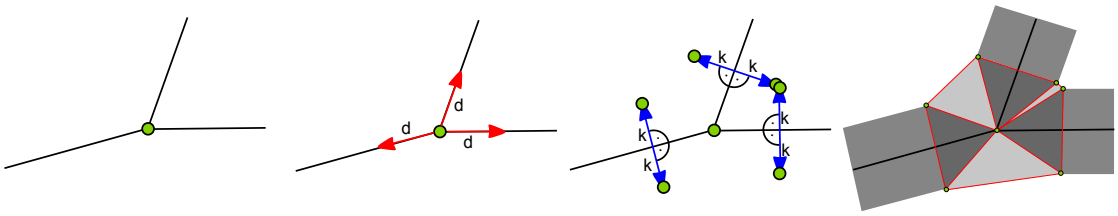


Abb. 4.4.5: Verfahren der Straßenerzeugung

Sollte der Winkel zwischen zwei Straßen zu gering werden, führt dies zu keiner Überlappung, da die korrekte Reihenfolge der Konturpunkte keinen Einfluss auf die Triangulation hat. Als Nebenprodukt der Triangulation entsteht eine korrekte Punktreihenfolge der Kontur (konvexe Hülle). Diese Kontur wird benötigt, um die triangulierte Fläche zu extrudieren. Der Grund hierfür liegt darin, dass Kreuzungen von schwebenden Straßen eine Dicke besitzen, so dass diese in der Welt von allen Seiten sichtbar sind. Beim Extrudieren werden die Vertices der Kreuzung dupliziert und entlang der negativen Flächennormale verschoben. Anschließend wird eine Indexliste für die Dreiecke erstellt, so dass ein geschlossenes Objekt entsteht.

4.5 Navigation

Neben der Erstellung neuer Inhalte bietet das Backend eine einfache Möglichkeit Informationen von Straßensegmenten bereitzustellen. Dies kann vom Frontend für die Navigation der Spielfigur verwendet werden. Das Backend speichert dazu für das Navigieren jeweils das zuletzt aktive Straßenelement zwischen.

Mit der Funktion *getCurrentTrack()* wird die Straßen-ID (z. B. „A17B39“) der aktuell aktiven Kanten zurückgeliefert. Zusätzlich werden Abbiegemöglichkeiten am Ende des Segmentes bestimmt. Diese können mit dem Aufruf *getStreetAngles()* abgerufen werden.

Das Ergebnis dieser Funktion entspricht einer Liste von Winkeln der angeschlossenen Straßen zur Herkunftskante (siehe Abbildung 4.3.3 auf Seite 62). Die Winkel werden um $-\pi$ bzw. $-180°$ verschoben, so dass ein Geradeaussegment den Winkel 0 erhält und eine Straße im 90°-Winkel nach links $\frac{-\pi}{2}$ bzw. -90° erhält. So kann später sehr einfach festgestellt werden, ob der Spieler geradeaus fahren, links oder rechts abbiegen kann.

Um festzustellen um welche Kante es sich bei einem Winkel handelt, kann mit *getNextTrack(_id)* die Straßen-ID (z. B. „A39B77“) ermittelt werden. Der Parameter der Funktion gibt hierbei den Index des Winkels vom Aufruf *getStreetAngles()* an.

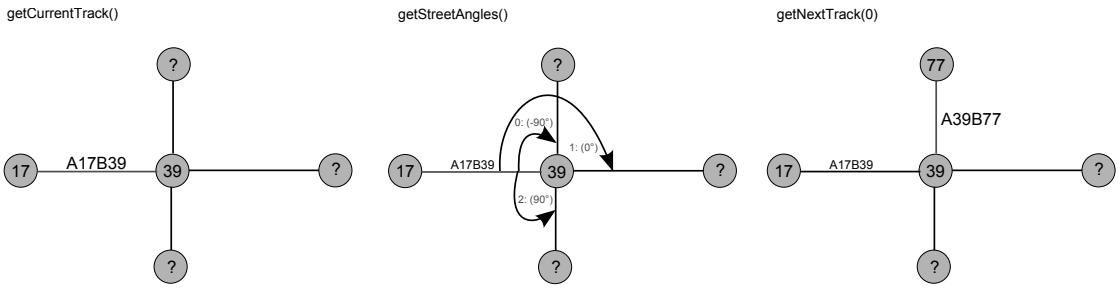


Abb. 4.5.1: Beispiel der Navigation entlang der Kanten

Zusätzlich kann mit der Funktion *setTrackUpVector(_upVector)* eine Standardebene festgelegt werden, in der sich der Spieler befindet. Der Normalenvektor dieser Ebene entspricht dem Upvektor der Spielfigur. Der Hintergrund dieser Funktionalität besteht darin, feststellen zu können, welche anschließenden Straßen planar zur aktiven Straße sind und welche außerhalb der Standardebene liegen. Die Funktionen *getNextTrack* und *getStreetAngles* besitzen aus diesem Grund einen optionalen Parameter, der es ermöglicht, jeweils die Segmente innerhalb und außerhalb der Standardebene abzufragen. Im Frontend kann somit die Navigation entlang waagrechter und senkrechter Straßen voneinander unabhängig behandelt werden.

Um die Navigationsschnittstelle abzurunden, wird mittels *resetTrack()* die Möglichkeit geboten, die Spielfigur auf Anfang bzw. ein optional angebbares Segment zurückzusetzen.

Bei der verwendeten Implementierung werden immer nur die nächst nötigen Daten vorgehalten und somit der Zugriff auf die Graphenstruktur minimiert. Durch die Regelung der Navigation im Backend ist es somit auch möglich, auf Pfade zuzugreifen, die erst noch in der Entstehung sind, was im Normalfall jedoch nicht nötig sein sollte.

4.6 Optimierung

Bei der prozeduralen Generierung gesamter Welten ist sehr schnell festzustellen, dass viele wiederkehrende laufzeitintensive Aufgaben existieren. Optimierungen können diese erheblich beschleunigen.

Abb. 4.5.2: Sicht aus erster Person ohne Shadowmapping

Abb. 4.5.3: Sicht aus erster Person mit Shadowmapping

4.6.1 Lookuptabellen

Zu den einfacheren Techniken gehört die Verwendung von Lookuptabellen. Diese kommen im Backend hauptsächlich für die beschleunigte Ermittlung von Pfaden zwischen Zeichenknoten sowie der Erkennung von Zyklenzusammenhängen zum Einsatz. Zusätzlich wird auch eine Tabelle über im Graph enthaltene Knoten geführt.

Konvertierung eines Zeichenpfads

Der wichtigste Teil stellt die Speicherung der Knotenpfade dar. Jeder Knoten erhält während der Erstellung eine fortlaufende ID. Durch eine Kette von IDs lässt sich ein Pfad zwischen zwei zeichenbaren Knoten (siehe 4.2.13) darstellen. Über diese Speicherung ist es möglich beispielsweise Zyklen ausschließlich über ihre Zeichenknoten zu beschreiben und die dazwischenliegenden Transformationen ohne Breitensuche nachzuschlagen.

In diesem Abschnitt wird in einem Minimalbeispiel die Konvertierung eines aus Zeichenknoten bestehenden Zyklus in einen vollständigen Zyklus beschrieben. In der Graphenrepräsentation aus Abbildung 4.6.1 auf der nächsten Seite ist zuletzt eine Verbindung zwischen Knoten 53 und Knoten 76 eingefügt worden.

Wird der Algorithmus auf den erkannten Zeichenzyklus [46, 76, 16, 1, 46] angewandt und die Daten aus Tabelle 4.6.1 verwendet, kann der vollständige Pfad, wie in Tabelle 4.6.2 auf Seite 73 beschrieben, erkannt werden. Die Pfadliste aus Tabelle 4.6.1 ist durch das Einfügen in den Graphen bekannt und von der kleineren zur größeren ID sortiert.

Zeichenkante	Zeichenpfad
0-1	(0,1)
1-16	(1,6,7,8,16)
1-31	(1,10,11,12,31)
1-46	(1,14,15,46)
16-61	(16,21,22,23,61)
16-76	(16,25,26,27,76)
16-91	(16,29,30,91)
31-106	(31,36,37,38,106)
31-121	(31,40,41,42,121)
31-136	(31,44,45,136)
46-76	(46,51,52,53,76)

Tabelle 4.6.1: Zuordnung der Zeichenkanten zu deren vollständigen Pfaden

Der vollständige Pfad wird ermittelt indem die jeweils aufeinanderfolgenden Knoten nach Größe sortiert in der Pfadliste gesucht werden. Die gefundenen Pfade werden jeweils an das aktuelle Zwischenergebnis angehängt.

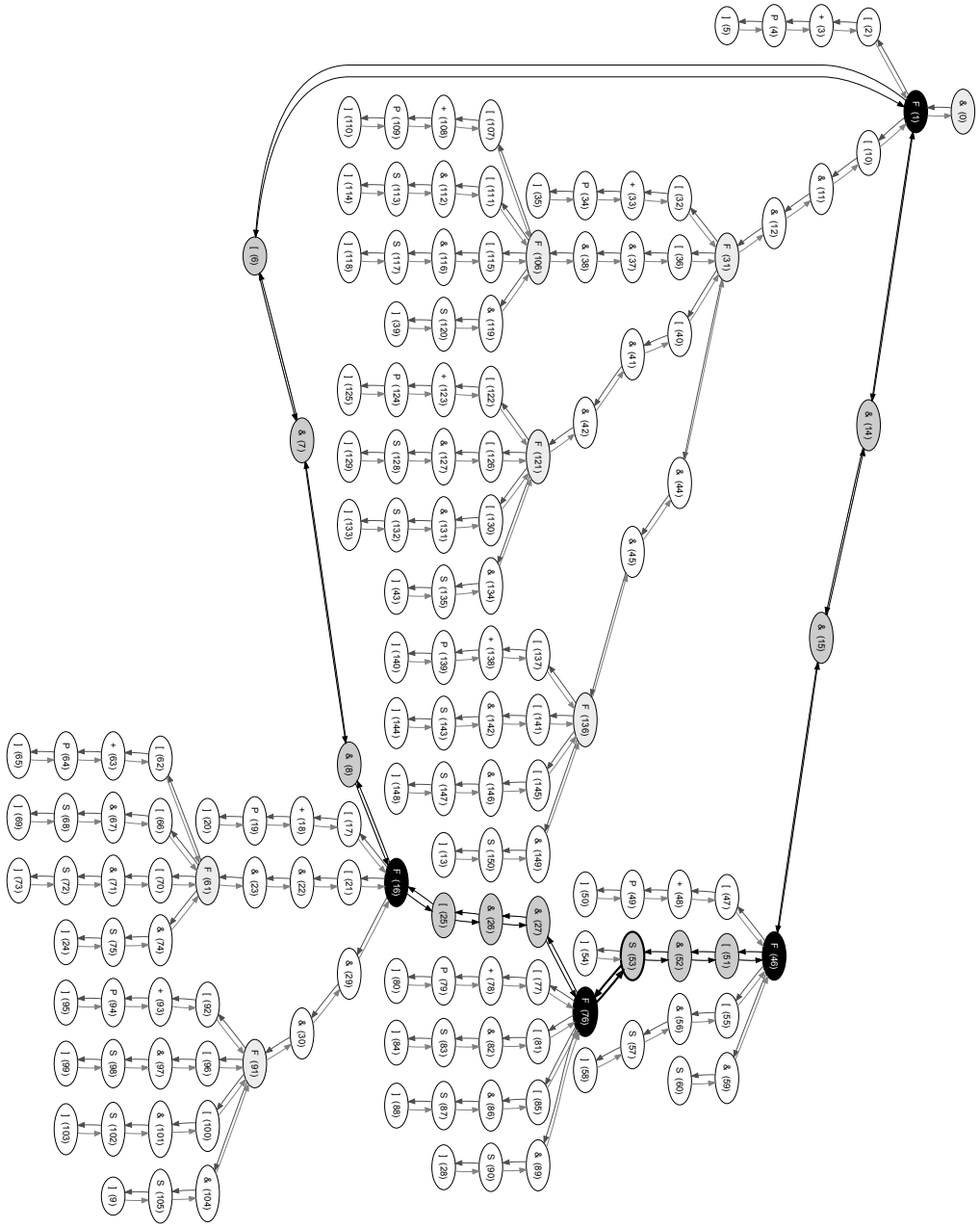


Abb. 4.6.1: Minimalbeispiel eines Graphen mit Zyklus

Eingabe	Suche	Suchpfad	Zielpfad	Ergebnis
46, 76	46, 76	(46,51,52,53,76)	(46,51,52,53,76)	(*46,51,52,53,76*)
76, 16	16, 76	(16,25,26,27,76)	(76,27,26,25,16)	(46,51,52,53,76,*27,26,25,16*)
16, 1	1, 16	(1,6,7,8,16)	(16,8,7,6,1)	(46,51,52,53,76,27,26,25,16,*8,7,6,1*)
1, 46	1, 46	(1,14,15,46)	(1,14,15,46)	(46,51,52,53,76,27,26,25,16,8,7,6,1,*14,15,46*)

Tabelle 4.6.2: Konvertierung eines Zeichenpfads

Räumliche Gliederung

Eine weitere Geschwindigkeitsoptimierung kann erreicht werden, indem die Zeichenknoten räumlich gegliedert werden. Durch geeignete Datenstrukturen wie z.B. Quadtrees bzw Octrees kann eine bessere räumliche Strukturierung der Daten erzielt werden. Dies ist sinnvoll für die Berechnung von Kollisionen mit benachbarten Elementen. Findet keine räumliche Gliederung statt, müssen alle Knoten und Kanten für jeden Kollisionstest betrachtet werden. Wird jedoch eine Grobgliederung in Oktanten vorgenommen, die in einem Octree organisiert werden, müssen nur jeweils die berührten Oktanten und deren Elemente in die Berechnung einbezogen werden.

4.6.2 Caching von Ergebnissen

Der Kollisionstest kann je nach Regelsatz und passenden Regeln mehrmals pro Ersetzungsschritt ausgeführt werden. Es ist festzustellen, dass zumeist die gleichen Kriterien auf nur unterschiedliche Ergebnisse geprüft werden. Eine erneute Berechnung des Schnittergebnisses ist somit überflüssig und kann durch eine Speicherung der letzten Eingabeparameter und dem dazugehörigen Ergebnis umgangen werden. Mit der Wiederverwendung des zwischengespeicherten Ergebnisses ist es möglich, effiziente Fallunterscheidungen durchzuführen, die im Regelsatz nicht ausgedrückt werden können.

4.6.3 Streaming und Caching von Unterschieden

Das Frontend kann vom Backend jederzeit unter Angabe einer Region aus Tiefe und Breite neue Daten in dieser Region anfordern. Das Backend bearbeitet die Anforderung und teilt dessen Ergebnis nach Bearbeitung dem Frontend mit. Das Frontend kann die Änderungen anschließend abfragen und auswerten. Beim Backend kann die Erstellung neuer Straßen (*updateStreetNetwork*) und neuer Parzellen (*updateLots*) angefordert werden. Nach Beendung der Erstellung können über *getStreetNetworkUpdate()* Straßen, über

getStreetCrossingsUpdate() Kreuzungen und über *getLotUpdate()* die Änderungen in Form einer Liste abgerufen werden.

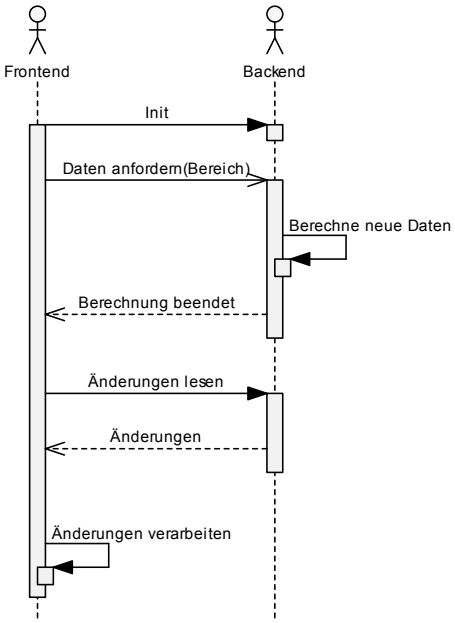


Abb. 4.6.2: Asynchrone Berechnung neuer Daten

Straßen werden durch einfache Kanten repräsentiert, die nur aus Start- und Endposition sowie den Positionen der anschließenden Kreuzungen bestehen. Zusätzlich wird mit einem Flag angegeben, ob eine Straße seit dem letzten Update hinzugefügt oder entfernt worden ist.

Die Straßenkreuzungen werden als fertige Triangulationen (siehe Abschnitt 4.4.2 auf Seite 68) zurückgeliefert und sind somit optional. Die Kreuzungen könnten auch über die Straßeninformationen im Frontend auf andere Weise erzeugt werden. Ist die Kreuzung seit dem letzten Update erstellt, gelöscht oder durch eine neue anschließende Straße verändert worden, wird dies ebenso mit einem Flag mitgeteilt. Es wird dem Frontend überlassen, wie es dies handhabt. Das Neuerstellen einer geänderten Kreuzung ist die einfachste Lösung.

Die Parzellen werden als Liste dreidimensionaler Konturen zurückgeliefert, die wie die Straßenobjekte ein Änderungsflag enthalten. Es wird an dieser Stelle kein Gebäude erstellt. Soll für eine Parzelle ein prozedural erzeugtes Gebäude erstellt werden, kann dies mit der Funktion *getBuilding(_lot)* erfolgen, die als Eingabe die Kontur erhält. Die Verwendung dieser Funktion ist rein optional, da das Frontend über die Kontur selbstständig ein geeignetes vormodelliertes Modell wählen könnte, das anstelle eines prozeduralen Gebäudes verwendet wird.

4.6.4 Multithreading

Multithreading spielt bei der Implementierung von Streaminganwendungen eine große Rolle. Der Vorteil von Multithreading besteht darin, dass arbeitsintensive Aufgaben auf andere Threads ausgelagert werden können. Bei der Verwendung eines Systems mit mehreren CPU-Kernen können somit intensive Rechnungen von einem Kern und die Darstellung der Ergebnisse von einem anderen Kern ausgeführt werden.

Das Gesamtsystem ist so entwickelt worden, dass das Rendern der jeweils aktuellen Stadt in einem Thread ausgeführt wird. Die Berechnung neuer Stadtteile wird in einem anderen Thread ausgeführt. Hiervon sind die *update*-Methoden betroffen, die ihre Aufgabe jeweils auf einen weiteren Thread auslagern. Das Betriebssystem kann diese dann auf die verschiedenen Kerne verteilen. Dies hat den Vorteil, dass der Anwender sich störungsfrei und ohne Unterbrechungen durch die Welt bewegen kann. Ladebildschirme sind durch die Hintergrundberechnung somit überflüssig.

Durch das Threading ist jedoch auch zu beachten, dass die Threads mit der Renderan-

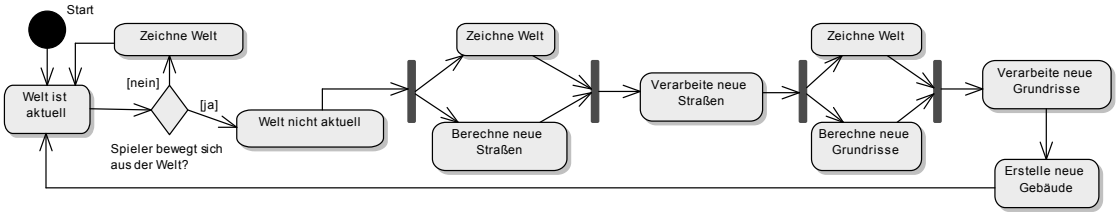


Abb. 4.6.3: Ablauf des Multithreadings

wendung korrekt synchronisiert werden müssen. Dies kann durch die Verwendung eines Zustandsautomaten erreicht werden. Der Zustand wird nur geändert, wenn alle Aufgaben des aktuellen Schrittes erfüllt sind. Der Zustandsautomat aus Abbildung 4.6.3 zeigt den Programmablauf. Es ist zu sehen, dass sich das Programm permanent in der Zeichenroutine befindet, sogar während neue Daten berechnet werden. Die Zeichenroutine wird nur kurzzeitig zur Verarbeitung der neuen Daten pausiert.

4.7 Debugging

Bei der Entwicklung des Systems hat sich herausgestellt, dass das Debugging einer derart komplexen Graphenstruktur bei zunehmender Größe nahezu unmöglich wird bzw. nur mit enormem Zeitaufwand möglich ist. Um dennoch komfortabel die Ergebnisse des Generierungsprozesses untersuchen zu können, sind einige optische Hilfsmittel geschaffen worden.

Grafische Darstellung des Graphen

Anfangs ist der Graph in einer Textform ausgegeben worden. Verschachtelungen sind durch Einrückungen und Verweise durch Angabe von IDs dargestellt worden. Diese Darstellung hat sich mit wachsender Komplexität als nicht mehr anwendbar herausgestellt. Eine grafische Darstellung der Knoten, Kanten und Attribute ist nötig gewesen.

Hierfür wird das DOT-Format verwendet, das in einfacher Form einen Graphen spezifiziert. Nachfolgend steht der DOT-Code des Graphen aus Abbildung 4.2.8 auf Seite 56.

Listing 4.5: Einfacher Graph im DOT-Format

```
1  digraph G {
2    0[label="& (0.0, 0.0, 0.0)\nid:0",style=filled,fillcolor=yellow];
3    1[label="F (10.0, 0.0, 0.0)\nid:1",style=filled,fillcolor=yellow];
4    2[label="[ (10.0, 0.0, 0.0)\nid:2"];
5    3[label="+ (10.0, 0.0, 0.0)\nid:3"];
6    4[label="P (10.0, 0.0, 0.0)\nid:4"];
7    5[label="] (10.0, 0.0, 0.0)\nid:5"];
8    6[label="[ (10.0, 0.0, 0.0)\nid:6"];
9    7[label="& (10.0, 0.0, 0.0)\nid:7"];
10   8[label="S (10.0, 0.0, 0.0)\nid:8"];
11   9[label="] (10.0, 0.0, 0.0)\nid:9"];
12   10[label="[ (10.0, 0.0, 0.0)\nid:10"];
13   11[label="& (10.0, 0.0, 0.0)\nid:11"];
14   12[label="S (10.0, 0.0, 0.0)\nid:12"];
15   13[label="] (10.0, 0.0, 0.0)\nid:13"];
16   14[label="& (10.0, 0.0, 0.0)\nid:14"];
17   15[label="S (10.0, 0.0, 0.0)\nid:15"];
18   0->1[label="",color=blue];
19   1->14[label="",color=blue];
```

```
20    1->10[label="",color=blue];
21    1->6[label="",color=blue];
22    1->2[label="",color=blue];
23    1->0[label="",color=red];
24    2->3[label="",color=blue];
25    [...]
26    15->14[label="",color=red];
27 }
```

Zu sehen ist, dass alle Knoten eine numerische ID erhalten, die mit einem Text (label) überschrieben werden können. Füllung, Aussehen des Rahmen und weitere optische Eigenschaften können zusätzlich angegeben werden. Die Verbindung der Knoten erfolgt mit dem Befehl „ID1->ID2“. Beschriftungen der Kanten und optisches Aussehen können erneut als Attribute angegeben werden. Die Umwandlung der Beschreibungssprache DOT erfolgt anschließend mit Hilfe eines Konverters. Hierfür wird das OpenSource-Programm GraphViz (Gansner et al.) eingesetzt. Es erzeugt Ausgabebilder sowohl als Vektorgrafiken (svg) als auch als Pixelgrafiken (z.B. png).

Mit der Möglichkeit, sich die gesamte Datenstruktur optisch aufbereitet ausgeben zu lassen und in dieser gezielt suchen zu können, ist eine sehr komfortable Programmanalysemöglichkeit geschaffen worden.

Anklicken von Objekten

Eine in dem Graphen nicht optisch kodierte Information ist die Position der einzelnen Knoten. Die Position der Knoten in der Welt wird zwar angegeben, jedoch entspricht diese nicht der Position im Debuggraph. Um dennoch Rückschlüsse aus der generierten Welt auf die Datenstruktur schließen zu können, kann zwecks Debugging mit dem Mauscursor ein Strahl durch die Welt geschossen werden. Dieser Strahl wird mit den Weltobjekten geschnitten und liefert somit alle Objekte unterhalb des Mauscursors. Die Weltobjekte enthalten als Attribut Informationen über die IDs der Knoten, aus denen das jeweilige Objekt entstanden ist. Ein Straßenobjekt zwischen zwei Knoten kann als Attribut den Bezeichner „streetA17B23“ enthalten, eine Kreuzung den Bezeichner „crossing17“. Nach den IDs kann anschließend gezielt gesucht werden und der Graph an entsprechender Stelle untersucht werden.

Lesbare Formen von Datentypen

An nahezu allen Stellen des Systems werden spezielle Datentypen verwendet, die keine lesbare Form besitzen, so dass diese zur Identifizierung geeignet wären. Während der Entwicklung ist dies jedoch zwingend nötig. Um diesem Mangel entgegenzuwirken besitzt jeder dieser Datentypen eine Funktion `toString()`, die diesen in eine lesbare Form umwandelt.

So wird beispielsweise ein Knotenobjekt in die Form „`<Ausdruck> (<Position>) <VorfolgerIDs><NachfolgerIDs>`“ umgewandelt, da sonst nur ein Zeiger auf dessen Speicherposition gegeben wäre.

Zeichnen von einfacher Geometrie

Eine immer wiederkehrende Aufgabe besteht darin, erkannte Schnittpunkte oder Zyklen auf Korrektheit zu überprüfen. Da es sehr zeitaufwendig ist, aus einer Liste von Punkten oder Segmenten heraus zu ermitteln, ob diese der gewünschten Form entsprechen, ist für diesen Zweck ein Hilfsprogramm geschrieben worden. Dieses Programm ermittelt mit regulären Ausdrücken automatisch aus einem Textabschnitt alle Vorkommen von Punktangaben und zeichnet aus diesen ein Polygon, Segmente oder Punkte.

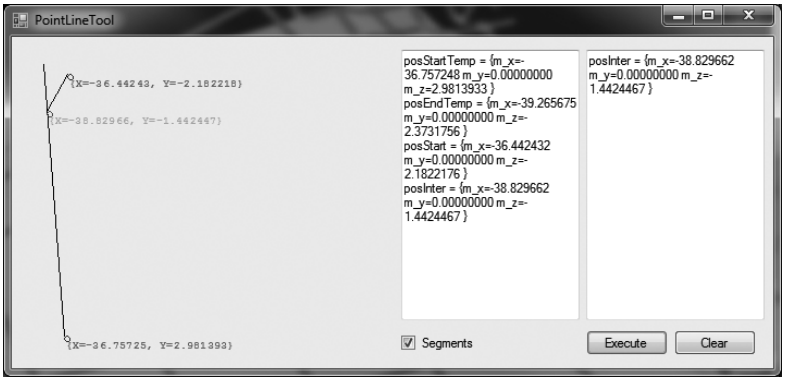


Abb. 4.7.1: Hilfsprogramm zum Zeichnen von Geometriedaten

Tracer

Das gesamte Backend enthält an entsprechenden Stellen im Programmcode Ausgaben über das aktuelle Geschehen. Der Tracer erhält als Angaben jeweils den Klassen- und Funktionsname, die Meldung und eventuell dazugehörige Parameter. Kann das Regelwerk nicht gelesen werden, wird ein Fehler der Form

```
1 Tracer::error("LSystemLoader", "loadFile", "File '%s' does not exist or is unaccessible.",
      _fileName.c_str());
```

generiert. Der Tracer erzeugt anschließend einen Eintrag der Form

```
1 [ERR] LSystemLoader:loadFile . File 'streetnet.txt' does not exist or is unaccessible.
```

auf der Konsole und schreibt diesen in eine Logdatei. Ist in der Konfigurationsdatei angegeben, dass eine Ausnahme generiert werden soll, wird an der Stelle der Fehlermeldung diese generiert, was im Debug-Modus zur Unterbrechung des Programms führt. Mit dem Tracer ist es möglich, den Programmablauf im Detail zu verfolgen und schnell auf Fehler oder Probleme zurückzuschließen. Zudem wird je nach Fehlerlevel dem Backend eine Chance geboten, einen Fehler, falls vorgesehen, selbst zu behandeln und in einen normalen Ablauf zurückzukehren. So führt der obige Fehler bei deaktivierter Ausnahmegenerierung für Fehler nur dazu, dass das gewählte Regelwerk nicht geladen wird. Über den Optionsdialog kann der Benutzer jedoch ein anderes Regelwerk wählen und somit den Fehler umgehen.

Konfigurationsdatei

Um grundlegende Eigenschaften am System zu manipulieren, besitzt das System eine Konfigurationsdatei, die Parameter für den Graphen, das L-System und das Gebäudeaussehen enthält. Diese Konfigurationsdatei ist im Stil einer INI-Datei gehalten. Das L-System kann über die Konfiguration wie folgt beeinflusst werden.

Listing 4.6: Optionen des L-Systems

```
[LSystem]
systemName=streetnet # initiales Regelwerk
iterationThreshold=0 # verbleibende Platzhalter vor Abbruch
volumeRadius=100 # initiale Breite/Länge der Weltgrenze
minimalDrawLength=3.0 # minimale Kantenlänge (Straßenlänge)
```

Das Graphensystem kann ebenso im Verhalten angepasst werden.

Listing 4.7: Optionen des Graphen

```
[Graph]
# Einfügung verhindern, falls ein Schnittpunkt gefunden wurde
failOnIntersection=0
# Länge in die Richtung des einzufügenden Elements, die als Vorwärtsschnitt behandelt werden soll
collisionOffset=2.5
# minimaler Winkel zwischen den Segmenten, der eingehaltern werden soll (in Radiant - 45°)
minimalCrossingAngle=0.7853981
```

Das Erzeugungsverhalten kann an zwei Stellen angepasst werden. Jede Programmklasse besitzt einen eigenen Konfigurationsbereich.

Listing 4.8: Optionen für die Gebäudegenerierung

```
[Lots]
BuildingMargin=0.6      # Abstand der Gebäude zueinander
RoadMargin=0.6          # Abstand der Gebäude zum Straßenzentrum/Halbe Straßenbreite

[Buildings]
createBuildings=true    # Legt fest ob die Gebäude zu Programmstart erzeugt werden sollen
maxBuildingHeight=18.0  # Legt die maximale Gebäudehöhe fest
minBuildingHeight=7.0   # Legt die minimale Gebäudehöhe fest
```

Auch das Laufzeitverhalten der Anwendung kann manipuliert werden.

Listing 4.9: Optionen des Tracers

```
[Tracer]
# Gibt an welche Nachrichten bis zu welcher Priorität angezeigt werden (2-Info, 1-Warnung, 0-Fehler)
messagePriorityLevel=2
# Gibt an ob, das Programm einen Ausnahmefehler generieren soll, falls eine Fehlermeldung erzeugt wurde.
raiseExceptionOnError=true
# Gibt an ob, das Programm einen Ausnahmefehler generieren soll, falls eine Warnung erzeugt wurde.
raiseExceptionOnWarning=false
```

Mit diesen Einstellungen kann man das Programmverhalten je nach Anwendungsszenario zu beeinflussen. Zudem lässt sich das Verhalten des Programms grundlegend verändern, ohne dass eine Neuerstellung der Anwendung nötig wird.

Kapitel 5

5 Techniken des Frontends

5.1 Bewegung in der Welt

Eine Herausforderung an das Frontend besteht in der korrekten Bewegung entlang der vom Backend zur Verfügung gestellten Pfade. Hierbei haben sich einige Aufgaben gestellt.

1. Auswahl des nächsten Weges
2. Abbiegen im Graph
3. Bewegung entlang senkrechter Kanten
4. Blickfeld (Kamera)

Auswahl des nächsten Weges

Bei der Auswahl des Szenarios futuristischer Städte ist entschieden worden, dass das Spielkonzept sich an den Film „Tron“(1982) anlehnen soll. Dem Film entsprechend ist die Steuerung auf dem Straßennetz umgesetzt. Die Spielfigur bewegt sich ununterbrochen mit einer konstanten Geschwindigkeit vorwärts. Wird eine Abzweigung erreicht, so wird der Weg gewählt, der nahezu gradlinig gegenüber liegt. Möchte der Spieler einen anderen Weg wählen, so erfolgt dies durch Angabe einer Abweichung vom Standardweg. Soll beispielsweise links abgebogen werden, so kann dies durch einmaliges Drücken der linken Pfeiltaste erfolgen. Die Wegwahl muss hierbei rechtzeitig vor dem Erreichen der Kreuzung geschehen. Diese Auswahl wird auf dem jeweiligen Streckensegment gespeichert und nach Überqueren jeder Kreuzung zurückgesetzt. Das Fortbewegungssystem entspricht somit am ehesten einer Weiche, die mit Links- und Rechtsbefehlen umgestellt werden kann.

Eine Sonderheit bilden die senkrechten Streckensegmente, die als Plattformen symbolisiert werden. Diese können durch Tasten für die Auf- und Abwärtsrichtung aktiviert werden. Hierbei wird die Weichenstellung der jeweils nicht aktiven Ebene (links/rechts bzw. auf-/abwärts) auf die neutrale Stellung zurückgesetzt. Die Unterscheidung zwischen

diesen Ebenen wird vom Backend vorgenommen. Das Frontend kann durch Angabe einer Normalen, die die neutrale Ebene angibt, die Sortierung der senkrechten und waagrechten Optionen beeinflussen.

Abbiegen im Graph

Beim Richtungswechsel der Spielfigur muss vom Frontend die aktuelle ($\overrightarrow{d}$) und gewünschte ($\overrightarrow{v}$) Blickrichtung bestimmt werden. Erstere ist durch die aktive Bewegungsrichtung bekannt und letztere kann über den Vektor, den die einzubiegende Straße besitzt, bestimmt werden. Zu beachten ist nun, dass die Spielfigur sich um ihre Achse entlang des kürzeren Drehwinkels bewegt. Die verwendete Grafikengine OGRE benötigt hierzu jedoch Start- und Endorientierung des Objektes als Quaternion. Hierfür ist es nötig, ein Quaternion (Q) wie in Formel 5.1.1 aus drei zueinander orthogonalen Vektoren (hier $\overrightarrow{f}$ (front),$\overrightarrow{u}$ (up), $\overrightarrow{l}$ (left)) aufzubauen. Hierzu wird der wie in Abschnitt 2.1.2 auf Seite 8 berechnete Upvektor ($\overrightarrow{u}$) verwendet.

$$\begin{aligned} \overrightarrow{d} &= \frac{P_{Ziel} - P_{Position}}{|P_{Ziel} - P_{Position}|} \\ \overrightarrow{l} &= \overrightarrow{d} \times \overrightarrow{u} \\ \overrightarrow{f} &= \overrightarrow{u} \times \overrightarrow{l} \\ Q &= (\overrightarrow{f}\,\overrightarrow{u}\,\overrightarrow{l}) \end{aligned} \tag{5.1.1}$$

OGRE kann nun zwischen der aktuellen Orientierung und der Zielorientierung (Q) so interpolieren, dass der kürzere Rotationsweg verwendet wird. Die Kamera wird hierbei entlang der lokalen x-Achse ($\overrightarrow{f}$) der Spielfigur ausgerichtet.

Bewegung entlang steiler Kanten

Anders als bei in der Waagerechten liegenden Straßen ist bei Straßen mit Gefälle eine andere Rotationsreihenfolge nötig. Im waagerechten Fall ist eine einfache Rotation um den Upvektor wie zuvor beschrieben möglich. Beim Passieren einer steilen Straße wie im linken Bild aus Abbildung 5.1.1 auf der nächsten Seite muss sich der Upvektor jedoch verändern, so dass der Spieler entlang des Aufstiegs bzw. des Gefälles schaut. Nach dem Verlassen der Steigung muss der Upvektor wieder nach oben ausgerichtet werden.

Vor dem Betreten einer neuen Kante und vor dem Verlassen einer Kante wird immer zwischen dem aktiven Upvektor und der Straßennormalen (schwarze Pfeile) interpoliert. An den Stellen bei denen eine Interpolation des Upvektors erfolgt, ist in der Abbildung eine Rotationsachse eingezeichnet. Durch das ausschließliche Nicken der Kamera bzw. der Spielfigur können jedoch nur Straßen, die in der Projektion koinzident sind, passiert werden.

Im rechten Bild aus Abbildung 5.1.1 auf der nächsten Seite findet ein Richtungswechsel zwischen zwei Straßen statt, die in ihrer Projektion nicht koinzident sind. Der Richtungswechsel lässt sich am Übergang nicht mehr durch eine einzelne Achsenrotation

(schwarz) ausdrücken. Die Rotation muss in drei Teilroationen zerlegt werden, die in der Abbildung grau eingezeichnet sind.

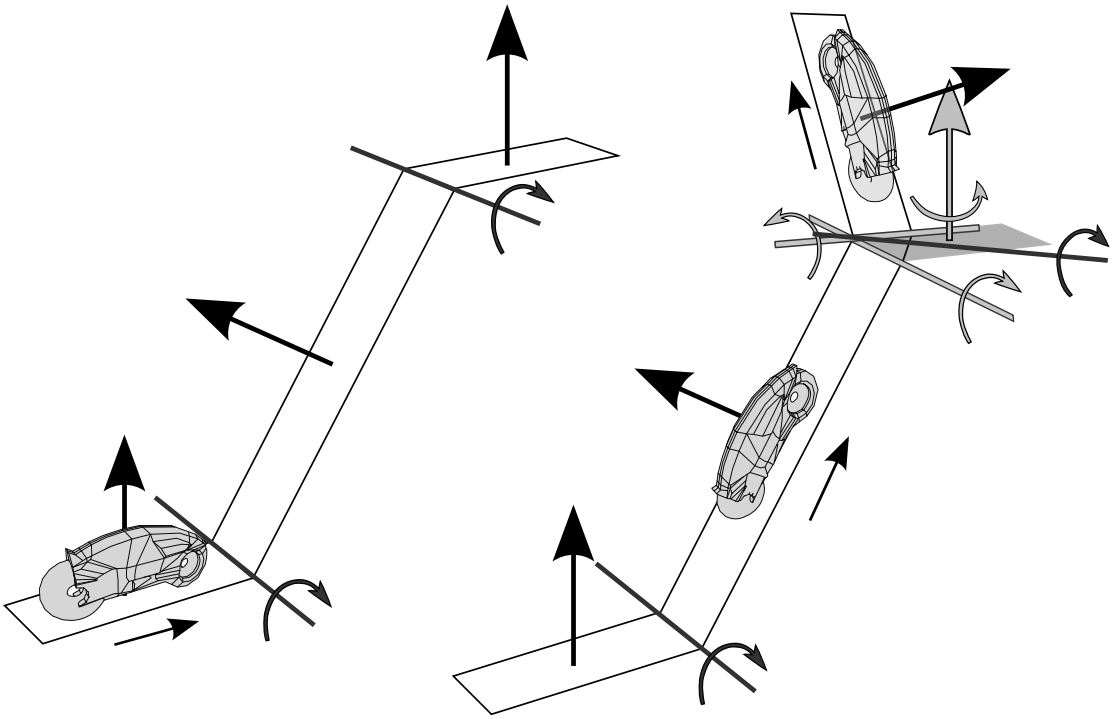


Abb. 5.1.1: Spielerausrichtung bei steilen Kanten

Wird keine Zerlegung in Nicken, Gieren und Nicken wie in Abbildung 5.1.2 durchgeführt, wäre der Horizont der Kamera verschoben bzw. die Spielfigur stünde schräg, wie in Abbildung 5.1.3 auf der nächsten Seite zu ersehen ist.

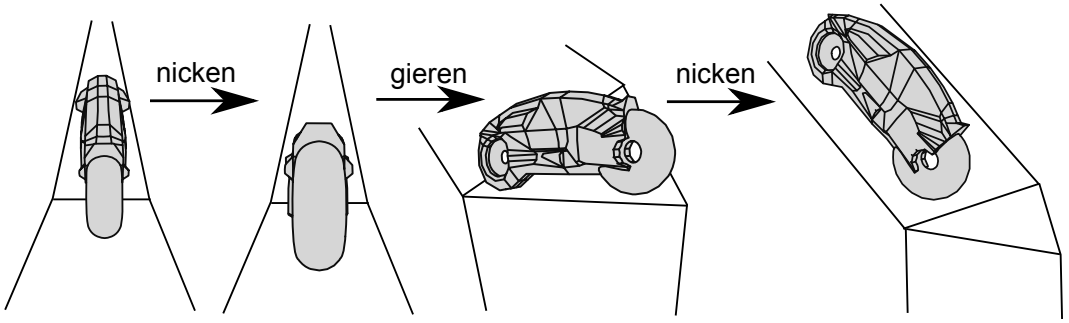


Abb. 5.1.2: Rotationsreihenfolge für Spielerausrichtung

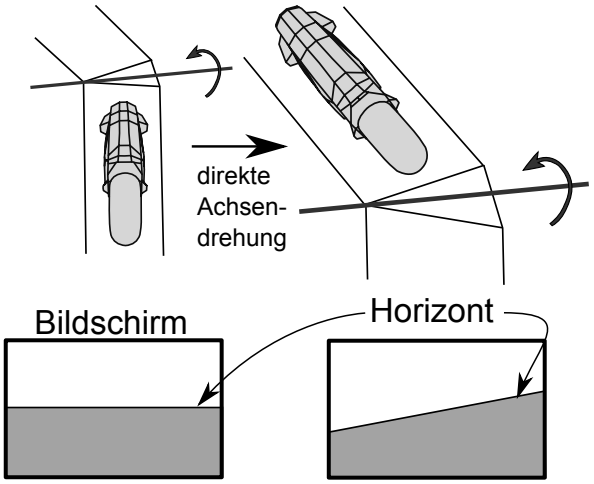


Abb. 5.1.3: Direkte Achsenrotation führt zu schiefem Horizont

Bewegung entlang senkrechter Kanten

Eine Bewegung entlang senkrechter Kanten führt dazu, dass eine Rotation zwischen den verschiedenen Upvektoren, wie bisher, mathematisch nicht mehr eindeutig wäre und so zu einem Orientierungsverlust führen würde. Um diesem Problem entgegenzuwirken, wird an diesen Stellen im Frontend eine schwebende Plattform verwendet, die den Spieler entlang einer solchen Kante transportiert. Hierbei muss vom Frontend entschieden werden, wie am Startpunkt der Plattform mit weiteren Spielern verfahren wird. Hierbei ist es am sinnvollsten die senkrechte Kante solange zu deaktivieren, bis die Plattform zu ihrer Ausgangsposition zurückgekehrt ist. Um dies zum Einen zu beschleunigen und das Problem der fehlenden Plattform am Zielpunkt zu lösen, werden zwei Plattformen verwendet, die sich zeitgleich zwischen Start und Ziel bewegen. Hier muss jedoch bedacht werden, dass es zu einer Kollision beider Plattformen in der Mitte des Weges kommen wird. Dies kann verhindert werden, indem ein leicht gekrümmter Pfad gewählt wird, der keine Kollision mit Gebäuden besitzt.

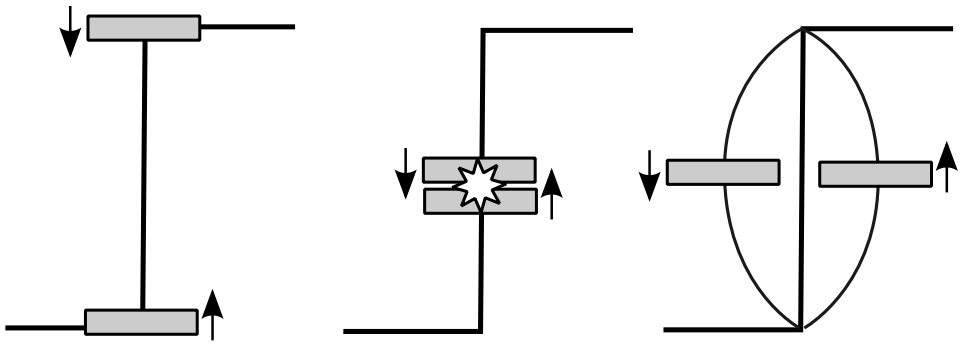


Abb. 5.1.4: Verhinderung kollidierender Lifte

Bei der Verwendung von Objekten, die die Spielfigur bewegen, ist zu berücksichtigen, dass sich die Objektabhängigkeiten verändern. Dies bedeutet, dass die Spielfigur ein Weltobjekt ist und globale Koordinaten besitzt. Betritt die Spielfigur nun z. B. eine Plattform, müssen die Koordinaten der Spielfigur in lokale Koordinaten, in Abhängigkeit der Plattform, umgerechnet werden, da die Spielfigur als Kindobjekt der Plattform zugeordnet wird. Erreicht die Plattform die Zielposition, muss die neue lokale Position erneut in eine globale Position umgerechnet werden und die Spielfigur wieder der Spielwelt zugeordnet werden.

Eine Beibehaltung der globalen Koordinaten der Spielfigur wäre zwar möglich, würde jedoch bedingt durch unterschiedliche Aktualisierungsintervalle der animierten Plattformen und der manuell bewegten Spielfigur zu einem Ruckeln führen. Dies tritt auf, wenn das Bild gerendert wird, nachdem erst eines der beiden interagierenden Objekte verschoben wurde.

Blickfeld (Kamera)

Da die Spielwelt sowohl in freiem Flug als auch in erster Person betrachtet werden kann, ist es nötig, korrekt zwischen diesen Ansichten wechseln zu können. Hierbei wird für die Vogelperspektive eine Spielfigur auf dem Straßennetz dargestellt. Wechselt der Anwender in die erste Person, nimmt die Kamera die Orientierung und Position der Spielfigur an und die Spielfigur wird ausgeblendet. Verlässt der Anwender die Ansicht aus erster Person, so wird die Kamera in eine entferntere Umlaufbahn um die nun wieder eingeblendete Spielfigur gesetzt.

5.2 Minimap

Um sich in einer dynamischen Welt orientieren zu können, ist eine Minimap eingeführt worden, die einen Ausschnitt der aktuellen Welt darstellt. Kanten des Straßennetzes werden im Frontend zwischengespeichert, um einen Ausschnitt der Welt erzeugen zu können. In regelmäßigen Abständen wird dann ein neuer Ausschnitt erzeugt. Zuerst wird eine Box (R) bestimmt, deren in der Waagerechten liegenden Kanten in Abhängigkeit der Spielerposition gleich weit entfernt sind.

Die Höhenausdehnung wird hierbei so gesetzt, dass jeweils nur die Straßen gesehen werden können, die sich in etwa auf gleicher Höhe befinden. Dann werden die Straßen ermittelt, die in dieser liegen oder diese teilweise passieren. Ist dies geschehen, müssen die gefundenen Straßensegmente für die Textur (T) der Minimap normiert werden (siehe Formel 5.2.1). Hierbei wird ein zweidimensionales Abbild der Welt aus der Vogelperspektive erstellt.

$$P_{neu} = \left(\frac{R_{links} \cdot (-P_x)}{R_{rechts} - R_{links}} \cdot T_x, \frac{R_{oben} \cdot (-P_z)}{R_{unten} - R_{oben}} \cdot T_y \right) \tag{5.2.1}$$

Um die Straßen zu zeichnen wird der Bresenham Algorithmus für Geraden verwendet, der für alle Oktanten des Koordinatensystems anwendbar ist. Dieser hat den Vorteil, dass Rundungsfehler minimiert werden, die durch die Diskretisierung kontinuierlicher

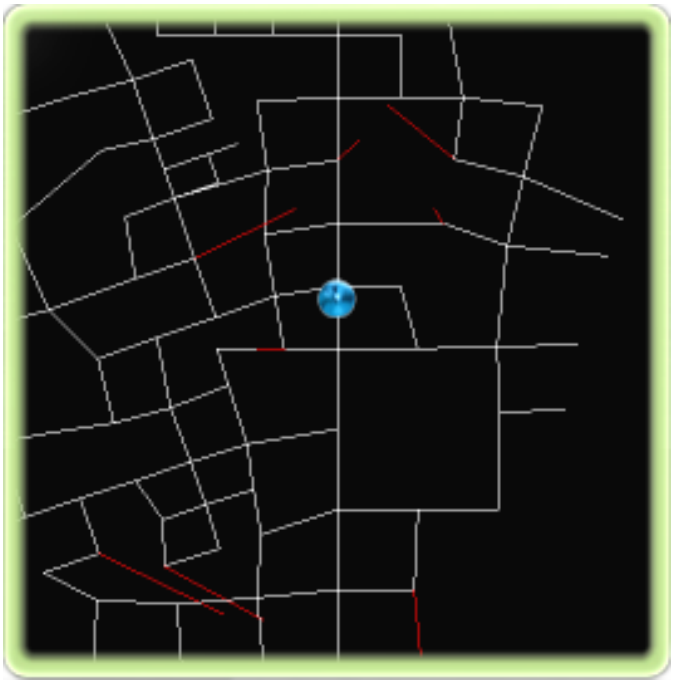


Abb. 5.2.1: Minimap

Koordinaten (Bresenham, 1965; Wikipedia, 2010a) zustande kommen. Der Algorithmus kommt hierbei ohne Gleitkommazahlen, Multiplikationen und Divisionen aus.

Bei der Verwendung einer Textur mit fortlaufendem Speicher ist zu beachten, dass Geraden, die nicht vollständig innerhalb der Textur liegen, nur im gültigen Bereich gezeichnet werden, da sonst die ungültigen Segmente zeilenversetzt gezeichnet werden würden.

Bei der Erstellung der Karte ist die Bedeutung von Straßen farbkodiert worden, so dass eine leichtere Orientierung möglich ist. So werden steile Straßen bzw. Aufzüge zwischen verschiedenen Höhenebenen rot gezeichnet und normale Straßen weiß. Zudem wird die Position des Spielers auf der Karte angezeigt.

Dynamische Texturen

Um die Minimap zu realisieren, kommt eine dynamische Textur zum Einsatz. Die Textur wird in OGRE, wie jede Textur, in ein Material gekapselt, das in einer Materialdatei definiert werden kann. Auf den Platzhalter „buffer“, der normalerweise auf eine Texturdatei verweisen würde, wird später zurückgegriffen.

Listing 5.1: Materialeintrag einer dynamischen Textur

```
1 material GUI/FGBuffer {
2   technique {
3     pass {
4       lighting off
5       depth_write off
6
7       texture_unit {
8         texture buffer
```

```
9         }
10      }
11    }
12  }
```

Sind die Materialdateien von OGREs Ressourcenmanager eingelesen worden, wird die Referenz zur „buffer“-Textur gesucht und notiert. Das Material, das diese Textur beinhaltet, wird in ein neues Menüobjekt eingefügt.

Listing 5.2: Dynamisches Material initialisieren und einem Overlayelement zuweisen

```
1   // create the dynamic texture with RGBA texels (textureWidth = 512)
2   TexturePtr tex = TextureManager::getSingleton().createManual("buffer", ResourceGroupManager::
        DEFAULT_RESOURCE_GROUP_NAME,
3     TEX_TYPE_2D, m_textureWidth, m_textureWidth, 0, PixelFormat::PF_BYTE_RGBA,
          TU_DYNAMIC_WRITE_ONLY);
4
5   // save a reference to the texture buffer
6   m_fgTexBuffer = tex->getBuffer();
7
8   // initialise the texture to black
9   m_fgTexBuffer->lock(HardwareBuffer::HBL_DISCARD);
10  memset(m_fgTexBuffer->getCurrentLock().data, 0x00, m_fgTexBuffer->getSizeInBytes());
11  m_fgTexBuffer->unlock();
12
13  // retreive ogres overlay manger
14  Ogre::OverlayManager& om = Ogre::OverlayManager::getSingleton();
15  // create sample thumbnail overlay
16  m_miniMap = (Ogre::BorderPanelOverlayElement*)om.createOverlayElementFromTemplate("SdkTrays/
        Picture", "BorderPanel", "BufferOverlay");
17  // set alignment
18  m_miniMap->setHorizontalAlignment(Ogre::GHA_RIGHT);
19  m_miniMap->setVerticalAlignment(Ogre::GVA_BOTTOM);
20  // assign the dynamic texture
21  m_miniMap->setMaterialName("GUI/FGBuffer");
22  // add the overlay to the gui manager
23  m_TrayMgr->getTraysLayer()->add2D(m_miniMap);
```

In diesem Zustand wird ein schwarzes Viereck im Vordergrund angezeigt, das in einen Menürahmen eingefasst ist. Der nachstehende Quellcode zeichnet einen Punkt an die übergebene Position mit entsprechender Farbe.

Listing 5.3: Zeichnen eines Punktes in eine dynamische Textur

```
1  void drawPixel(int _x0, int _y0, Ogre::ColourValue _color) {
2    m_fgTexBuffer->lock(HardwareBuffer::HBL_NORMAL);
3
4    // get access to raw texel data
5    uint8* data = (uint8*)m_fgTexBuffer->getCurrentLock().data;
6    unsigned int maximum = m_textureWidth*m_textureWidth*4;
7
8    // determine pixel position in memory
9    target = (_x0 + (_y0*m_textureWidth))*4+3;
10   // assure that the position is in range
11   if (target < maximum && target >= 0 && _y0 < m_textureWidth && _x0 < m_textureWidth)
12   {
13     // convert and assign the pixel color
14     data[(_x0 + (_y0*m_textureWidth))*4+3] = (uint)_color.a*255;
15     data[(_x0 + (_y0*m_textureWidth))*4+2] = (uint)_color.r*255;
16     data[(_x0 + (_y0*m_textureWidth))*4+1] = (uint)_color.g*255;
17     data[(_x0 + (_y0*m_textureWidth))*4+0] = (uint)_color.b*255;
18   }
19
20   m_fgTexBuffer->unlock();
21 }
```

In diesem Beispiel wird die Textur sehr kostenintensiv vor und nach jedem Pixel gesperrt und entsperrt. Idealerweise sollte dies einmalig vor und nach allen anstehenden Zeichenoperationen geschehen.

5.3 Eine endlose Welt echtzeitfähig halten

Über die Spielerposition ist es möglich, ein Rechteck zu bestimmen, in dem die Welt jeweils erzeugt werden soll. Dieses Feld wird als Parameter verwendet. Bewegt sich der Spieler auf den Rand des zuletzt definierten Feldes zu, so wird ein neues bestimmt und zur Befüllung an das Backend weitergegeben. Die Position des Spielers wird aus diesem Grund regelmäßig überprüft.

Zu beachten ist, dass das Frontend regelmäßig Geometrie verwirft, die weit entfernt von der Spielerposition liegt. Sollte der Spieler an eine Position zurückkehren, an der die Daten verworfen wurden, können diese über eine Funktion regional erneut angefordert bzw. rekonstruiert (siehe Seeds in 4.2.10 auf Seite 53) werden.

5.4 Texturierung der Gebäude

Bei der Erstellung von prozeduralen Gebäuden durch das Backend wird eine einfache geometrische Beschreibung zurückgeliefert. Eine Texturierung wird jedoch nicht durchgeführt und ist vom Frontend zu erledigen. Wände, Decken und bei schwebenden Gebäuden müssen auch Böden vom Frontend texturiert werden.

Jede Wand wird durch ein Viereck repräsentiert, dieses kann abhängig von der Länge in u- und v-Richtung mit einer nahtlosen Textur überzogen werden. Dabei ist zu beachten, dass die gewählten Texturkoordinaten auf die nächstgelegene Ganzzahl gerundet werden, jedoch immer mindestens eine Wiederholung der Textur stattfindet. So wird ein realistisches Fassadenbild erzeugt, bei dem keine Fenster auf der Wandtextur abgeschnitten oder zu stark gezerrt werden.

Die Decken und Böden werden durch eine Dreiecksliste repräsentiert. Da diese Flächen eine beliebige Form annehmen können, wird eine einfache planare Texturierung vorgenommen. Die Koordinaten entsprechen hierbei denen der Projektion auf die XZ-Ebene.

Für die Beleuchtung und das Culling ist es weiterhin erforderlich, die Flächennormalen zu berechnen. Diese lassen sich für die Wände leicht über die Vertexpositionen der Wandkanten bestimmen.

$$\overrightarrow{n} = (\overrightarrow{p_1} - \overrightarrow{p_0}) \times (\overrightarrow{p_1} - \overrightarrow{p_3})$$

Bei der Normalenberechnung der Decken und Böden ist das Vorgehen das Gleiche, jedoch muss der negative Umlaufsinn der Bodenkonturen bei der Erstellung der Dreiecke beachtet werden. Diese müssen in umgekehrter Reihenfolge erstellt werden, da diese sonst, trotz korrekter Normale, durch das Culling nicht gezeichnet werden würden.

5.5 Optische Aufbereitung der Daten

Eine Aufbereitung der Daten des Backends vom Fontend ist problemlos möglich. Dies kann exemplarisch an verschiedenen Beispielen gezeigt werden.

Aufbereitung der Straßen

Die Straßen werden vom Backend an Ebenen ausgerichtet, die parallel zueinander sind und mit Verbindungsstraßen verknüpft werden. Eine optische Aufbereitung dieser Ebenen besteht darin, dass die Kreuzungen der oberen Straßenebenen in ihrer Höhe leicht verschoben werden. Somit entstehen in diesen Ebenen zwischen diesen Kreuzungen Straßen, die ein Gefälle bzw. eine Steigung besitzen. Die Bewegung entlang der Straßen wird in Abschnitt 5.1 auf Seite 79 beschrieben.

Zudem hat man sich dazu entschieden senkrechte Straßensegmente durch Aufzüge, die als schwebende Plattformen dargestellt werden, zu ersetzen. Dies reduziert die Gefahr, dass der Nutzer die Orientierung verliert, während er sich zwischen einem waagrechten und senkrechten Segment bewegt.

Abwechslungsreichere Gebäude

Eine weitere Aufwertung besteht in der Verwendung von animierten Objekten. Im Szenario der futuristischen Großstädte werden die Gebäude der oberen Ebenen als schwebende Objekte dargestellt, die sich geringfügig zwischen der vorgegebenen und leicht verschobenen Höhe bewegen. Um ein abwechslungsreicheres Stadtbild zu erreichen, werden für die Gebäude mehrere Texturen verwendet, die zufällig ausgewählt und verwendet werden.

Atmosphäre

Mit wenigen Mitteln ist es möglich, das Gesamtbild deutlich aufzubereiten. Im Frontend wird ein animierter Himmel in Form eines Skydomes verwendet, der die Welt atmosphärisch stark aufwertet. Auf der untersten Ebene wird eine Bodenplatte eingefügt, die sich stets unterhalb der Spielfigur befindet und somit eine bessere räumliche Aufteilung vermittelt. Zuletzt wird die Spielwelt mit einem direktionalen Licht beleuchtet und mit Schatten versehen. Insgesamt ergibt sich hieraus ein deutlich realistischeres Gesamtbild.

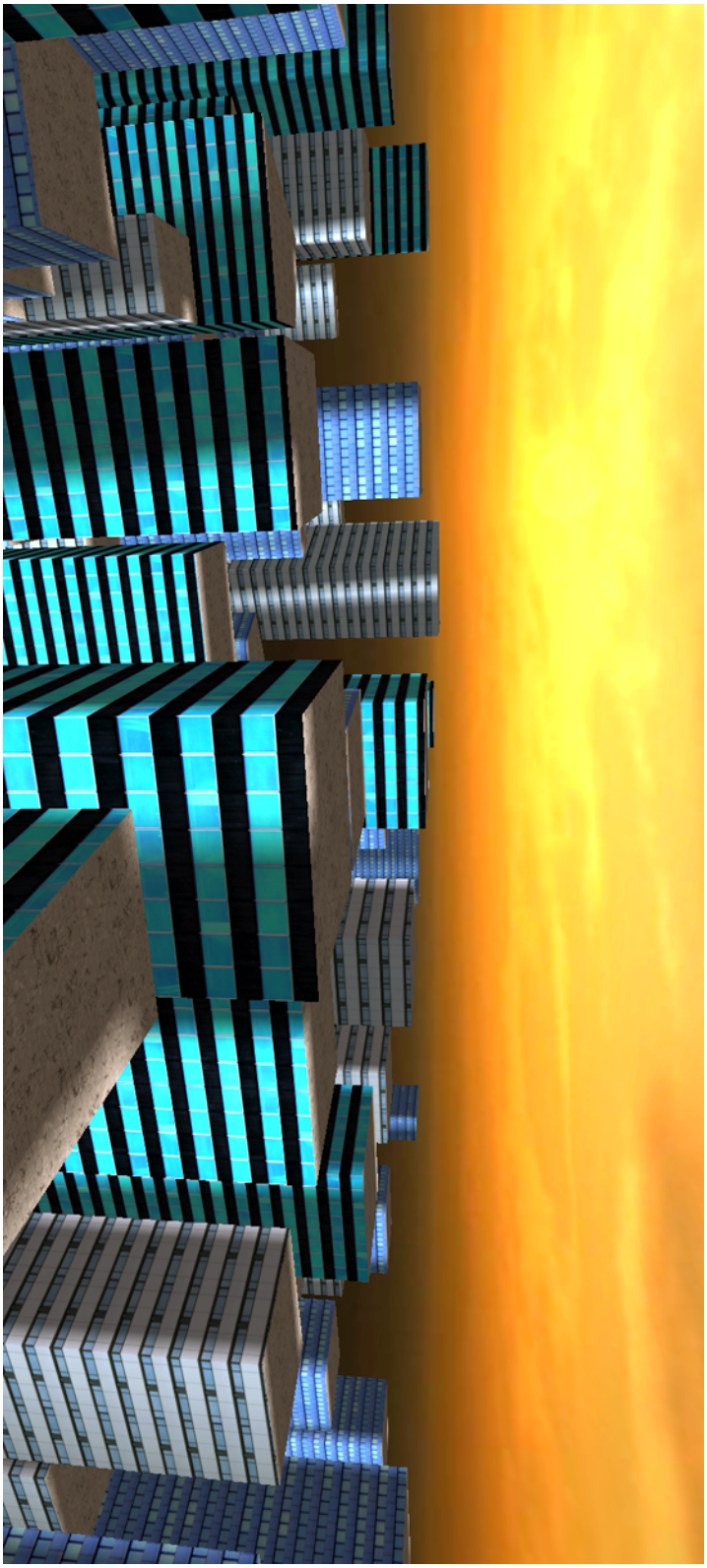


Abb. 5.5.1: Screenshot mit aktivierten atmosphärischen Objekten

Kapitel 6

6 Bewertung des Systems

Die Evaluierung des Backends durch Probanden gestaltet sich schwierig, da eine objektive Beurteilung der Ergebnisse des Backends nahezu nicht möglich ist. Denn die optische Qualität des Ergebnisses vom Backend ist von der Aufbereitung des Frontends abhängig. Ein Vergleich zwischen Generierungsergebnissen und realen Städten wäre zwar denkbar, müsste jedoch stark auf das Wesentliche reduziert werden, so dass viele technische Aspekte vernachlässigt werden müssten. Aus diesem Grund findet eine Bewertung des Backends über das Frontend statt. Anhand softwaretechnischer Methoden wie z.B. Anforderungslisten und Eigenschaften wie Wiederverwendbarkeit wird das System evaluiert. Durch das Frontend wird gezeigt, welche zusätzlichen Fähigkeiten im Backend implementiert und welche durch dieses bereits zur Verfügung gestellt worden sind.

6.1 Wiederverwendbarkeit des Systems

Ein großer Wert ist auf die Wiederverwendbarkeit des Systems gelegt worden. Dies lässt sich daran festmachen, dass das Ergebnis immer von einem frei definierbaren Regelwerk bestimmt wird. Zudem ist es möglich, während der Laufzeit das Regelwerk zu wechseln. Dies ist wünschenswert um verschiedene Welten schaffen zu können. Im Szenario der futuristischen Städte besteht der Anwendungsfall für Computerspiele beispielsweise darin, zwischen verschiedenen Stadtformen zu wechseln. So kann im Betrieb zwischen Kleinstadt, Großstadt und futuristischer Stadt mit schwebenden Gebäuden umgeschaltet werden.

Ein weiterer Vorteil besteht in der klar definierten Schnittstelle (siehe Abbildung 6.1.1 auf der nächsten Seite), die durch ein DLL-Interface[1] festgelegt ist. So kann das Frontend beliebig ausgetauscht oder angepasst werden, um mit dem Backend z. B. Pflanzen oder einen ganzen Wald zu generieren.

Zusätzlich wurde das Backend so konzipiert, dass die Klasse für die Regelauswertung beliebig ausgetauscht werden kann, so dass ein anderer Parser als der eines L-Systems verwendet werden kann.

[1] DLL: Dynamic Linkable Library (Microsoft Windows spezifisch)

Abb. 6.1.1: Schnittstelle der DLL

6.2 Anforderungen an das System

Neben dem Klassendiagramm sind zu Beginn Anforderungen verfasst worden, die im System umgesetzt werden sollten. Die umgangssprachlich verfassten Anforderungen haben im Verlauf der Umsetzung zu größeren Überlegungen geführt, da sie nicht so leicht auf den Programmcode abzubilden waren. Von diesen Anforderungen werden jeweils einige des Front- und Backends exemplarisch behandelt.

Anforderungen an das Frontend

Die Anforderungen an das Frontend lassen sich in verschiedene Kategorien einteilen, wie Spielkonzept und Darstellung.

Anforderungen an das Spielkonzept

Anforderung 1 Der Anwender steuert ein Fahrzeug, um sich auf den Straßen fortzubewegen. (Priorität: mittel)

Umsetzung Der Anwender kann von der Vogelperspektive in die erste Person auf die Straße versetzt werden. Er schaut in Richtung der nächsten Kreuzung.

Anforderung 2 Die Fahrzeugsteuerung soll an die der Lightcycles aus dem Film „Tron“ (Lisberger/MacBird, 1982) angelehnt sein. Dies bedeutet, dass das Fahrzeug immer geradeaus fährt. Es kann unmittelbar im 90°-Winkel links und rechts abgebogen werden. (Priorität: mittel)

Umsetzung Das Fahrzeug fährt immer entlang des Weges, der den höchsten Winkel zur Herkunftsstraße besitzt. Eine geradeaus verlaufende Straße besitzt den höchs-

ten Winkel von 180°. Gibt es mehr als eine links- oder rechtsläufige Straße, kann durch mehrmaliges Links- bzw. Rechtswählen zwischen diesen Straßen gewechselt werden. Der Straßenwechsel muss bis kurz vor der Abbiegung bekannt sein, um den Richtungswechsel animieren zu können. Zusätzlich ist zu beachten, dass an einer Kreuzung Passagen zu anderen Höhenebenen bestehen können. Für die Nutzung dieser Passagen ist zusätzlich eine Möglichkeit zum Wechsel zwischen diesen realisiert worden.

Anforderung 3 Der Spieler sammelt während einer Runde Gegenstände ein. (Priorität: niedrig)

Umsetzung Neben der Spielsteuerung ist zeitlich bedingt leider die Umsetzung einer weiteren Spiellogik nicht möglich gewesen.

Anforderungen an die Darstellung

Anforderung 1 Grafik wird aus Beschreibungssprache vom Hauptprogramm dargestellt. (Priorität: hoch)

Umsetzung Gebäude werden als Kontur aus Punkten repräsentiert, Straßen als einfache Kanten zwischen zwei Punkten und Kreuzungen als Triangulationen. Die MeshFactory überführt die Konturen in Gebäude, die Straßenkanten in dreidimensionale Straßenobjekte und texturiert die triangulierten Kreuzungen.

Anmerkung Die Straßenkreuzungen könnten durch das Frontend eigenständig berechnet werden und somit optisch beliebig abgewandelt werden.

Anforderung 2 Gebäude und Straßen werden mit leuchtenden Silhouetten dargestellt und sollen somit die Optik der Welt aus „Tron“ darstellen. (Priorität: niedrig)

Umsetzung Durch den technischen Mehraufwand Straßen korrekt zu texturieren, insbesondere die Kreuzungen, ist die Idee der ausschließlichen Silhouettenrepräsentation verworfen worden. Straßen werden mit normalen Straßentexturen dargestellt und Kreuzungen erhalten eine einfache planare Textur.

Anmerkung Um nicht für jede Straßenform eine eigene Textur zu erstellen, wäre die Verwendung eines NPR-Shaders (non photorealistic) möglich gewesen, um nur die Silhouetten der Straßen zu zeichnen. Das Frontend müsste jedoch um eine Shaderfunktionalität erweitert werden, die nichtautomatisierte Shader, wie das Shadowmapping der 3D-Engine, erlaubt.

Anforderungen an das Backend

Auch für das Backend sind im Vorfeld, nach Erschließung vorhandener Techniken, Anforderungen erarbeitet worden. Diese bewegen sich alle in einem höheren Prioritätsbereich, da der Schwerpunkt der Arbeit auf dem Backend liegt.

Anforderung 1 Straßen werden mit rekursivem L-System erzeugt. (Priorität: hoch)

Umsetzung Die Anforderung ist erfüllt. Das L-System kann zusätzlich durch globale Kriterien beeinflusst werden.

Anforderung 2 Straßennetz wird intern als ungerichteter Graph repräsentiert. (Priorität: hoch)

Umsetzung Ist erfüllt. Zusätzlich werden Attribute für Kanten und Knoten unterstützt.

Anforderung 3 Straßen werden nach dem Manhattanschema erzeugt. (Priorität: mittel)

Umsetzung Ist durch entsprechendes Regelwerk des L-Systems realisiert.

Anforderung 4 Straßennetze der verschiedenen Ebenen haben eine Ähnlichkeit von ca. 70%. (Priorität: mittel)

Umsetzung Diese Anforderung ist angepasst worden, um eine bessere Unterstützung des L-Systems zu ermöglichen. Die Ebenen werden durch einen Regelsatz des L-Systems erstellt, wodurch ein organischeres Bild entsteht. Der Grundgedanke wird dennoch erfüllt. Durch die Ähnlichkeit sollte gewährleistet werden, dass mehrere Verbindungen zwischen den Ebenen zustande kommen. Dies geschieht nun durch die Nachbarschaftssuche für senkrecht verlaufende Kanten.

Anforderung 5 Die Spielewelt wird nur in einem vorgegebenen Umkreis generiert. (Priorität: mittel)

Umsetzung Das Straßennetz wird innerhalb eines zweidimensionalen Rechteckes generiert, das vom Frontend gesetzt werden kann.

Anforderung 6 Gebäude werden durch die „Shape-Grammar“ dargestellt. (Priorität: niedrig)

Umsetzung Gebäude werden mittels einer Form dargestellt, die aus ein in die Tiefe extudiertes Shape (eine Form) zustande kommt. Die Form wird an der Bodenplatte ausgerichtet. Mehrere Stockwerke werden durch mehrere untereinander liegende Shapes ausgedrückt. Der Umriss der Shapes wird jeweils aus einer Verschmelzung zwischen einem neuen Vieleck und dem jeweils darüberliegenden Shape erzeugt. So können mehrere Stockwerke und verschiedene Teilformen nach dem Extrusionsverfahren (siehe 3.3.3 auf Seite 37) erzeugt werden.

Anmerkung Gebäude werden vom Backend nur aus einem Stockwerk und einem Dach erzeugt, da der für Verschmelzung zuständige Algorithmus zeitlich bedingt nicht für alle optischen Sonderfälle fertiggestellt werden konnte.

6.3 Vielfältigkeit

Die Vielfältigkeit des Backends zeigt sich besonders im Benutzerinterface (siehe Abbildung 6.3.1 auf der nächsten Seite) des Frontends, in dem Vieles konfiguriert werden kann.

- Die Gebäudegenerierung ist abschaltbar, so dass sich die Erzeugung auf die Straßen beschränkt. Dies eignet sich für andere Anwendungszwecke wie z.B. Pflanzen.
- Wechsel des verwendeten Regelsatzes zur Laufzeit, beispielsweise beim Wechsel des Spiellevels.
- Beibehaltung des zufällig gewählten Seeds der aktuellen Generierung, um Ergebnisse reproduzieren zu können.
- Objekte können von innen betrachtet werden, indem durch das Ein- und Ausschalten des Backfacecullings von der Kamera abgewandte Objekte sichtbar werden.
- Ein-/Ausschalten der Subdivision ermöglicht es, Parzellen ungeteilt zu erhalten.
- Kopplung des Spielers an die Weltgrenze: Beim Durchqueren der Welt wird die Stadt automatisch am Horizont erweitert.
- Ein-/Ausschalten von rechenintensiven Grafikeffekten wie Licht und Schatten.
- Änderung der Spielergeschwindigkeit beeinflusst das Generierungsintervall.
- Die Welt kann in erster und dritter Person durchschritten und untersucht werden.
- Die Möglichkeit, die Welt iterativ Schritt für Schritt zu erzeugen, veranschaulicht die Logik des Regelsatzes.
- Das Zurücksetzen der Welt und/oder der Spielfigur erlaubt es, das Backend mit verschiedenen Szenarien zu testen.
- Zugriff auf spezielle Entwicklungsfunktionen der Schnittstelle wie z.B. Ausgabe der Graphenstruktur.

Das Interface basiert auf der Interfaceklasse OgreBites der Grafikengine OGRE und wurde auf die Bedürfnisse des Frontends angepasst.

6.4 Statistik

Ein Überblick über den Umfang des Systems soll über dessen Quellcodemenge (LOC[2]) geschaffen werden. Dies ist kein Kriterium für Qualität, kann jedoch einen groben Eindruck über den Aufwand der Entwicklung von Front- und Backend verschaffen.

[2]LOC: Lines of code

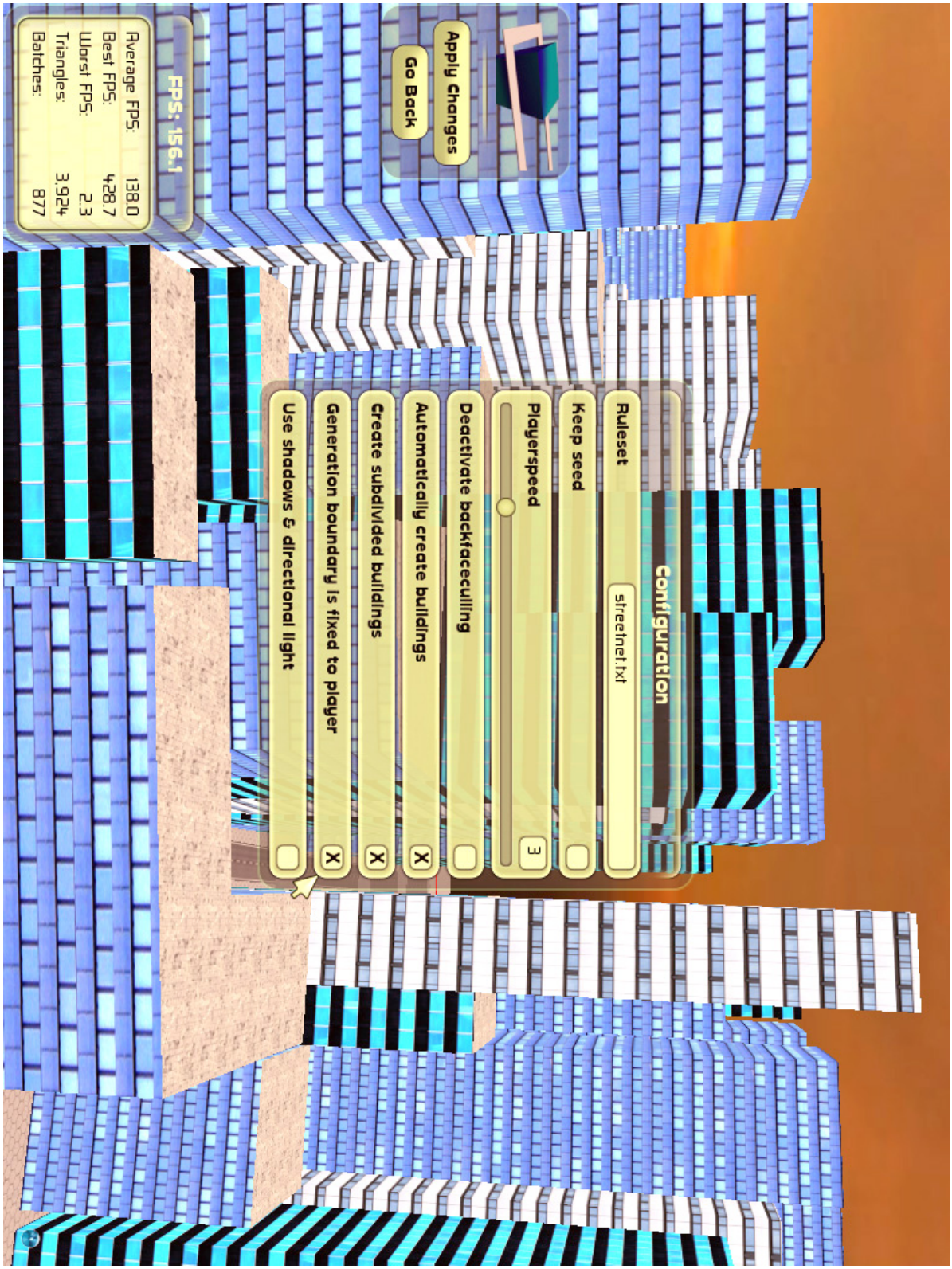


Abb. 6.3.1: Screenshot des Benutzerinterfaces

	LOC		LOC		LOC
Geometrie	8.900	OGRE I/O	3.900	Backend	28.200
Bibliotheken	9.800	Frontend	8.600	Frontend	12.500
Backend	9.500				
Total (Backend)	28.200	Total (Frontend)	12.500	Total	40.700

Tabelle 6.4.1: Umfang des Systems in LOC

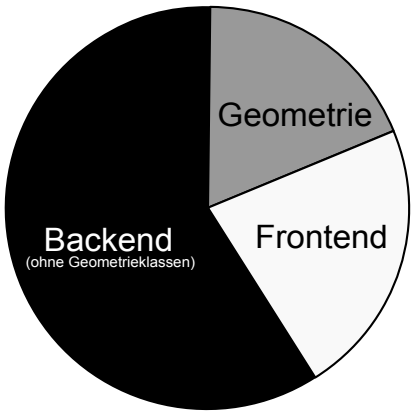


Abb. 6.4.1: LOC der Systemteile. Backend ohne Geometrieklassen (schwarz), Geometrieklassen (grau) und Frontend (weiß).

Es fällt auf, dass eine sehr umfangreiche Codebasis im Backend geschaffen werden musste. Die Ursache dafür wird im Ausblick erläutert. Erste sichtbare Ergebnisse konnten erst, nachdem vom Umfang das halbe Backend implentiert worden ist, geliefert werden. Die Entwicklung von PCG-Verfahren bringt somit eine anfangs ungewisse Entwicklungsphase mit sich, kann nach Abschluss dieser jedoch mit erstaunlichen Ergebnissen aufwarten.

Generatorgeschwindigkeit

Mit dem Generator sind einige Dauertests durchgeführt worden, um die Geschwindigkeit zur Berechnung der Städte zu errechnen. Hierfür ist folgendes Testsystem eingesetzt worden:

CPU Intel Core 2 Quad Q9650 (4 × 3,0 Ghz)

Arbeitsspeicher 4 GB RAM

Grafikkarte NVIDIA GeForce 8800 GTX (768 MB), Treiberversion 197.45

Betriebssystem Windows 7 Professional x64

Das Programm arbeitet mit zwei Threads und somit nur mit zwei der vier Kerne. Für den Test ist ein Feld berechnet worden, das in etwa 7x7 Straßenblöcken entspricht. Bei einer Straßenblocklänge von ca. 200 Metern entspricht dies einer Fläche von $1,4\,km^2$. Zur Bereithaltung der Daten dieses Bereichs in abstrakter Form und Geometrieform werden ca. 10 MB Arbeitsspeicher benötigt.

2D-Stadt	Dauer in ms	Einheiten	Polygone
Initialisierung	1500		
Straßennetz	390	294	
Gebäude	380	322	
Total (netto)	770	616	13.750

Tabelle 6.4.2: Geschwindigkeit des Systems für 2D-Städte

Die Erzeugung von dreidimensionalen Städten liefert sehr ähnliche Ergebnisse, die von der Anzahl der gewählten Höhenebenen abhängen. Die Dauer ist somit linear zu den normalen auf einer Ebene liegenden Städten. Festzustellen ist, dass die Gebäudegenerierung nahezu genauso viel Zeit benötigt, wie die Straßennetzerstellung. Durch geeignete Optimierungen kann die Dauer dieses Teilprozesses jedoch auf einen Bruchteil reduziert werden.

Wird die Welt bei Erreichen der Weltgrenze nach dem Passieren von ca. $1\frac{1}{2}$ Blöcken im Radius von ca. $3\frac{1}{2}$ Baublöcken um den Spieler herum ergänzt, werden im Schnitt die Werte aus Tabelle 6.4.3 erreicht.

	Dauer in ms	Einheiten
Straßennetz	510	67
Gebäude	165	95
Total (netto)	675	162

Tabelle 6.4.3: Geschwindigkeit bei Veränderung der bestehenden Welt

Die benötigte Rechenzeit beträgt durch das Multithreading nur ca. 165 ms, da die Straßengenerierung im Hintergrund ausgeführt wird. So entsteht während des Betriebs nur ein leichtes Ruckeln bei der Anwendung der Änderungen. Wie bereits erwähnt, lässt sich diese Zeit jedoch reduzieren. Durch Verlagerung auf einen weiteren Thread, der nur für die Geometrieverarbeitung des Frontends zuständig ist, kann das Ruckeln vollständig unterdrückt werden.

Kapitel 7

7 Ausblick

Die prozedurale Inhaltserzeugung bietet ein weites Spektrum an Forschungsmöglichkeiten, das je nach gewähltem Szenario völlig verschiedene Gebiete umfasst. Der Realismusgrad der Ergebnisse steht in direkter Verbindung zum Detailreichtum der umgesetzten PCG-Verfahren.

Die Möglichkeit, Gebiete aus dem Straßennetz zu entfernen und anderweitig neu zu generieren, könnte genutzt werden, um in Echtzeit spezielle Objekte (Flughafen, Fußballstadion, Ziel, etc.) in die Welt einzufügen, die einen Zweck für das Spiel bzw. die Anwendung erfüllen. Dies müsste in für den/die Anwender nicht sichtbaren Gebieten geschehen.

Die generierten Städte könnten ein realistischeres Erscheinungsbild erlangen, wenn die Gebäudehöhen durch zufällig gesetzte Ballungsgebiete beeinflusst werden würden. Dies würde einen Ersatz für Dichtekarten mit fester Größe darstellen, die Besiedlungsdichten angeben. Denn eine Karte festen Ausmaßes ist bei einer unendlichen Stadt nicht brauchbar.

Eine weitere Realitätssteigerung könnte erreicht werden, indem nicht mehr mit zufällig bestimmten unbebaubaren Bereichen gearbeitet werden würde, sondern diese gezielt gewählt werden würden. So könnte eine prozentuale Flächenaufteilung von Land, Wasser und Grünflächen nach dem in der Realität vorherrschenden Verhältnis geschehen.

Optik

Um ein noch futuristischer wirkendes Bild zu erlangen, wären weitere optische Veränderungen im Frontend denkbar. Es könnten absichtlich Hochhäuser eingefügt werden, die in das Straßennetz hereinragen und durch Portale überwunden werden könnten. Portale könnten zwischen Straßenenden im Netz Verbindungen schaffen.

Die beweglichen Plattformen, mit denen ein Wechsel zwischen den Höhenebenen erreicht wird, könnten durch andere optische Übergänge wie Straßenspiralen und -loopings realisiert werden. Straßenabzweigungen könnten blockiert werden, so dass an manchen Abzweigungen immer nur eine, aber wechselnde Abbiegemöglichkeit besteht. Weiterhin könnte das Straßennetz benutzt werden, um an den Unterseiten der Straßen zu fahren.

Spezielle Elemente, wie sich drehende Plattformen könnten den Wechsel zwischen der Ober- und Unterseite einer Straße ermöglichen.

Andere Szenarien

Neben der prozeduralen Stadt, in der man sich auf vorgegebenen Straßen bewegt, wäre eine Optimierung für ein Jump'n'Run-Szenario denkbar. Gebäude könnten so generiert werden, dass die Höhen benachbarter Gebäude sich immer in einem überwindbaren Maß bewegen. So könnten Städte wie aus dem Spiel „Mirrows Edge“ prozedural generiert und für die Sportart Parkoure angepasst werden, so dass es immer einen Pfad über die Gebäude gibt, der zum Ziel führt. Gebäude könnten das Höhenkriterium auch durch Durchbrüche in den Gebäuden erfüllen, um somit kein zu gleichmäßiges Stadtbild zu erlangen.

Optimierungen

Das gezeigte prozedurale Verfahren kann durch eine Vielzahl von Techniken in der Geschwindigkeit, mit der Objekte dargestellt werden, optimiert werden. Gebäude und Straßen werden bereits nur in einem festen Bereich um den Betrachter herum erzeugt und an den Grenzen prozedural ergänzt, sobald er sich auf diese zubewegt. Neben dieser Geometrie- und Rechenzeiteinsparung können Verfahren wie Level of Detail oder Level of Quality (Döllner/Buchholz, 2005) eingesetzt werden. Diese Verfahren können den Detailgrad entfernterer Objekte reduzieren oder diese bereits während der Erstellung durch einfachere und schnellere Algorithmen mit minderer Qualität erzeugen. Neben von der Hardware umgesetzten Verfahren wie View Frustum Culling, können durch Out-of-Core-Verfahren sogar ganze Quadranten bzw. Oktanten ausgelagert oder entfernt werden, um bei Wiederkehr in einen solchen Bereich erneut geladen oder generiert zu werden.

Abhängig von der Leistung des verwendeten Systems könnte eine Wiederverwendung bereits erzeugter Geometrie geschehen. Ist das System zu langsam um neue Inhalte zu generieren, werden alte Inhalte von alter Stelle übernommen und kopiert. Das Gleiche könnte generell mit den meist gleich aussehenden Straßensegmenten geschehen, die durch Techniken wie Instancing performant dupliziert und mit anderer Transformation dargestellt werden könnten.

Fazit

In den letzten Jahrzehnten hat eine stetige Weiterentwicklung prozeduraler Generierungsmethoden begonnen. Im Bereich der photorealistischen Stadterzeugung hat Pascal Müller um das Jahr 2000 herum den Grundstein gelegt und hat bis heute einen respektablen Stand der Technik etabliert. Für spezielle Szenarien ist dennoch eine spezifische Anpassung nötig, wie es in dieser Arbeit am Szenario der futuristischen Großstädte gezeigt worden ist. Mit einem großen Entwicklerteam kann eine entsprechend komplexe Aufgabenstellung durch den Einsatz prozeduraler Algorithmen ab einem speziellen Punkt (siehe Abbildung 7.0.1 auf der nächsten Seite) zu einem geringeren Aufwand führen. Es ist ab diesem Punkt festzustellen, dass ein Gewinn gegenüber der klassischen Methode

der Inhaltserzeugung erwirtschaftet werden kann. Durch hochauflösendes Material ist der Break-Even-Point zugunsten der prozeduralen Inhaltserzeugung nach vorne gerückt, wodurch in Zukunft vermutlich zunehmend auf generierte Inhalte gesetzt werden wird.

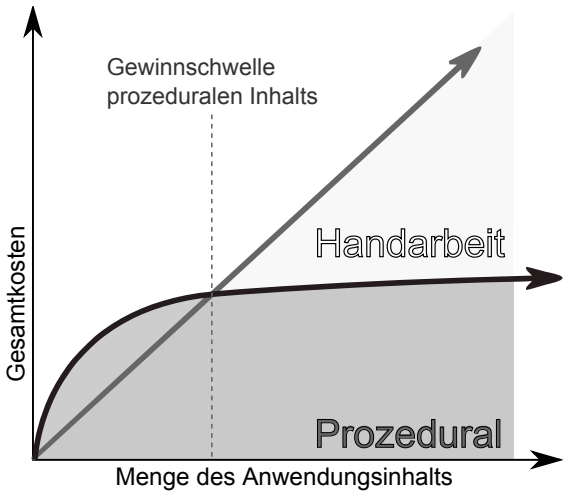


Abb. 7.0.1: Nutzen prozeduraler Inhalte nach „www.lostgarden.com“

Literaturverzeichnis

Agu, Emmanuel: CS 543: Computer Graphics - Lecture 8: 3D Clipping and Viewport Transformation. ⟨URL: `http://web.cs.wpi.edu/~emmanuel/courses/cs543/slides/lecture8.pdf`⟩ – Zugriff am 16. 08. 2010

Alexander, Christopher/Ishikawa, Sara/Silverstein, Murray: A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). Oxford University Press, 1977, ISBN 0195019199

Bresenham, J. E.: Algorithm for computer control of a digital plotter. IBM Syst. J. 4 1965 Nr. 1, 25–30, ISSN 0018–8670

Bruneton, Eric: Modeling and Rendering Rama. 2005 ⟨URL: `http://ebruneton.free.fr/rama3/rama.pdf`⟩ – Zugriff am 16. 08. 2010

Chen, Yanyun et al.: Modeling and rendering of realistic feathers. In SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM, 2002, ISBN 1–58113–521–1, 630–636

Delaunay, Boris N.: Sur la sphère vide. Nr. 6 Auflage. 1934, 793–800

Döllner, Jürgen/Buchholz, Henrik: Continuous level-of-detail modeling of buildings in 3D city models. In GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems. New York, NY, USA: ACM, 2005, ISBN 1–59593–146–5, 173–181

Du, D. z./Hwang, F.: Computing in Euclidean Geometry (Seri Pemikiran Bung Karno). World Scientific Publishing Company, 1992, ISBN 9810209665

Enix, Square: Final Fantasy XIII Scenario Ultimania. Square Enix, 2010 ⟨URL: `http://www.amazon.com/Final-Fantasy-XIII-Scenario-Ultimania/dp/B003NNEX7W%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3DB003NNEX7W`⟩, ISBN 4757527756

Fortune, S: A sweepline algorithm for Voronoi diagrams. In SCG '86: Proceedings of the second annual symposium on Computational geometry. New York, NY, USA: ACM, 1986, ISBN 0–89791–194–6, 313–322

Gansner, Emden et al.: Open-Source-Software zur Visualisierung von Graphen im DOT-Format. ⟨URL: `http://www.graphviz.org/`⟩ – Zugriff am 24. 08. 2010

Greuter, Stefan et al.: Real-time procedural generation of 'pseudo infinite' cities. In GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia. New York, NY, USA: ACM, 2003, ISBN 1–58113–578–5, 87–ff

Haines, Eric: Graphics Gems IV (IBM Version) (Graphics Gems - IBM) (No. 4). Morgan Kaufmann, 1994 ⟨URL: `http://erich.realtimerendering.com/ptinpoly/`⟩, ISBN 0123361559

Heineberg, Heinz: Grundriss Allgemeine Geographie: Stadtgeographie, 3. Auflage. UTB, Stuttgart, 2006, ISBN 3825221660

Henricsson, Olof/Streilein, André/Gruen, Armin: Automated 3-D Reconstruction of Buildings and Visualization of City Models. In In Proceedings of the Workshop on 3D City Models. 1996

Jacobs, Allan B.: Great Streets. The MIT Press, 1995, ISBN 0262600234

Johnstone, Brian et al.: Object-Oriented Graphics Rendering Engine (OGRE) 1.7.1 - Cthugha. ⟨URL: `http://www.ogre3d.org/`⟩ – Zugriff am 20. 08. 2010

Kazemi, Leyla et al.: Optimal traversal planning in road networks with navigational constraints. In GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems. New York, NY, USA: ACM, 2007, ISBN 978–1–59593–914–2, 1–8

Kinder, Prof. Dr. Sebastian: Standortsysteme von Dienstleistungen. 2009 ⟨URL: `http://www.geographie.uni-tuebingen.de/fileadmin/arbeitsbereiche/AG_KINDER/Lehrveranstaltungen/WS0809/8._Vorlesung.pdf`⟩ – Zugriff am 13. 09. 2010

Lechner, Thomas/Watson, Ben/Wilensky, Uri: Procedural city modeling. In In 1st Midwestern Graphics Conference. 2003

Lehner, Thomas: Digitale Geländemodellierung mittels Delaunay-Triangulierung und Abbauplanung in AutoCAD. Diplomarbeit, Johannes Kepler Universität Linz, 2002, ⟨URL: `http://www2.gup.jku.at/~gk/Diplom/AbbauplanungACAD.PDF`⟩

Lipp, Markus: Interactive Computer Generated Architecture. Diplomarbeit, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2007, ⟨URL: `http://www.cg.tuwien.ac.at/research/publications/2007/LIPP-2007-CGA/`⟩

Lipp, Markus/Wonka, Peter/Wimmer, Michael: Parallel Generation of L-Systems. In **Marcus Magnor, Bodo Rosenhahn, Holger Theisel (Hrsg.):** Vision, Modeling, and Visualization Workshop 2009 (VMV). nov 2009 ⟨URL: `http://www.cg.tuwien.ac.at/research/publications/2009/LIPP-2009-PGL/`⟩, ISBN 978–3980487481

Lisberger, Steven/MacBird, Bonnie: Tron (DVD) - 92 min. USA: Disney-Studios, 1982

Müller, Pascal: Pascal Mueller's Wiki. ⟨URL: `http://www.vision.ee.ethz.ch/~pmueller/wiki/CityEngine/Front`⟩ – Zugriff am 15. 04. 2010

Müller, Pascal: Procedural Inc. - 3D Modeling Software for Urban Environments. ⟨URL: `http://www.procedural.com/`⟩ – Zugriff am 15. 04. 2010

Müller, Pascal: Prozedurales Modelieren einer Stadt. Semester Thesis ETH Zürich, 1999, ⟨URL: `http://www.vision.ee.ethz.ch/~pmueller/documents/eth_semester_thesis_1999/prozedurales_modelieren_einer_stadt__pmueller_sa99.pdf`⟩

Müller, Pascal: Design und Implementation einer Preprocessing Pipeline zur Visualisierung prozedural erzeugter Stadtmodelle. MSc Thesis ETH Zürich, 2001, ⟨URL: `http://www.vision.ee.ethz.ch/~pmueller/documents/master_thesis_pascal_mueller.pdf`⟩

Müller, Pascal: Procedural modeling of cities. In SIGGRAPH '06: ACM SIGGRAPH 2006 Courses. New York, NY, USA: ACM, 2006, ISBN 1–59593–364–6, 139–184

Müller, Pascal et al.: Interactive procedural street modeling. In SIGGRAPH '07: ACM SIGGRAPH 2007 sketches. New York, NY, USA: ACM, 2007, 35

Müller, Pascal et al.: Procedural modeling of buildings. In SIGGRAPH '06: ACM SIGGRAPH 2006 Papers. New York, NY, USA: ACM, 2006, ISBN 1–59593–364–6, 614–623

O'Rourke, Joseph: Computational Geometry in C. New York, NY, USA: Cambridge University Press, 1998, ISBN 0521640105

Pascal Müller, Yoav I. H. Parish: Procedural modeling of cities. In SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM, 2001, ISBN 1–58113–374–X, 301–308

Prusinkiewicz, Przemyslaw/Lindenmayer, Aristid: The Algorithmic Beauty of Plants (Virtual Laboratory). Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 1990, ISBN 3540972978

Rudzicz, Verbrugge: An iterated subdivision algorithm for procedural road plan generation. In An iterated subdivision algorithm for procedural road plan generation. School of Computer Science, McGill University, 2008

Schrader, Robert: Prozedural unterstützte Generierung von Gebäudemodellen für interaktive Anwendungen. Master Thesis Universität Koblenz-Landau, Campus Koblenz, 2007

Shewchuk, Jonathan Richard: Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. ⟨URL: `http://www.cs.cmu.edu/~quake/triangle.html`⟩ – Zugriff am 16. 08. 2010

Stiny, G.: Introduction to shape and shape grammars. Environment and Planning B, 7 1980 Nr. 3, 343–351

Sun, Jing et al.: Template-based generation of road networks for virtual city modeling. In VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology. New York, NY, USA: ACM, 2002, ISBN 1–58113–530–0, 33–40

Sun, Min/Chen, Jun: Data structure research of 3D city road network. In ISPRS Archives - Volume XXXIII, Part B4(1-3). Institute of Surveying and Mapping July 2000, 1027ff

Sutherland, Ivan E./Hodgman, Gary W.: Reentrant polygon clipping. Commun. ACM, 17 1974 Nr. 1, 32–42, ISSN 0001–0782

Tarjan, Robert Endre/Aspvall, Bengt/Plass, Michael F.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters, 8 1979 Nr. 3, 121 – 123 ⟨URL: `http://www.sciencedirect.com/science/article/B6V0F-45FCF92-2/2/5309e3f9271fa947b1af0f7ea9922c32`⟩, ISSN 0020–0190

Tiernan, James C.: An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM, 13 1970 Nr. 12, 722–726, ISSN 0001–0782

University of Manchester: The University of Manchester General Polygon Clipper library. ⟨URL: `www.cs.man.ac.uk/~toby/alan/software`⟩ – Zugriff am 16. 08. 2010

Watson, Benjamin: Modeling land use with urban simulation. In SIGGRAPH '06: ACM SIGGRAPH 2006 Courses. New York, NY, USA: ACM, 2006, ISBN 1–59593–364–6, 185–251

Weiler, Kevin/Atherton, Peter: Hidden surface removal using polygon area sorting. In SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM, 1977, 214–222

Wikipedia: Bresenham-Algorithmus — Wikipedia, Die freie Enzyklopädie. 2010 ⟨URL: `http://de.wikipedia.org/w/index.php?title=Bresenham-Algorithmus&oldid=76938981`⟩ – Zugriff am 27. 08. 2010

Wikipedia: Delaunay-Triangulation — Wikipedia, Die freie Enzyklopädie. 2010 ⟨URL: `http://de.wikipedia.org/w/index.php?title=Delaunay-Triangulation&oldid=76329385`⟩ – Zugriff am 27. 08. 2010

Wikipedia: Strongly connected component — Wikipedia, The Free Encyclopedia. 2010 ⟨URL: `http://en.wikipedia.org/w/index.php?title=Strongly_connected_component&oldid=374814214`⟩ – Zugriff am 27. 08. 2010

Wikipedia: Tarjan's strongly connected components algorithm — Wikipedia, The Free Encyclopedia. 2010 ⟨URL: `http://en.wikipedia.org/w/index.php?title=Tarjan%27s_strongly_connected_components_algorithm&oldid=380366092`⟩ – Zugriff am 27. 08. 2010

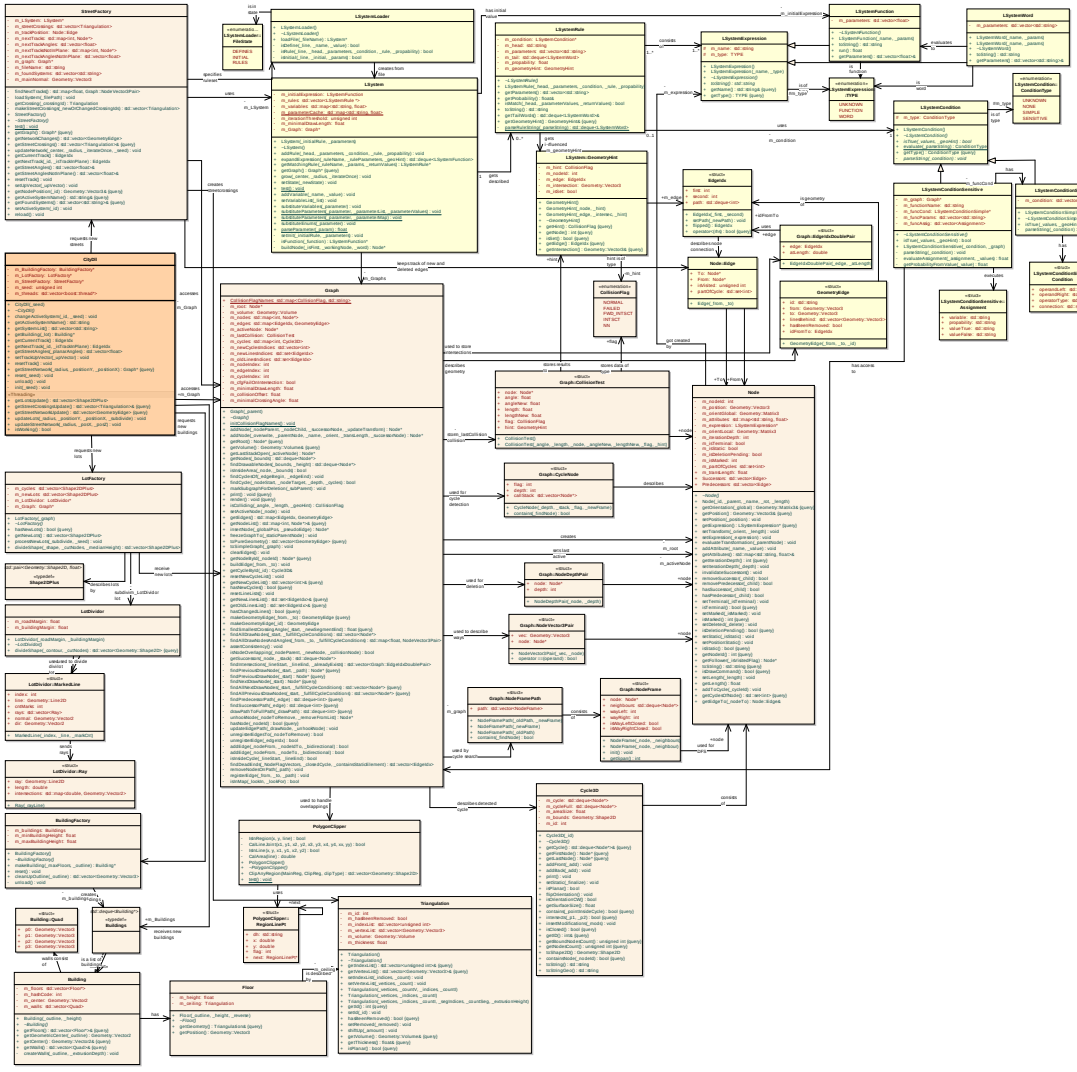
Wonka, Peter et al.: Instant architecture. ACM Trans. Graph. 22 2003 Nr. 3, 669–677, ISSN 0730–0301

Wulff-Nilsen, Christian: Minimum Cycle Basis and All-Pairs Min Cut of a Planar Graph in Subquadratic Time. CoRR abs/0912.1208 2009

Xiao, Jianxiong et al.: Image-based street-side city modeling. In SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers. New York, NY, USA: ACM, 2009, ISBN 978–1–60558–858–2, 1–12

Anhang

Klassendiagramm des Backends[1]



[1]Nur in PDF-Version lesbar.