# Prozedurale Generierung von Echtzeit-optimierten 3D-Stadtmodellen

Studiengang Medieninformatik der
Technischen Hochschule Mittelhessen

## Bachelorarbeit

vorgelegt von

**Oliver Kulas**
geb. in Meißen

30. Juni 2012

durchgeführt bei


| | |
|---|---|
| Referent der Arbeit: | Prof. Dr. Cornelius Malerczyk |
| Korreferent der Arbeit: | Dipl.-Ing. (FH) René Nold |
| Betreuer bei weltenbauer.: | Dipl.-Inf. (FH) Christian Rathemacher |


Fachbereiche
Informationstechnik-Elektrotechnik-Mechatronik
und
Mathematik-Naturwissenschaften-Datenverarbeitung

Wiesbaden, 2012

# Danksagung

Mehrere Personen haben zum Gelingen dieser Arbeit beigetragen, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte. Für die hervorragende Betreuung während der Bachelorarbeit und des Studiums an der Technischen Hochschule Mittelhessen möchte ich mich bei Prof. Dr. Cornelius Malerczyk bedanken. Für die Unterstützung, Betreuung und vor allem Geduld danke ich Dipl.-Inf. Christian Rathemacher und Dipl.-Ing. René Nold sowie allen anderen Kollegen bei weltenbauer. Ein besonderer Dank geht an Dipl.-Math. Sabine Langkamm, die mich wie kaum jemand anders für den Fachbereich der Grafischen Datenverarbeitung ermutigte.

Ich danke ganz besonders meinen Eltern, ohne die das Studium, diese Bachelorarbeit und vieles mehr nicht möglich gewesen wäre. Außerdem danke ich meiner Freundin Julia Immel für die Unterstützung, die vielen späten Stunden des Korrekturlesens, die Ermutigung während der Arbeit und für vieles darüber hinaus. Vielen Dank auch an alle Freunde, die stets hinter mir stehen.

# Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wiesbaden, Juni 2012

Oliver Kulas

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation

Virtuelle dreidimensionale Umgebungsdaten finden heutzutage nicht nur Verwendung bei CAD-Software, Stadtplanung oder Architekturvisualisierung, sondern darüber hinaus auch in Bereichen von naturwissenschaftlichen und ingenieurtechnischen, sowie in multimedialen und interaktiven Anwendungen. Von Systemen im Bereich allgemeiner *Virtual Environments*, sowie *Virtual* oder *Augmented Reality* Lösungen, über den Einsatz in Film und TV-Produktionen bis hin zu industriellen Simulationen und Computerspielen für Endverbraucher sind viele Einsatzgebiete möglich. 3D-Stadtmodelle sollen dabei unter anderem die sozialen, kulturellen und wirtschaftlichen Strukturen eines Stadtgebiets in einer räumlicher Umgebung visualisieren. Aufgrund der vielen verschiedenen fachlichen Anwendungsbereiche sollten die Modelle die Möglichkeit bieten vielseitig eingesetzt werden zu können.

**Abbildung 1.1**: Echtzeit-3D-Modell der Stadt Düsseldorf

Aufgrund der hohen Komplexität eines Stadtmodells und der Einzigartigkeit einzelner Städte untereinander kann ein manueller Produktionsaufwand, insbesondere für größere Städte oder viele verschiedene Stadtgebiete, unverhältnismäßig aufwändig sein. Durch individuelle automatische Generierung und Platzierung von einzelnen Objekten innerhalb einer Stadt, wie Gebäuden oder Straßen, kann die Produktionszeit erheblich gesenkt werden. Dies erfordert hohe Anforderungen an eine geschickte prozedurale Algorithmik.

Darüber hinaus stellen Echtzeit-Grafikanwendungen bei der Visualisierung von 3D-Daten eine große Herausforderung dar, da sie neben der extrem schnellen Darstellbarkeit zu einem der höchst interaktiven Bereiche der Computergrafik gehören, da der weitere Prozess zur Laufzeit des Programms nicht feststeht. Ein Betrachter agiert bzw. reagiert auf die gerenderten Bilder und entscheidet somit was als Nächstes passiert. Er sieht also nicht nur eine Abfolge aneinander gereihter bewegter Bilder, sondern taucht in einen dynamischen Prozess ein [AMHH08]. Vorausgesetzt die Hardware ist in der Lage die Bilder mit einer Bildwiederholfrequenz von mindestens 25 Hz zu rendern. Ab dieser Wiederholfrequenz erscheint einem Betrachter die Darstellung bewegter Bilder uneingeschränkt flüssig.

Neben sämtlichen technischen Voraussetzungen oder den effektiven Produktionsabläufen steht das visuselle Resultat. Eine schwierige Aufgabe ist die Visualisierung real existierender Städte und das sie auch als solche erkennbar sind. Der Wiedererkennungswert einer Stadt wird nicht durch die Masse der Gebäude geprägt, sondern vielmehr durch eine markante Skyline (vgl. Abb 1.2).
Dazu müssen hauptsächlich Terrain, Stadtbild, Wahrzeichen und Infrastruktur möglichst real nachempfunden werden. Die Anforderung dabei einen hohen und möglichst realitätsnahen Detailgrad darzustellen steht der Begrenzung durch die Rechenleistung aktueller Hardware gegenüber.

**Abbildung 1.2**: Silhouette der Berliner Skyline[1]

[1] http://commondatastorage.googleapis.com/static.panoramio.com/photos/original/20481960.jpg

## 1.2 Problemstellung

Bei der Erzeugung und Darstellung von 3D-Stadtmodellen für den Einsatz in Echtzeitanwendungen stehen sich Anforderungen wie eine detaillierte und realitätsnahe Darstellung, eine performante Rechenleistung der eingesetzten Hardware und eine adäquate Produktionszeit bei der Erstellung gegenüber.

Um einen hohen Detailsgrad von 3D-Modellen zu erreichen sind zum einen viele Polygon-Unterteilungen im *Mesh* des Modelles und zum anderen hochauflösende Texturen erforderlich. Je öfter ein *Mesh* unterteilt wird, desto mehr Details können modelliert werden. Gleiches gilt bei Texturen. Je größer die Texturauflösung, desto höher ist der auf ihr abgebildete Detailgrad. Das Rendern eines für Realtimeverhältnisse detailierten Modells in Verbindung mit einer großen Textur stellt im Einzelfall kein größeres Problem an die Hardware dar. Sollen aber tausende solcher Modelle gleichzeitig und in Echtzeit gerendert werden sind die Leistungsgrenzen, wie Rechenleistung der *GPU* und Texturspeicher, aktueller Grafikhardware schnell erreicht. Darüber hinaus ist die manuelle Erstellung einer 3D-Stadt sehr aufwändig, da sehr viele Objekte, wie Gebäude und Straßen, erstellt und gemäß Straßenverlauf oder Stadtgebiet innerhalb einer Stadt platziert werden müssen. Ein Automatismus zur Erzeugung und Platzierung dieser Objekte kann wertvolle Produktionszeit einsparen. Bei Nachempfindung realer Vorbilder stellen die Einzigartigkeit einzelner Städte, die Variation einzelner Häuser, Gebiete oder Wahrzeichen weitere Schwierigkeiten dar. Um dem Erreichen der Leistungsgrenzen der Grafikhardware entgegen zu wirken und den Produktionsablauf zu optimieren ist ein Konzept zur prozeduralen Erstellung von Echtzeit-optimierten 3D-Stadtmodellen erforderlich.

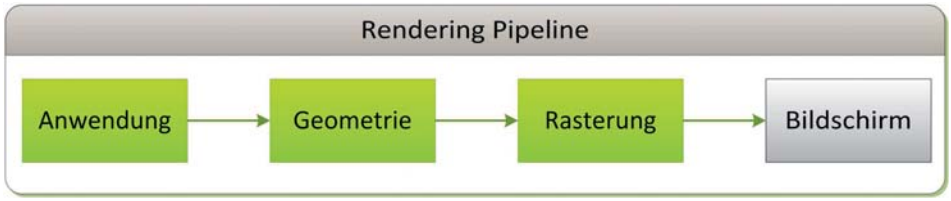


**Abbildung 1.3**: Vereinfachte Rendering Pipeline. Angelehnt an [AMHH08].

Die Grundlage heutiger Echtzeit-Computergrafik bildet die *Rendering Pipeline*. Sie beschreibt den Weg zur Darstellung der virtuellen Szene auf den Bildschirm. Grafik-APIs, wie die plattformunabhängige Spezifikation *OpenGL* oder die *Microsoft Windows*-basierende Programmierschnittstelle *Direct3D*, werden dabei üblicherweise zur Ansteuerung der Grafikhardware verwendet. Die Leistungsfähigkeit von Grafikhardware, speziell Grafikprozessoren, kann aufgrund des rasanten technologischen Fortschritts zwar stets verbessert werden, jedoch entstehen die meisten Performanceengpässe innerhalb der

softwareseitigen Pipeline-Verarbeitung unabhängig von Grafikhardware, sodass die Optimierung der einzelnen Arbeitsschritte innerhalb der Grafik-Pipeline eine zentrale Rolle in der Entwicklung der Echtzeitgrafik spielt [Bie08].

Für die Erstellung komplexer, Echtzeit-optimierter 3D-Stadtmodelle gilt es also das optimale Gleichgewicht zwischen realitätsnaher Darstellung der Modelle und der erforderlichen Echtzeit-Performance zu erreichen. Da virtuelle Städte oft zur visuellen Unterstützung einer Gesamtszenerie dienen muss sichergestellt werden, dass genügend Echtzeit-Performance zur Berechnung der restlichen Szenerie und Objekte zur Verfügung steht. So stellt nicht nur die begrenzte Anzahl der *Draw Calls* an die *GPU* ein Problem dar, sondern darüber hinaus viele weitere Faktoren, wie die verfügbare Größe des Texturspeichers oder die maximale zulässige Menge von *Vertices* aller Objekte innerhalb der eingesetzten *3D-Engine*.

Zur Maximierung der Echtzeit-Perfomance auf aktueller Standard-Hardware kommen verschiedene Methoden in Frage, die evaluiert und ggf. umgesetzt werden. Für die Darstellung von Objekten in verschiedenen Entfernungen zur Kamera empfiehlt sich ein *Level of Detail*-System, welches erlaubt die Komplexität der Modelle im Verhältnis zur Kameraentfernung zu regulieren. Mit Hilfe des *Backface Culling* kann das Rendern der Kamera abgewandten Flächen eingespart werden. Das *View Frustum Culling* kann die Berechnung von Objekten außerhalb des Kamerasichtbereichs vermeiden. Mit *Clipping* kann unter Umständen auch Leistung eingespart werden, indem irrelevante Teilflächen für die Berechnung des Bildes außen vor gelassen werden. Des weiteren kann mittels *Occlusion Culling* verhindert werden, dass Objekte, die von anderen verdeckt werden, gerendert werden. Zur Minimierung des notwendigen Texturspeichers empfiehlt sich der Einsatz spezieller *Shader*, mit denen es möglich ist einzelne 8 Bit Farbkanäle innerhalb einer Textur mit verschiedenen Texturinformationen auszuwerten. Zur Farbdarstellung können die *Vertex Colors* der Objekte ausgewertet werden und mit dem jeweiligen Texturfarbkanal verechnet werden. Die Anzahl der in der 3D-Szene verwendeten Objekte und Materialien muss so gering wie möglich gehalten werden, damit diese mit Hilfe des *Draw Call Batching* effektiv zusammengefasst werden können. Beispielsweise kann dabei mit der Zusammenfassung von Texturen in *Texture Atlases* die Anzahl der *Draw Calls* für Materialien minimiert werden. Unter Umständen können bzw. müssen einige dieser Maßnahmen zur Leistungsoptimierung bereits vor der Laufzeit der Echzeitanwendung durchzuführen. Dadurch kann bei der Echtzeitberechnung ebenfalls Rechenleistung eingespart werden.

## 1.3 Zielsetzung

Ziel dieser Arbeit ist es ein Konzept zu entwickeln mit dem 3D-Stadtmodelle prozedural erstellt werden können, welche sowohl die visuellen als auch die Anforderungen des *Realtime Renderings* erfüllen. Es soll einen effizienten Produktionsablauf zur Erzeugung und Platzierung der Stadtmodelle beschreiben. Die Modelle sollen einen möglichst

realitätsnahen Eindruck vermittlen und gleichzeitig gewährleisten, dass die notwendige Grafikleistung zur Verfügung steht. Dazu werden zunächst die infrage kommenden Methoden zur Performanceeinsparung evaluiert und die Leistungsstatistiken für die Optimierung analysiert. Innerhalb einer modular aufbauten Softwarelösung wird das Konzept prototypisch integriert, sodass es als Grundlage zur Erstellung von weiteren Echtzeit-optimierten 3D-Stadtmodellen dienen kann.


**Abbildung 1.4**: Schiffsimulator 2012 - Binnenschifffahrt

In erster Linie soll die Umsetzung des Konzepts, die Performanceanalyse und die darauffolgende Optimierung innerhalb der 3D-Entwicklungsumgebung *Unity3D Pro* in der Version 3.4 erfolgen. Um möglichst reale Einschätzungen zur Leistungsoptimierung innerhalb einer komplexen Echzeitanwendung treffen zu können, wird die Softwarelösung des Konzepts innerhalb des kommerziellen Computerspiels *Schiffsimulator 2012 - Binnenschifffahrt* getestet und umgesetzt. In diesem Simulationsspiel soll der Flusslauf des Rheins von Mainz bis zur niederländischen Grenze, sowie ein Teil des Mains von Frankfurt bis zur Rheinmündung, nachgebildet werden. Auf einer Strecke von über 300 Kilometern sollen sämtliche Großstädte, unter anderem Koblenz, Köln, Düsseldorf und Duisburg, darstellt werden. Das Spiel sieht außerdem einen realen Tag/Nacht-Modus vor, welcher für die Städte berücksichtigt werden muss.

Zur Erzeugung der Stadtmodelle werden frei verfügbare Geodaten und reales Kartenmatieral verwendet. Ebenso werden Wahrzeichen, Brücken, Infrastruktur oder Häfen anhand solcher Daten erstellt und platziert. Weiterhin werden architektonische oder geografische Gegebenheiten mit Bildern verglichen um die Städte an die Realität anzupassen und den Erkennungswert zu erhöhen. Im Vordergrund der Arbeit steht die konzeptionelle und technische Umsetzung der prozeduralen Stadtgenerierung. Deshalb und aufgrund der ausreichend gegebenen Materialien wird zunächst auf eine Evaluation der Realtätsnähe bzw. der allgemeinen Optik durch Anwender verzichtet.

## 1.4 Organisation dieser Arbeit

Nachdem in diesem Kapitel eine kurze Einführung in die Thematik erfolgt werden im Kapitel 2 Konzepte und Lösungsansätze für die prozedurale Stadtgenerierung erläutert. Weiterhin werden in diesem Kapitel kommerzielle Lösungen und Projekte beschrieben, die alle das Ziel der automatischen Erzeugung von 3D-Inhalt verfolgen. Die Verwendung neuer Technologien für die Darstellung von 3D-Inhalten in Internet-, Browser- und mobilen Applikationen werden aufgezeigt.
Im Kapitel 3 werden Grundlagen der Echtzeitgrafikverarbeitung beschrieben. Das Kapitel stellt das Basiswissen für die weiteren Kapitel dar. Die heutige Verarbeitung von Daten durch Grafikhardware und die Einflussnahme durch Programmierung der Hardware wird beschrieben. Erfahrene Leser können dieses Kapitel überspringen.

Anschließend beginnt der Hauptteil der Arbeit. Zunächst wird im Kapitel 4 eine Konzeptentwicklung für eine Softwarelösung zur automatisierten Erzeugung einer echtzeitfähigen Stadt beschrieben, indem die Anforderungen an die Software definiert und anschließend Optimierungsmethoden erläutert werden.
Im Kapitel 5 erfolgt die Realisierung des zuvor entwickelten Konzepts. Nach dem eine Vorgehensweise ausgewählt wird, werden notwendige vorbereitende Maßnahmen vorgenommen. Schließlich wird die Softwareimplementation durchgeführt und ausführlich beschrieben.
Im Kapitel 6 werden die Ergebnisse aus dem Konzepts aufzeigt. Anhand einer Performanceanalyse wird die Echtzeitfähigkeit der Stadtmodelle, die mit der prototypischen Software erstellt werden, bewertet.

Das letzte Kapitel stellt die Zusammenfassung der Arbeit dar und gibt einen Ausblick auf weiterführende Schritte.

## 1.5 Zusammenfassung der Ergebnisse

Bei der 3D-Visualisierung von realen Szenarien innerhalb von Echtzeitanwendungen, wie Simulationen oder Computerspielen, ist häufig die Darstellung realitätsnaher und detailgetreuer Stadtgebiete erforderlich. Die Erstellung einer real wirkenden virtuellen Stadt kann sehr aufwändig sein. Ein authentischer Eindruck kann dabei nur durch ausreichend variable Gebäude und flexibel kombinierbare Gebäudeteile erreicht werden, beispielsweise durch verschiedene Formen und Größen der Häuser und Dächer, unterschiedliche Fassadenfarben, Anzahl der Stockwerke oder Platzierung einzelner Gebäudeteile.

Auf Grundlage existierender Geodaten ist es möglich Stadtmodelle anhand realer Vorgaben in einer virtuellen Welt nachzubilden. So kann beispielsweise anhand eines Straßennetzwerks, Flusslaufs oder Schienennetzes die Generierung und Platzierung von Gebäuden und Anlagen inmitten der 3D-Umgebung vorgenommen werden. Die Kom-

plexität der Stadtmodelle ist dabei begrenzt durch die Rechenleistung aktueller PC-Systeme. Um mit diesen Systemen bessere Ergebnisse zu erzielen, ist daher die Optimierung der 3D-Modelle, sowie deren Texturen, unabdingbar.

Ziel dieser Bachelorarbeit ist die Entwicklung eines Konzepts, um Gebäude auf Basis vorhandener Einzelteile prozedural zu erzeugen und entsprechend auf einem Terrain in einer 3D-Echtzeit-Umgebung zu platzieren. Das Konzept wird prototypisch als Teil in einer modular aufgebauten Softwarelösung umgesetzt und evaluiert. Um zu gewährleisten, dass bei möglichst detaillierter Darstellung der Städte die notwendige Echtzeit-Performance zur Verfügung steht, werden verschiedene Methoden zur Optimierung der Grafikleistung bewertet und die optimale Lösung implementiert. Die Echtzeitfähigkeit des in *Unity3D* prototypisch implementierten Konzepts wird anhand von objektiv messbaren Parametern bewertet. Bei der Konzeptentwicklung wird für unterschiedliche Szenarien und Systemanforderungen die Zielsetzung der Echtzeitfähigkeit bei möglichst hoher Darstellungsqualität gewährleitet.

# Kapitel 2

# Stand der Technik

Im ersten Kapitel wurde eine kurze Einführung in die Thematik geliefert. Die Intention dieser Arbeit wurde erläutert und anhand der Problemstellungen erfolgte die Zielsetzung. In diesem Kapitel werden zeitgemäße Konzepte und Ideen zur automatisierten Erstellung von 3D-Stadtmodellen sowie kommerzielle Lösungen beschrieben. Im folgenden Kapitel werden die Grundlagen der Computergrafikverarbeitung veranschaulicht, wobei der Bezug speziell auf Echtzeitrendering ausgerichtet ist.

## 2.1 Einleitung

Mit dem stetigen Ansteig der Leistungsfähigkeit aktueller Hardwaresysteme werden immer häufiger 3D-Stadtmodelle in verschiedensten Bereichen der Echtzeitgrafikanwendungen eingesetzt. Dabei finden die Modelle in interaktiven Consumer-Anwendungen von 3D-Navigationssystemen über Film- und Fernsehproduktionen bis hin zu Computerspielen ihren Einsatz [Lie08]. Zum täglichen Wetterbericht gehört oft der 3D-Geländeflug, bei dem Städte visualisiert werden, um dem Zuschauer einen realistischeren Eindruck der Umgebung zu vermitteln [Nol05]. Die Industrie und Forschung kommt heutzutage ohne interaktive Simulationsumgebungen nicht aus. Die Entwicklung neuer Produkten oder Erforschung neuer Standards geschieht zunächst oft in virtuellen Umgebungen. Sind die Ergebnisse aus den virtuellen Tests erfolgreich geht die Entwicklung über die Simulationsumgebungen hinaus. So findet beispielsweise die Entwicklung moderner Fahrassistenzsysteme in einer virtuellen Echtzeitumgebung statt [Wag11].
Es exisiteren bereits viele unterschiedliche Lösungsansätze prozedurale Erzeugung von 3D-Städten. Oft handelt es sich dabei um Lösungen für Visualisierungen durch aufwändige Renderings. Selten können diese Modelle tatsächlich ohne Weiteres im Realtime-Bereich genutzt werden. Einige dieser Konzepte und Lösungen werden im Folgenden vorgestellt.

## 2.2 Konzepte zur prozeduralen Stadterstellung

So viel wie es denkbare Einsatzmöglichkeiten für prozedural erzeugten 3D-Inhalt gibt, so groß ist auch die Anzahl der existierenden Konzepte. Unterschiedlichste Ansätze verfolgen dennoch stets das selbe Ziel - die automatisierte Erzeugung von Objekten oder ganzer virtueller Welten. Oft handelt es sich bei den Umgebungen um Städte. Hier stehen sehr viele Objekte auf kleinstem Raum. Die manuelle Erstellung der einzelnen Modelle kann innerhalb eines angemessen Zeitrahmens nicht erfolgen. Darüber hinaus ist bei der Anordnung der Bauwerke und Pflanzen in diesen Bereichen meist eine Struktur durch Straßen und Wege erkennbar. Eine ideale Voraussetzung zur automatisierten Erstellung solcher Gebiete.

### 2.2.1 Photogrammetrie

Das Gebiet der Photogrammetrie beschäftigt sich mit der Auswertung von Bild- und Messdaten. Es liegt nah zweidimensionale Karten-, Luftbild- oder Satellitendaten als Basis zur Erzeugung von 3D-Objekten zu verwenden. Eine Gruppe japanischer Wissenschaftler entwickelte im Auftrag des Ministeriums für Wirtschaft, Handel und Industrie Japan ein System um solche Bilddaten effektiv für die automatische Stadtgenerierung zu nutzen [TNAK03]. Neben den Satelliten- und Luftbildern werden durch einen Laser gescannte Höhenprofildaten des Geländes und der Gebäude geliefert. Die Daten sind als Punktwolke vorhanden und besitzen eine Genauigkeit von 15 cm in der Höhe und 1 m in der Breite. Zusätzlich werden Luftbilder und vektorbasierte Kartendaten zur Generierung herangezogen (vgl. Abb. 2.1).

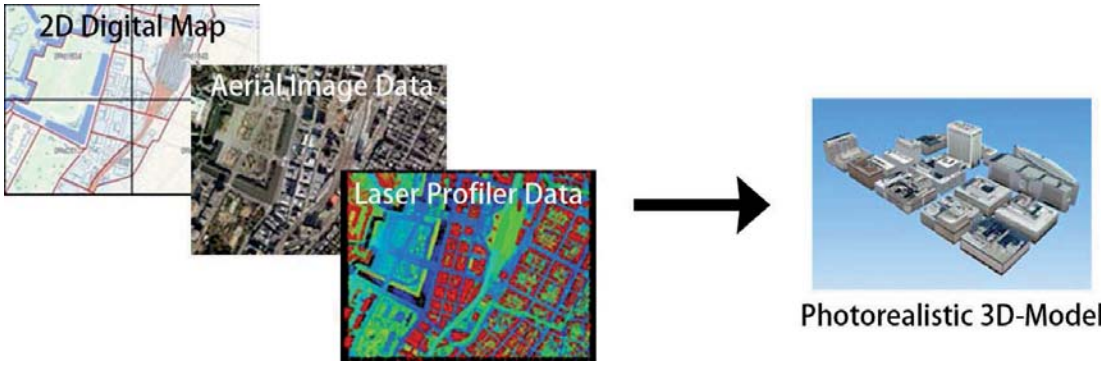


**Abbildung 2.1**: Automatisches System zur 3D-Modell-Erzeugung auf Basis von Karten-, Luftbild- und Laserprofildaten. Quelle: [TNAK03].

Das entwickelte Tool bietet die Erstellung der Gebäudemodelle und deren nachträgliche Bearbeitung, das automatisierte Modellieren des Terrains anhand der Daten sowie die Erzeugung von Straßen, Schienen und Brücken. Lediglich das Texturieren ist bei dieser Umsetzung nicht prozedural möglich, da die vorliegenden Daten einizg Bilder aus der Vertikalen darstellen. Texturen für die Modell müssen zusätzlich vorhanden sein. Mo-

delle markanter Bauwerke und Wahrzeichen werden aus einer exisitierenden Datenbank entnommen. Mit Hilfe dieser Anwendung konnten sämtliche größeren Städte Japans innerhalb von ca. zwei Jahren als 3D-Modell nachgebildet werden (vgl. Abb. 2.2). Die 3D-Stadtmodelle werden unter dem Namen *MapCube* veröffentlicht [TNAK03].

**Abbildung 2.2**: Prozedural erstelles 3D-Modell der Stadt Tokio. Links: Original-Foto, rechts: *MapCube* 3D-Stadtmodell. Quelle: [TNAK03].

### 2.2.2 Digital Surface Model und Digital Terrain Model

Ähnliche Ansätze durch Auswertung von Bilddaten zur automatischen Gebäude- und Geländeerstellung mit Hilfe der Photogrammetrie verspechen Verfahren auf Basis von *Digital Surface Model (DSM)* bzw. *Digital Terrain Model (DTM)*. Das DSM repräsentiert dabei die Erdoberfläche mit allen auf ihr befindlichen Objekten, wie Gebäude, Straßen, Gewässer oder Vegetation. Das DTM bildet dagegen die natürliche unbebaute Erdoberfläche ab. Zwischen den Modellen bestehen meist geometrische Differenzen. So besteht ein Terrainmodell überwiegend aus weichen, glatten Formen mit wenigen Kanten, die natürlich und unstrukturiert auf dem Gelände vorkommen. Modelle der Oberfläche mit sämtlichen Bebauungen erlauben dagegen anhand von harten Kanten und Strukturen Rückschlüsse auf Objekte, die sich auf der Erdoberfläche befinden [Wei]. Durch algorithmische Erkennung von Strukturen und Farbunterschieden unterstützen die orthogonalen Satellitenbilddaten dabei die Auswertung. Je mehr Informationen hinzukommen, desto genauer werden die Ergebnisse. Mit Hilfe von Kartendaten lassen sich bespielsweise Straßen und Wege vordefinieren und stellen dadurch die eindeutige Grundstruktur eines urbanen Gebietes her. Bei Auswertung aller Daten entsteht ein Höhenprofil der Oberfläche einschließlich aller Objekte. Anhand des Profils erfolgt eine automatische Extraktion der Oberfläche, sodass 3D-Objekte auf der Oberfläche entstehen [HB98] (vgl. Abb. 2.3) . Durch diese Methoden und die stetig ansteigende Datenqualität ist eine immer genauere Erkennung von polyedrischen Objekten möglich. Gebäudedächer, Straßen, Parkplätze oder Geländeformen lassen sich gut rekonstruieren. Angesichts der geometrischen Komplexität von Bäumen und Pflanzen beschränkt man sich oft bei der automatischen Erkennung von Vegetation auf die Einteilung der Flächen [Wei].

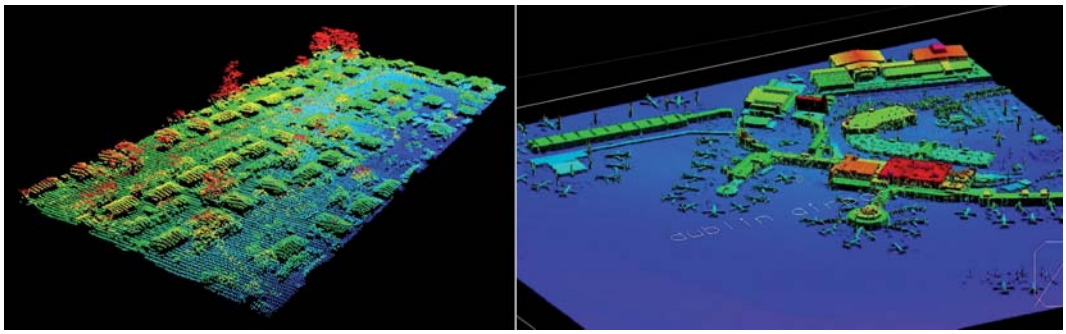

**Abbildung 2.3**: *Digital Surface Model*. Links: Höhenprofildaten in Form einer Punktwolke, rechts: DSM eines 3D-Modell des Dubliner Flughafen[1]

### 2.2.3 Rome Reborn

Nicht immer sollen oder können Städte anhand realer Vorgaben mittels Bild oder Profildaten nachgebildet werden. Mit dem internationalen Projekt *Rome Reborn*[2] der Universität Virginia wird versucht ein detailliertes interaktives Modell des antiken Roms aus dem Jahr 320 n.Chr. zu erschaffen.
Mittlerweile wird an der zweiten Version des Projektes gearbeitet. Zahlreiche archäologische Information liegen zur Nachbildung der antiken Stadt vor. Allerdings weisen diese Informationen bis dato Lücken auf, sodass ein vollständig exakter Nachbau nicht möglich ist. Im antiken Stadtmodell werden ca. 7.000 Gebäude nachbildet. Bei 250 von ihnen handelt es sich um eindeutig wiederekennbare Monumentalbauten, wie das *Collosseum*, das *Forum Romanum* oder der *Circus Maximus*. Diese Bauwerke wurden detaillgetreu und aufwändig manuell nachempfunden oder aus verschiedenen Quellen bezogen. Bei allen anderen Bauwerken handelt es sich um gewöhnliche Wohnhäuser, Lager oder sonstige Gebäude, zu denen keine präzisen Informationen vorhanden sind. Diese ca. 6.750 Gebäude wurden in der ersten Version des Projekts manuell erstellt und platziert. Das erforderte einen enormen Personal- und Zeitaufwand [DFM+]. Mit zunehmender Bekanntheit wurde das Projekt auch innerhalb von *Google Earth* implementiert. *Google Earth* stellt eine der weltweit umfangreichen Lösungen zur Visualisierung von Stadt- und Geländemodellen in Echtzeit dar. Durch ein standardisiertes Austauschformat für Geodaten (*KML, Keyhole Markup Language*) konnten die 3D-Daten des virtuellen Roms importiert werden [WFRK]. Eine automatisierte Lösung zur Erstellung oder Portierung ist nicht vorgesehen. Für die zweite Version von *Rome Reborn* werden diese Gebäudemodelle prozedural erstellt. Dazu wird die kommerzielle Anwendung *CityEngine* eingesetzt. Informationen zum Gelände für die richtige Platzierung der Modelle liegen

[1] Bild links: Schlenker Mapping, Australia (http://www.schmap.com.au/Images/lidar1.jpg). Bild rechts: Ordnance Survey, Ireland (http://www.osi.ie/Products/Professional-Mapping/Height-Data.aspx).

[2] http://www.romereborn.virginia.edu

als DTM vor. Straßennetzwerke können daraufhin innerhalb ausgewählter Bereiche automatisch erstellt werden. Die entstehenden Zwischenräume entsprechen den Bereichen in denen Gebäude platziert werden können. Zunächst werden einfache Formen an diesen Stellen positioniert, welche dann in einem späteren Erstellungsschritt durch detaillierte Häusermodelle ersetzt werden. Dabei werden die Häuser auch prozedural mit zufälligen Texturen versehen. [Feu10]. Die *CityEngine* wird im Folgenden näher erläutert.

**Abbildung 2.4**: 3D-Modell des antiken Roms aus dem Projekt *Rome Reborn*[3].

## 2.3 Kommerzielle Lösungen

### 2.3.1 CityEngine

Bei der wohl am weit verbreitetsten Software zur prozeduralen Erstellung von 3D-Stadtmodellen handelt es sich um die *CityEngine*[4] der Firma *ESRI Inc.* (ehem. *Procedual Inc.*). Die *CityEngine* stellt ein riesiges Potential an Funktionen und Einsatzmöglichkeiten bei der automatischen Erzeugung der Modelle zur Verfügung. Die Schwerpunkte der Software richten sich auf die Erzeugung von 3D-Stadtmodellen aus 2D-Geodaten, 3D-Design basierend auf 2D-Geodaten bzw. anhand festgelegter Regeln sowie auf die dreidimensionale Modellierung virtueller urbaner Gebiete für Simulationen und Unterhaltunsindustrie[5]. Dabei werden sämtliche gängigen Industriestandards für Im- und Export unterstützt.

[3]Bild: ©2010 Bernard Frischer. 3D-Modell: ©2008 University of California.
[4]http://www.esri.com/software/cityengine/index.html
[5]http://www.esri.com/software/cityengine/common-questions.html

Der gesamte prozedurale Erstellungsprozess (vgl. Abb. 2.5) der *CityEngine* basiert auf dem parametrisierbaren Regelwerk für die Automatisierung. Anhand von veränderbaren Regeln lassen sich dynamisch nahezu alle Eigenschaften steuern und anpassen. Der übliche Erstellungsprozess beginnt mit dem Import von Geodaten, zum Beispiel Kartenmaterial in Form von 2D-Vektordaten. Anhand dieser Daten werden Straßen mit verschiedenen Eigenschaften definiert. Das entstehende Straßennetzwerk gibt die Grundstruktur der Stadt vor. An den Straßen müssen Parzellen definiert werden, in denen später Häuser erstellt werden. Mit den Parzellen reserviert die Software den maximal möglichen Platz pro Gebäude. Danach folgt die Erzeugung der Gebäude in den Parzellen nach den vorher festgelegten Regeln. Gleichzeitig werden die Modelle mit ebenfalls prozedural erzeugten Texturen ausgestattet. Für nachträgliche Bearbeitung erstellter Modelle können die Erstellungsregeln jederzeit geändert und die Änderungen automatisch für alle Gebäude dynamisch übernommen werden. Auch Straßen können in Echtzeit anders positioniert oder verändert werden. Die von der veränderten Straße abhängigen Häuser passen sich automatisch an.

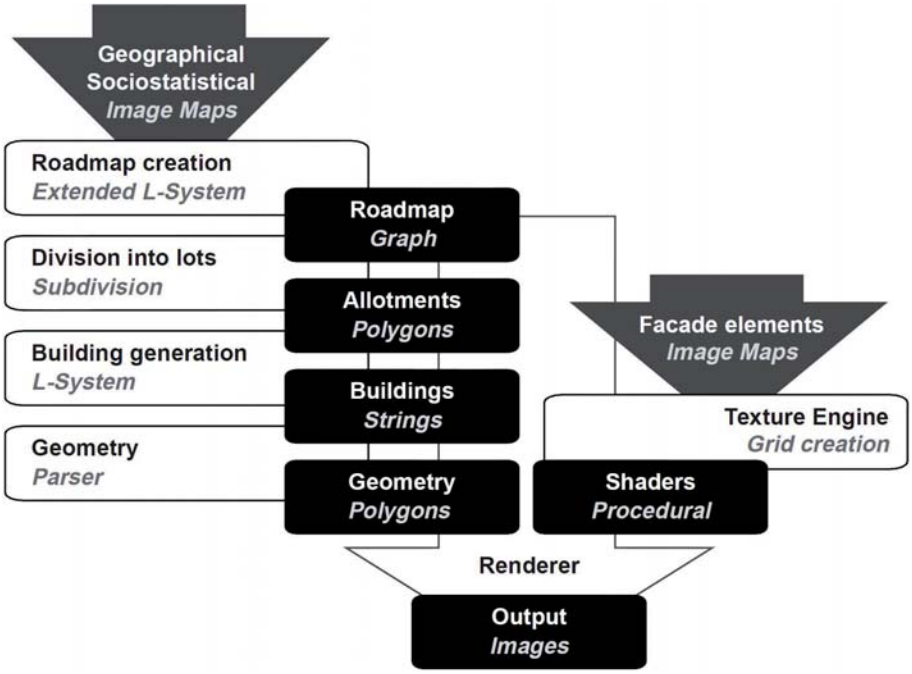


**Abbildung 2.5**: Pipeline des Stadterstellungsprozesses der CityEngine [MP01]. Resultierend aus den weißen Boxen repräsentieren die schwarzen Boxen die Ergebnisse und Daten des Erstellungsprozesses.

Die Verwendung von Straßennetzwerken anhand von Karten- oder anderen Geodaten ist nicht zwingend vorgeschrieben. Auch eigene individuelle Modelle oder Texturen können erstellt werden. Allerdings nur auf prozeduralen Weg und auch nur anhand von Regeln innerhalb der Software. Komplette eigene Modelle, die nicht prozedural verar-

beitet werden sollen, können nicht verwendet werden. Besonders interessant für den Einsatz in Echzeitgrafikanwendungen ist die Erstellung verschiedener *Level of Detail* der Häusermodelle. Bei der niedrigsten LOD-Stufe sind die *Meshes* allerdings immernoch verhältnismäßig komplex unterteilt. Die *Meshunterteilungen* sind notwendig, da die Anwendung auf Basis dieser Unterteilungen die Erstellung der LOD-Stufen und die Positionierung der Texturen eines Modells steuert (vgl. Abb. 2.6, linkes Bild). Solche Modelle können mit der hohen Anzahl der Polygone, insbesondere bei tausendfachem Vorkommen, schnell zur Performancegrenze einer Echtzeitanwendung führen.

**Abbildung 2.6**: *CityEngine*-Fassade mit *Level of Detail*. V.l.n.r.: LOD0, LOD1, LOD2[6].

Die *CityEngine* bietet außerdem eine Funktion für die automatische Erzeugung von *Collider Meshes*, wie sie oft in Spielen oder interaktiven Anwendungen notwendig sind. Jede Änderung am Modell überträgt sich automatisch auf das *Collider Mesh*. Ein weiterer Vorteil zum Einsatz in *Game Engines* ist die integrierte Exportfunktion, die einzelne Gebäude oder ganze Stadtmodelle zu *Unity* oder *Unreal* übertragen kann. Beim Export einzelner Modelle werden auch alle Texturen einzeln exportiert und erhalten in *Unity* jeweils ein Material, wodurch eine manuelle Opmtimierung zur Einhaltung der Echtzeitperformance erforderlich wird (siehe Kapitel 4.3.2). Eine Stadt kann auch als Gesamtmodell exportiert werden, womit allerdings ein gezieltes *Culling* von Einzelmodellen oder mehrere bereichsabhängiger Modelle verhindert wird (siehe Kapitel 4.3.1), sodass stets das gesamte Stadtmodell aufwändig gerendert werden muss. Weiterhin hat *CityEngine* keine Möglichkeit auf ein in der *Game Engine* verwendetes Terrain zuzugreifen. Das bedeutet für jede Änderung des Terrains in der *Game Engine* ist ein erneuter Export zur *CityEngine* erforderlich um dort die Änderungen an der Stadt zu vollziehen, die wiederum zurück exportiert werden müsste. Darüber hinaus ist für *Unity* vom Hersteller keine Funktion zum Terrianexport vorgesehen.

[6]Bilder: http://www.arcgis.com/home/item.html?id=6099d8a2e81f4c33b9f9021f1fee266d

Um einer statischen Stadt Leben zu verleihen werden in Spielen oder Simulationen oft KI-Systeme (Künstliche Intelligenz) implementiert. Mit ihnen kann computergesteuert die Interaktion von Menschen und Maschinen mit der Umgebung oder dem Benutzer simuliert werden. Die *CityEngine* sieht dafür keine Lösung vor. Mit der *CityEngine Advanced* besteht durch ein integriertes *Python Scripting Interface* die Möglichkeit, umfangreiche, wiederkehrende Operationen zu automatisieren. Erforderliche Funktionen können auf diese Weise unter Umständen nachgerüstet werden.

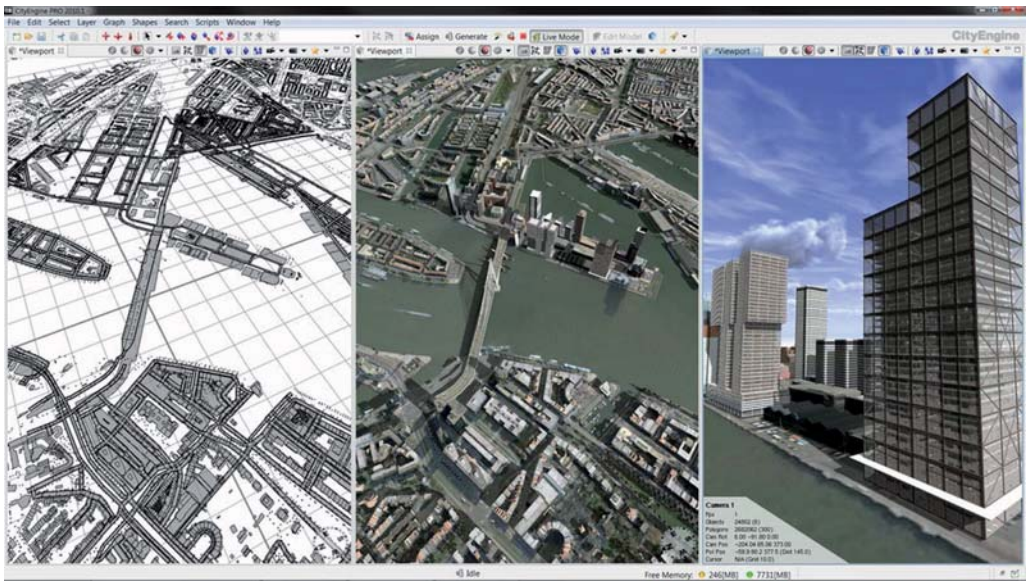

**Abbildung 2.7**: Grafische Benutzeroberfläche der *CityEngine*. Links: 2D-Geodaten, Mitte und rechts: prozedural erzeugte Stadtmodelle[7].

### 2.3.2 Lösungen für Internet und mobile Geräte

Neben der prozeduralen Erzeugung von 3D-Modellen für die Verwendung in Echtzeitumgebungen oder in aufwändigen Renderings setzen sich verstärkt Enwicklungen im Bereich der Internet- und Browserapplikationen durch. Neueartige Technologien erlauben es aufwändige Rechenoperationen mittels *Cloud Computing* und schnellen Datenübertragungsraten auf Internetdienste zu verlagern. Umgangssprachlich als *Web 2.0* bekannt stellen auch aktuelle Browser Möglichkeiten für interaktive und grafisch anspruchsvolle Anwendungen bereit. Durch eine rasante Weiterentwicklung der Leistungsfähigkeit mobiler Endgeräte können die meisten dieser neuen Internetangebote auf Smartphones oder Tablet-Computern jederzeit und standortunabgängig genutzt werden. Basierend auf der Spezifikation von *OpenGL* ist es mit *WebGL (Web Grapics Li-*

[7] Bild: ©Pascal Mueller (Quelle: http://www.esri.com/software/cityengine/features.html)

*brary)* möglich 3D-Grafik durch die Einbindung entsprechender Hardwarekomponenten in Echtzeit direkt im Browser darzustellen [Gab11].
Internetdienste wie *Google Earth* ermöglichen die dreidimensionale Darstellung von Geoinformationen. Terrain und enthaltene Bauwerke werden als 3D-Inhalt angezeigt. Zwar kann die Darstellungsqualität der 3D-Grafiken noch nicht vollständig mit denen von Echtzeitumgebungen oder gar Renderings entsprechen, aber viele weltweite Projekte beschäftigen sich kontinuierlich mit der Entwicklung im Medium des Internets.

### Rotterdam in 3D mittels Cloud Computing

Bei der Zusammenarbeit von *Mental Images*[8], einer Tochtergesellschaft der *NVIDIA Corporation*, und *ESRI CityEngine* versuchen die Firmen mittels neuer Technologien eine Webanwendung zur Darstellung von fotorealistischen und interaktiven 3D-Szenen zu entwickeln. Das Projekt kollaboriert mit der Stadt Rotterdam um die Stadt mit Hilfe von Geodaten, prozeduralen Erstellungstools und *GPU-Rendering* mittels *Cloud Computing* als 3D-Modell nachzubilden[9]. Durch das Projekt soll eine neue Art der Erstellung, Analyse und Visualisierung von fotorealistischen 3D-Stadtmodellen geschaffen werden. Der bei Desktopanwendungen übliche prozedurale Stadterstellungsprozess anhand von Geoinformationen soll mittels *RealityServer*[10] GPU-basierend in einer *Cloud Computing Environment* berechnet werden. Dadurch sollen unter anderem mobile Endgeräte in der Lage sein die Grafikdaten der komplexen Stadtmodelle mittels Echtzeitübertragung darzustellen. Darüber hinaus ist eine Echtzeitbearbeitung der Gebäudemodelle mit Hilfe der bewährten *CityEngine*-Tools möglich sein. Das Stadtmodell besteht dazu aus verschiedenen *Layern* mit denen gezielte Änderungen an bestimmten Modellen vorgenommen werden können, ohne dabei die gesamte Stadt editieren zu müssen. Die *Layer* werden bei nach erfolgter Änderung an den *RealityServer* übertragen, welcher die entsprechenden Objekte neu rendert.

**Abbildung 2.8**: Fotorealistisches 3D-Stadtmodell von Rotterdam für mobile Endgeräte umgesetzt mit Hilfe der *CityEngine* und *Cloud Computing* via *RealityServer*[11].

[8] http://www.mentalimages.com/index.php
[9] http://www.esri.com/software/cityengine/casestudies/rotterdam.html
[10] http://www.mentalimages.com/products/realityserver.html

### Nokia Maps

Mit ähnlichen Zielen zur webfähigen Darstellung kompletter Städte im Browser beschäftigte sich auch einer der weltgrößten Mobiltelefon-Hersteller *Nokia*. Auf Basis von Luftbilddaten einer schwedischen Firma namens *C3 Technologies* bietet *Nokia* einen eigenen Landkartendienst im Internet an, mit dem beeindruckende 3D-Stadtansichten ermöglicht werden. Die ursprünglich zur Navigation von Marschflugkörpern entwickelte Software, berechnet dazu in einem Browser-Plugin für jeden Blickwinkel eine fotorealistische Darstellung aus verschiedenen Winkeln der Luftbilder. Bewegt sich der Betrachter innerhalb der 3D-Stadt wird zunächst perspektivisch korrekt eine Fassadenfront angezeigt, gefolgt vom Boden und schließlich werden die gegenüberliegenden Fassaden berechnet und angezeigt [Sch12]. Anders als bei *Google Earth* werden hierbei alle Objekte dreidimensional gezeigt, nicht nur ausgewählte bzw. einzeln erstelle Gebäude aus einer Datenbank[12]. Da die Berechnung ausschließlich auf den Luftbilddaten aus verschiedenen Winkeln erfolgt wirken die Texturen teilweise verzehrt [Sch12]. Bei Flug über die Stadt wird dennoch ein bemerkenswert realistischer Eindruck geschaffen. Um den dreidimensionalen Eindruck zu verstärken wird zu jeder 3D-Stadt auch eine stereoskopische Ansicht angeboten.

**Abbildung 2.9**: WebGL basiertes 3D-Modell der Stadt Prag mit *NokiaMaps*. Unten: Stereoskopische Darstellung[13].

[11] Bild links: http://www.esri.com/software/cityengine/casestudies/rotterdam.html.
Bild rechts: http://www.mentalimages.com/products/realityserver.html

[12] http://www.technologyreview.com/news/423838/ultrasharp-3-d-maps/

[13] Bilder: http://maps.nokia.com/webgl/

Bis heute sind 25 amerikanische und europäische Stadte komplett als 3D-Modell integriert. Weitere Ziele des Projekts sind bis dato nicht offizell bekannt, die Firma *C3 Technologies* wurde im Sommer 2011 durch *Apple* übernommen. Vorraussichtlich werden die Kartendaten in das kommende mobile *Apple*-Betriebssystem *iOS 6* integriert[14].

### 2.3.3 Tridicon

Die vorangegangen beschriebenen Lösungen bieten Möglichkeiten zur nahezu komplett automatisierten Erstellung von 3D-Stadtmodellen. Dabei stellen die Automatismen von Modellierung und Platzierung keine Unwegbarkeiten dar. Problematischer sind die Verfahren für Texturierung. Der manuelle Aufwand sämtliche innerstädtischen Modelle exakt nach realen Vorbildern zu texturieren ist enorm. *Tridicon* der Firma *GTA Geoinformatik GmbH* ist eine Softwarelösung, die den gesamten Prozess von prozeduraler Erstellung bis hin zur automatischen Texturierung abbilden soll. Dabei nutzt die Software viele der vorher beschriebenen Methoden. Die Grundlage können Geländedaten in eines *Digital Surface Model* bilden. Durch Kombination von *Digital Terrain Model*-Daten, Stereoluftbildern sowie Gebäudegrundrissdaten können mit der Software *tridicon 3D City Modeler*[15] Objekte generiert und den realen Bauwerken exakt nachempfunden werden. Darüber hinaus ist auch hier die Erstellung der einzelnen Modelle in mehreren *Level of Detail* möglich. Viele der bereits beschriebenen Funktionen bildet die Software ab. Dennoch besitzt die Anwendung durch ihre *tridicon TEXTURE*- Optionen[16] ein wesentliches Alleinstellungsmerkmal. Damit können aus georeferenzierten terrestrischen Digitalbildern Dächer und Fassaden der Gebäude automatisch mit einer realitätsnahen Textur versehen werden. Unterstützend können Stereoluftbilder bzw. Schrägsichtluftbilder zu hilfe genommen werden.
Zunächst werden anhand der Bilddaten die Farben von Dächern und Fassaden ermittelt. Anschließend werden mittels Bildanalyse Form und Ausmaße von Fensterreihen ermittelt [Lie08]. Auf Grundlage der errechneten Daten können prozedural detaillierte Modelle der Fassadenelemente erstellt werden. Damit lässt sich der Erstellungsprozess komplett auf die Analyse verschiedener Bild- und Geländedaten übertragen, sodass der Erstellungsaufwand auf ein Minimum reduziert werden kann.
Die Firma stellt für die georeferenzierten Bilder regional angepasste Texturbibliotheken bereit, in denen alle abfotografierten Ortschaften gespeichert werden. Zur Texturierung und der darauffolgenden automatischen Modellerzeugung können auch andere Bilder oder Texturen herangezogen werden. Die Software unterstützt darüber hinaus den Im- und Export aller gängigen Dateiformate. Auch eine Schnittstelle zur *CityEngine* ist implementiert, mit der Daten aus der *tridicon*-Software in die die regelbasierte Umgebung der *CityEngine* übertragen werden können.

[14] http://www.heise.de/mac-and-i/meldung/Geruecht-iOS-6-setzt-auf-Apple-Kartenmaterial-1574025.html
[15] http://www.tridicon.de/software/tridicon-3d-citymodeller/
[16] http://www.tridicon.de/software/tridicon-texture/

**Abbildung 2.10**: Automatisierte Modellierung und Texturierung anhand von Bilddaten mit *tridicon* [Lie08]. Links: anhand von Geo- und Bilddaten automatisch modellierte Gebäude, mitte: mittels Bilddaten ermittelte Fensterreihen, rechts: automatische Texturierung mit georeferenzierten Originalbildern[17].

## 2.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Lösungen oder Lösungsansätze beschrieben, die alle das Ziel verfolgen möglichst realitätsnahe Stadte- bzw. Gebäudemodelle durch automatische Prozesse zu erzeugen. Neue Technologien zur Realisierung solcher Modelle als Internet- und Browserapplikationen wurden erläutert und lassen darauf schließen, dass sich in diesem Bereich zukünftig weitere Projekte entwickeln und Lösungen realisiert werden. Die in der Vergangenheit deutlich spürbaren Unterschiede und Grenzen in Bezug auf Leistungsfähigkeit bei Grafik zwischen Desktop-, Echtzeit- und Internetanwendungen werden mehr und mehr verwischen. Aus den beschriebenen Methoden und Softwarelösungen lassen sich weitere Ideen ableiten oder gar weiterdenken. Keine der genannten Vorgehensweisen deckt alle Bereiche der Computergrafik ab.
Im folgenden Kapitel werden einige Grundlagen aus dem Bereich der Echtzeitgrafik beschrieben.

[17] Bilder: http://www.tridicon.de/download/informationsmaterial/

# Kapitel 3

# Echtzeitgrafikverarbeitung

Im vorangegangen Kapitel wurden aktuelle Konzepte und Softwarelösungen für die prozedurale 3D-Stadterzeugung bzw. Darstellung für verschiedenen Bereiche der Computergrafik beschrieben. In diesem Kapitel soll das Basiswissen insbesondere im Echtzeitbereich der Computergrafikverarbeitung vermittelt werden. Es stellt das grundlegende Verständnis für die weiteren Kapitel her, bei denen dieses Wissen angewandt wird. Erfahrende Leser können dieses Kapitel überspringen.
Die folgenden Kapitel beschreiben die Erreichung der im ersten Kapitel gesetzten Ziele anhand eines Konzepts. Anschließend werden die Ergebnisse aufgezeigt und ein Ausblick für weitere Maßnahmen gegeben.

## 3.1 Einleitung

Das Gebiet der Computergrafik beschäftigt sich als Teilbereich der Informatik mit der Erzeugung computergenerierter Bilder. Computergrafiken werden dazu hauptsächlich in die Bereiche zweidimensionale bzw. dreidimensionale Grafiken unterteilt. Beide Teilbereiche befassen sich zusätzlich mit der Herstellung bewegter Bilder und Animationen.
Die generierten Bilder oder Bilderfolgen durchlaufen für die Darstellung auf dem Bildschirm viele Teile der zugrunde liegenden Hardware-Architektur eines Computersystems. Die zur Zeit vorherrschende Technik zur Verarbeitung comuptergenerierter Bilder ist die Rastergrafik. Eine rechteckige Matrix mit Werten der darzustellenden Pixel beschreibt dabei das Bild auf dem Bildschirm, der sogenannte *Frame Buffer* [BB06].

## 3.2 Die Rendering Pipeline

Eine Pipeline beschreibt einen festgelegten Ablauf mehrerer aufeinander folgender Schritte. Die Reihenfolge der Prozesse ist nicht veränderbar, da die Folgeschritte innerhalb der Pipeline voneinander abhängig sind. Jeder Einzelschritt liefert ein Ergebnis worauf ein nachfolgender aufbaut. Die interne Ausführung eines Einzelschrittes erfolgt jedoch unabhängig von den restlichen Verarbeitungsprozessen. Durch gleichzeitge Berechnung der Einzelschritte kann die Verarbeitungszeit der Gesamtpipeline wesentlich verringert werden. Auf diese Weise kann die Gesamtpipelineverarbeitung nur maximal so schnell sein, wie der langsamste Einzelprozess innerhalb der Pipeline. Dieser Prozess wird als *Bottleneck* bezeichnet. Von ihm ist die gesamte Pipeline und alle darauf folgenden Prozesse abhängig. Abgeschlossene Prozesse, die schneller berechnet werden können, sind gezwungen auf die noch laufenden Prozesse zu warten. Erst wenn alle Schritte abgearbeitet wurden kann das Gesamtergebnis aus der Pipelineverarbeitung zusammengesetzt werden. Idealerweise kann zur effektiven Ausnutzung der Kapazitäten die Verarbeitung aller Prozesse so gestaltet werden, dass deren Leerlaufzeit gegen null geht. Innerhalb der einzelnen Schritte kann wiederum eine Pipeline-Verarbeitung stattfinden [AMHH08].

Diese Form der Pipeline-Verarbeitung stellt in der Computergrafik die *Rendering Pipeline* dar. Sie ist die Grundlage heutiger Echtzeit-Computergrafik und beschreibt den Weg eines Bildes oder einer virtuellen Szenerie zur Darstellung auf dem Bildschirm. Aufgeteilt werden kann die *Rendering Pipeline* in drei Abschnitte, in denen jeweils die Verarbeitung einer weiteren internen Pipeline-Struktur stattfindet (vgl. Abb. 3.1).

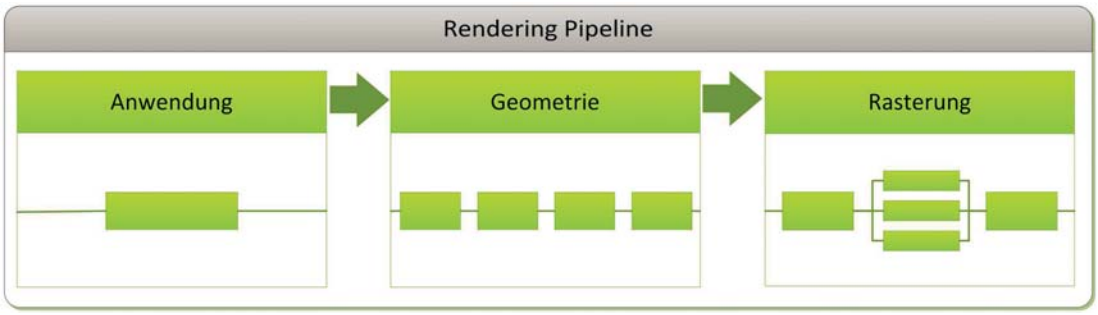


**Abbildung 3.1**: *Rendering Pipeline*. Angelehnt an [AMHH08].

Der erste Pipeline-Abschnitt *Anwendung* wird durch Funktionen gesteuert, die in Software implementiert sind. Die Verarbeitung dieser Anwendungen wird üblicherweise innerhalb der *CPU* vorgenommen. Die *CPU (Central Processing Unit)* besteht heutzutage aus meheren Rechenkernen, wodurch eine parallele Berechnung verschiedener Operationen ermöglicht wird. Dadurch kann die *CPU* eine Reihe von Aufgaben effizient gleichzeitig verarbeiten. Die Verarbeitung der Softwareanwendungen und die Steuerung der *CPU* liegt in der Verantwortung des Pipeline-Abschnittes der *Anwendungen*. Abhängig von der Softwareanwendung handelt es sich bei den durch die *CPU* zu verarbeitenden

Schritte beispielsweise bei einer Grafik- oder Spielesoftware um Kollisionserkennung, Animationen, Physiksimulationsberechnungen oder das Aussortieren nicht sichtbarer Objekte (siehe Kapitel 4.3.1). Die Entwickler einer Software haben volle Konrtolle über die interne *CPU*-basierte Verarbeitung einer Software. Zur Verbesserung der Performance können die Entwickler die Art der Verarbeitung durch das System anpassen. Die Änderungen können sich auf den kompletten weiteren Verarbeitungsprozess auswirken. Beispielsweise durch das Weglassen von Geometrieobjekten in einer 3D-Szene sind weniger Objekte und Daten an die weiteren Schritte der Pipeline zu übergeben bzw. zu berechnen, wodurch eine Beschleunigung des Gesamtprozesses erzielt werden kann [AMHH08].

Die Daten werden nach der Berechnung dem nächsten Abschnitt *Geometrie* übergeben, indem ebenfalls eine interne Pipeline abgearbeitet wird. Die Berechnung und Verarbeitung sämtlicher Prozesse erfolgt ab jetzt durch die *GPU (Graphis Processing Unit)*. Die Daten werden dazu durch die *CPU* aus dem Hauptspeicher des Systems in den Speicher der Grafikkarte übertragen. Zur schnelleren Berechnung der im Grafikspeicher befindlichen 3D-Objekte werden für die Instanzen eines Objektes *Vertex Buffer Objects (VBO)* angelegt. Diese VBO speichern Indizes aller *Vertices* eines Objektes in einem Puffer. Soll ein Objekt gezeichnet werden müssen die *Vertex*-Koordinaten nicht vielfach zur Laufzeit neu gelesen und berechnet werden, sondern können über diese Referenzierung schneller abgerufen werden [NVI03]. Die im Abschnitt der *Geometrie* befindliche interne Pipeline (vgl. Abb. 3.2) wird anschließend abgearbeitet.

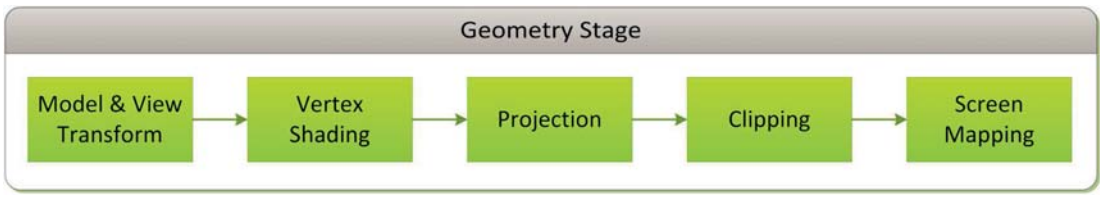


**Abbildung 3.2**: Geometrie-Verarbeitung innerhalb der *Rendering Pipeline*. Angelehnt an [AMHH08] (vgl. Abb. 3.3).

Im ersten Schritt *Model & View Transform* werden alle *Vertices* und die *Normalen* jedes Objekts in ein Weltkoordinatensystem transformiert, sodass es im 3D-Raum orientiert bzw. positioniert werden kann. Instanzen, die auf ein Objekte verweisen, erhalten unabhängig vom Objekt jeweils einen eigenen *Transform*. Beim Schritt des *Vertex Shading* wird für alle Materialen der Objekte die Lichtsituation simuliert, wodurch die Objekte ihr Erscheinungsbild erhalten. Wenn erforderlich werden dabei Eigenschaften wie Glanz oder Reflektionen berücksichtigt.
Im nächsten Verarbeitungsschritt *Projection* wird das dreidimensionele Sichtfeld der Kamera (*View Frustum*) mit Hilfe eines Einheitswürfels auf eine zweidimensionale Fläche projiziert (vgl. Abb. 3.3). Diese Fläche entspricht der Bildfläche [Feu10]. Objekte die außerhalb des *View Frustum* der Kamera liegen oder nicht sichtbare Flächen werden

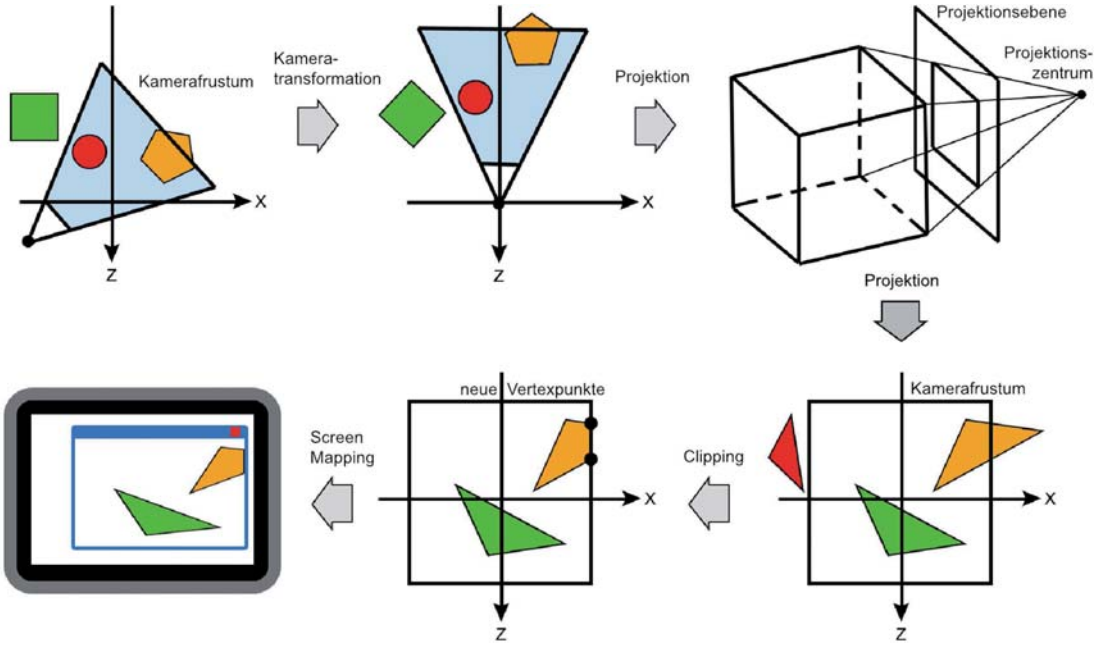


**Abbildung 3.3**: Einzelschritte der Geometrie-Verarbeitung innerhalb der *Rendering Pipeline* visuell verdeutlicht. Angelehnt an [AMHH08]. Quelle [Feu10].

für die weitere Verarbeitung nicht berücksichtigt und verworfen (*Culling*, siehe Kapitel 4.3.1). Beim *Clipping*, dem nächsten Schritt, werden irrelevante Teilflächen außerhalb des Einheitswürfels, und somit außerhalb der Bildfläche, durch Wegschneiden entfernt (vgl. Abb. 3.3). Die spätere pixelweise Verarbeitung durch den *Rasterizer* kann so ausschließlich auf die sichtbaren Flächen bezogen und Verarbeitungszeit eingespart werden [BFN03]. Die zerschnittenen Objekte werden dem letzten Schritt der *Geometrie*-Pipeline übergeben. Die dreidimensonalen Koordinaten der Objekte befinden sich innerhalb des Einheitswürfels. Das *Screen Mapping* ermittelt die zweidimensionalen Koordinaten für die Bildschirmausgabe und speichert zusätzlich die Tiefeninformationen als Z-Koordinaten [AMHH08]. Die in der *Geometrie*-Pipeline errechneten Daten werden dem *Rasterzier* zu Weiterverarbeitung übergeben.

Ähnlich wie der Verarbeitungsabschnitt der *Geometrie* findet innerhalb der *Rasterung* ebenfalls eine Pipeline-Verarbeitung statt (vgl. Abb. 3.4). Bei der *Rasterung* werden die Farben jedes Pixels der Objekte für die spätere Bildschirmausgabe errechnet. Zunächst werden dazu beim *Triangle Setup* die Differentiale der Oberflächen der Polygone (*Triangles*) ermittelt. Dadurch kann jedem Pixel ein Wert zugewiesen werden [Feu10]. Im nächsten Schritt *Triangle Traversal* erhalten die vorher berechneten inneren Flächen der Polygone ihre Farben. Dieser Schritt der Rasterung wird auch als *Scan Conversion* bezeichnet, bei der einfache Objekte nach einem Pixelraster gezeichnet werden.

Auf Grundlage der beim *Vertex Shading* entstandenen interpolierten *Shading*-Daten erfolgen beim *Pixel Shading* alle weiteren Berechnungen pro Pixel. Hier erhält jedes Pixel seine entgültige Farbe. Im Gegensatz zum *Triangle Setup* und *Triangle Traversal* ist der *Pixel Shader* nicht unveränderbar auf der Hardware eingebunden. Das *Pixel Shading* wird auf der programmierbaren *GPU* (siehe Kapitel 3.3) ausgeführt, wodurch Einfluss auf die Ausführung genommen werden kann. Der letzte Schritt im *Rasterizer* ist das *Merging*, bei denen die vom *Shading* gelieferten Farben zusammengefasst werden. Die Farbwerte werden als *Color Buffer* geliefert, bestehend aus einer Rot-, Grün- und Blau-Komponente. Beim *Merging* wird darüber hinaus sichergestellt, dass die Farben des *Color Buffes* der Objekte durch Verdeckung anderer Objekte ausgehend vom aktuellen Betrachtungspunkt korrekt dargestellt werden. Kameranahe Objekte verdecken weit entferntere Objekte. Mit Hilfe des *Z-Buffer* oder auch *Depth Buffer* werden die Entfernungen einzelner Objekte zur Kamera gespeichert. Dadurch kann die korrekte Renderreihenfolge gewährleistet werden [AMHH08].

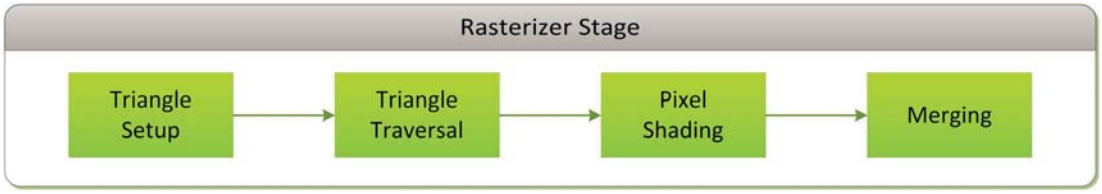


**Abbildung 3.4**: Rasterisierung innerhalb der *Rendering Pipeline*. Angelehnt an [AMHH08].

Vor die Ausgabe des *Frame Buffer* auf dem Bildschirm erfolgen mit den *Raster Operations (ROP)* einige abschließende Maßnahmen, erneut in Form einer internen *Pipeline*. Die ROP stellen die letzten Schritte bei der Verarbeitung der *Rendering Pipeline* dar. Eine Reihe verschiedener Tests wird durchgeführt um sicherzustellen, dass die Objekte im Bild korrekt darstellt werden. So wird das Rendern von Objekten hinter transparenten Flächen durch den *Alpha Test* geprüft. Mit Hilfe des beim *Merging* gespeicherten *Z-Buffer* wird die Renderreihenfolge der Objekte untersucht und mittels *Stencil Test* werden schließlich verdeckte Flächen eliminiert. Schlägt einer der durchgeführten Tests bei einzelnen Bildfragmenten fehl, wird das Fragment verworfen und die Pixel des neues Bildes werden an diesen Stellen nicht aktualisiert. Anschließend werden die Pixel des aktuellen Bilds beim *Blending* mit dem des neuen Bildes vermischt. Durch den Prozess des *Dithering* werden dabei entstehende Fehler oder Farbübergange ausgeglichen bzw. opmitmiert [FK03].

**Zusammenfassung**

Die *Rendering Pipeline* beschreibt die Umwandlung der 3D-Modelldaten in eine zweidimensionales Bild [Sud05]. Die wichtigsten Schritte in der Verarbeitung der *Pipeline* durch die Grafikkarte werden beispielhaft durch die Abbildung 3.5 visuell verdeutlicht. In der Abbildung erfolgt die Rasterung zweier Dreiecke (*Triangles*). Der Prozess beginnt mit der Transformation und der Einfärbung der *Vertices*. Als nächstes werden die einzelnen *Vertices* zu Primitiven zusammengesetzt, in diesem Fall Dreiecke. Anschließend *füllt* der *Rasterizer* die Dreicke mit Fragmenten. Abschließend werden die Registerwerte der *Vertices*, die anfangs gespeichert wurden, zur Interpolation, Texturierung und Färbung der Primitive genutzt. Es fällt auf, dass bereits zur Darstellung dieser einfachen Dreiecke verhältnismäßig viele Fragmente gebildet werden mussten [FK03].

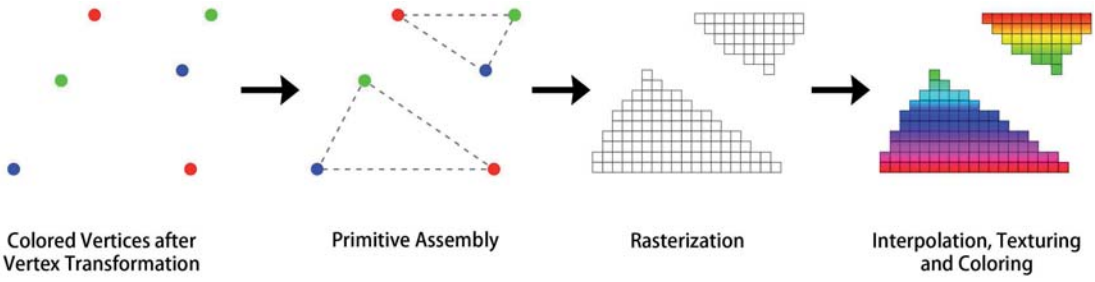


**Abbildung 3.5**: Vereinfachte, visualisierte Verarbeitung durch die *Rendering Pipeline*. Angelehnt an [FK03].

## 3.3 Shader

Der erste Abschnitt der *Rendering Pipeline* (vgl. Abb. 3.1) beschreibt die *CPU*-basierte Steuerung sämtlicher Prozesse durch Softwareanwendungen. Grafikdaten dieser Prozesse werden zur Verarbeitung an die *Grafik-API (Application Programming Interface)* übertragen. Sie stellt die Schnittstelle des Rechnersystems zur Grafikhardware dar. Die bekanntesten *Grafik-APIs* sind die plattformunabhängige Spezifikationen *OpenGL* sowie die von *Microsoft* entwickelte Spezifikation *Direct3D*. Beide stellen heutzutage standardisierte Programmierschnittstellen für unterschiedlichste Grafikhardware dar [Ebs05].
Die *Grafik-API* ermöglicht den Zugriff auf viele Komponenten und Verarbeitungsschritte der *Rendering Pipeline*. Durch eine programmierbare *GPU* ist es weiterhin möglich einige dieser Komponenten zur Verarbeitung der Grafikdaten zu kontrollieren. Die Programme zur Steuerung der Grafikkhardware werden als *Shader* bezeichnet. Viele der abstrahierten Einzelschritte innerhalb der *Rendering Pipeline* sind wenig oder gänzlich unveränderbar. Mit jeder Weiterentwicklung der *Grafik-APIs* wird der programmierbare Funktionsumfang stetig gesteigert und erlaubt den Zugriff auf mehr Komponenten. So können heute aufwändige Berechnungen von der *CPU* auf leistungsstarke *GPUs* ausge-

lagert werden. Dadurch können mehr Grafikoperationen gleichzeitig berechnet werden, was sich letztlich in einer gesteigerten Grafikqualität im gerenderten Bild niederschlägt.

Bei den wesentlichen programmierbaren Komponenten handelt es sich derzeit um den *Vertex Shader*, den *Pixel* bzw. *Fragment Shader* sowie den *Geometry Shader*.
Mit dem *Vertex Shader*, der innerhalb der *Rendering Pipeline* für die Transformation der Geometrien zuständig ist, kann Einfluss auf die *Vertices* der Modelle genommen werden, indem beispielsweise die Positionen einzelner *Vertices* verändert werden können. Darüber hinaus kann mittels *Vertex Colors* der Farbwert eines Objektes manipuliert werden [Sud05].
Der *Pixel Shader* oder auch *Fragment Shader*, welcher im letzten Abschnitt der *Rendering Pipeline* ausgeführt wird, bestimmt die endgültigen Farben der Objekte bzw. Fragmente pro Pixel. Die zuvor im *Vertex Shader* interpolierten *Vertex*-Attribute, mit denen Licht, Schatten und Farben simuliert wurden, dienen dem *Pixel Shader* als Grundlage. *Pixel Shader Programme* eignen sich besonders gut zur Anpassung von Materialeigenschaften [Sud05].
Der *Geometry Shader* stellt eine relative neue Komponente innerhalb der klassischen *Rendering Pipeline* dar. Er ist daher nur auf aktueller Grafikhardware anwendbar. Die Ausführung in der *Pipeline* erfolgt nach dem *Vertex Shader*. Über die Funktionen des *Vertex Shaders* hinaus, kann der *Geometry Shader* neue Primitve innerhalb der *Rendering Pipeline* erzeugen. Dabei beschränkt sich der *Geometry Shader* auf die Erzeugung einfacher Primitive sowie kleine Modifikationen der Geometrien. Anwendung findet die neuartige Komponente beispielsweise bei der Darstellung von *Shadow Volumes* und ersetzt damit die durch den *Stencil Test* erzeugten flachen Schattenzeichnungen durch Volumenschatten, ohne dass dafür eine seperate Geometrie berechnet werden muss. Weiterhin wird der *Geometry Shader* zur Erzeugung von Haar- oder Fellgeometrie in Echtzeit eingesetzt [NVI12].

Für die Ansteuerung der *Grafik-API* stehen verschiedene höhere *C-basierte* Programmiersprachen zur Verfügung. Zur *OpenGL*-Entwicklung wird *GLSL (OpenGL Shading Language)* eingesetzt. Analog dazu wird für die Programmierung der *Direct3D*-Schnittstelle *HLSL (High Level Shader Language)* verwendet. Zusätzlich werden proprietäre Programmiersprachen der Hersteller verwendet, die durch eine eigene Laufzeitumgebung unabhängig von der verwendeten *Grafik-API* eingesetzt werden können. Zwei dieser unabhängigen Sprachen sind beispielsweise *Cg (C for Graphics)* von *NVIDIA* (vgl. Abb. 3.6) oder *ASHLI (Advanced Shading Language Interface)* von *ATI* respektive *AMD*[1].

[1] http://developer.amd.com/archive/gpu/ashli/Pages/default.aspx

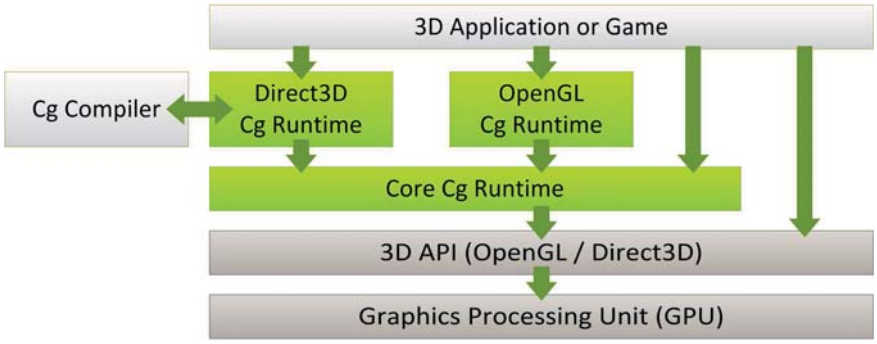


**Abbildung 3.6**: Einbindung es unabhängigen *Cg Shaders.* Angelehnt an [FK03].

## 3.4 Game Engine

Die Nachbildung der Realität in virtuellen Welten ist sehr aufwändig. Stets steht bei kommerziellen Projekten der Zeitaufwand für die Erstellung im Vordergrund, an dem unweigerlich die Wirtschaftlichkeit des Projektes geknüpft ist. Viele der Funktionalitäten innerhalb einer interaktiven Anwendung, eines Spiels oder einer Simulation müssen vielfach verwendet und implementiert werden. Die fristgerechte Erstellung und Bearbeitung virtueller Welten kann oft nur durch viele Entwickler und Editoren realisiert werden. Dazu müssen sie einerseits in der Lage sein zusammen zu arbeiten und gemeinsam zu entwickeln, anderseits ist oft auch eine unabhängige Bearbeitung erforderlich. Es muss also eine Entwicklungs- und Laufzeitumgebung exisiteren mit der all diese Anforderungen zumindest ansatzweise erfüllt werden können. Als Grundlage für die Entwicklung einer virtuellen Umgebung kann eine *Game Engine* dienen. Mit vielen vorgefertigen Modulen und Funktionen stellt eine *Game Engine* die Basis für die Entwicklung von virtuellen Umgebungen dar (vgl. Abb. 3.7). Meist ist für die Grafik eine interne *Grafik-Engine* implementiert.


**Abbildung 3.7**: Schichtenmodell von 3D-Echtzeitanwendungen mit modularer *Game Engine.* Angelehnt an [LJ02].

Mittlerweile existieren zahlreiche *Game Engines* mit verschiedensten Lizenzmodellen auf dem Markt. Von kostenlosen *Open Source* Systemen bis hin zur kommerziellen Software mit verschlossenem Quellcode ist alles zu finden. Der Großteil der Lösungen bringt bereits notwendige Basismodule mit, wie *Grafik-Engine*, Steuerung, Physik- und Soundsystem. Bei den bekanntesten *Engines* handelt es sich um *Unreal Development Kit (UDK)*[2] von *Epic Games*, die *CryEngine*[3] von *CryTek* oder *Unity3D*[4] der Firma *Unity Technologies*. Die *Unity Engine* ist eine der meist verbreiteten *Engines*, deren große Beliebtheit auf eine ausführliche Dokumentation des Systems, die intuitive Bedienung und die plattformübergreifenden Entwicklungsmöglichkeiten zurückzuführen ist. Aus diesen Gründen dient die *Unity Engine* in dieser Arbeit als Grundlage für die Entwicklung einer Softwarelösung zur prozeduralen Erzeugung von 3D-Stadtmodellen.

### 3.4.1 Unity3D

Bei der Entwicklung einer Anwendung in *Unity* gilt die zwei grundlegende Modi *Editor Mode* und *Play Mode* zu unterschieden (vgl. Abb. 3.8). Der *Editor Mode* bzw. *Scene View* dient zur Erstellung und Bearbeitung der 3D-Szenerie. Alle *Game Objects* in der Szene können selektiert und beliebig positioniert werden. Die Bearbeitung mittels *Scene View* stellt die wichtigste Funktion innerhalb der *Unity Engine* dar[5]. Im *Editor Mode* wird das Verhalten zur Laufzeit des Programms durch Konfiguration der *Game Objects* oder Skripte festgelegt. Der *Play Mode* bzw. *Game View* stellt die tatsächliche Laufzeit des Programms dar. Der Blick im *Game View* ist abhängig von der verwendeten Kamera. Es können nur die vorher festgelegen Verhalten und Funktionen ausgeführt werden. Änderung in der 3D-Szene oder Einstellungen der *Game Objects* während des *Play Mode* werden nur temporär gespeichert. Bestimmte Methoden und Funktionen stehen ausschließlich zur tatsächlichen Programmlaufzeit (*Play Mode*) zur Verfügung. Analog dazu ist die Ausführung von *Editor Skripten* im *Play Mode* nicht möglich.

Für die Programmierung von Logiken und Skripten bietet *Unity* die Unterstützung für *JavaScript*, *Boo* und *C# (C-Sharp)*. Bei *JavaScript* handelt es sich um eine klassenlose objektorientierte Skriptsprache. Innerhalb von *Unity* können mittels *JavaScript* oder *Boo* sämtliche zur Verfügung stehenden internen Funktionen eingebunden werden, allerdings können diese Funktionen nur eingeschränkt erweitert werden. Die Klassenlosigkeit verhindert darüber hinaus eine strukturierte Programmentwicklung innerhalb umfangreicher Projekte. Bei *C#* handelt es sich um eine objektorientierte Programmiersprache des *.NET-Frameworks* von *Microsoft*. Das *.NET-Framework* bezeichnet eine standardisierte Software-Plattform die als Basis zur Entwicklung von Anwendungen dient[6]. Mit *C#* können alle *Unity*-internen Funktionen eingebunden und uneingeschränkt erweitert

[2] http://udk.com/
[3] http://mycryengine.com/
[4] http://unity3d.com/
[5] http://docs.unity3d.com/Documentation/Manual/LearningtheInterface.html
[6] http://msdn.microsoft.com/de-de/netframework/aa496123.aspx

werden. Innerhalb des *.NET-Frameworks* ist darüber hinaus der Zugriff sämtlicher Systemfunktionen möglich. Durch die Programmierung innerhalb von *Klassen* kann eine strukturierte Entwicklung gewährleistet werden. Angesichts dieser Vorteile erfolgte die gesamte Entwicklung im Projekt des *Schiffsimulators* mit *C#*.

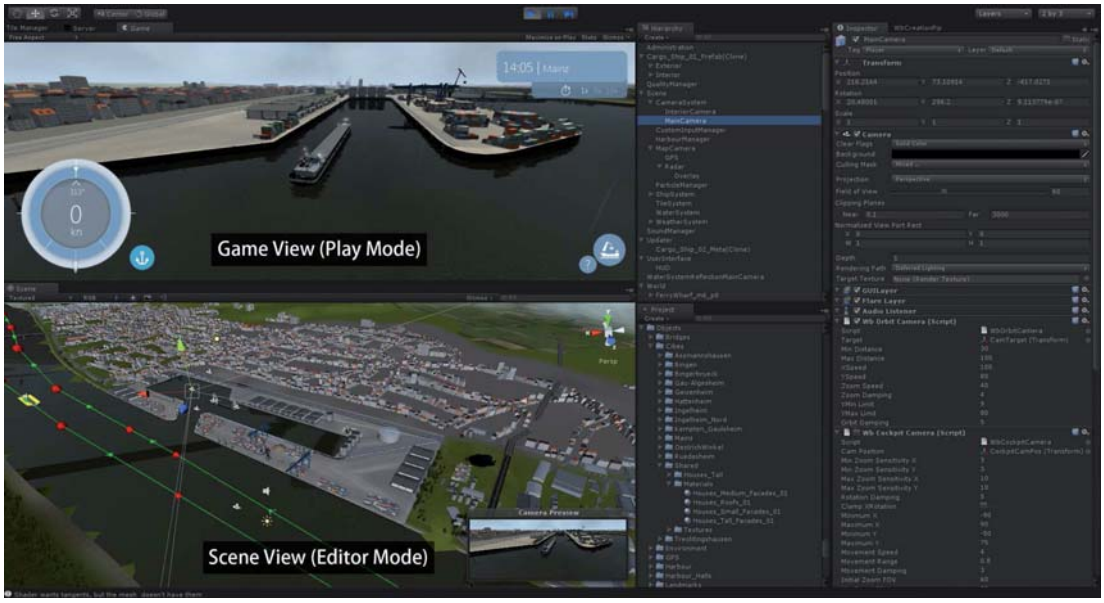


**Abbildung 3.8**: Grafische Benutzeroberfläche der *Unity Game Engine*

## 3.5 Zusammenfassung

In diesem Kapitel wurden Grundlagen der Echtzeitgrafikverarbeitung vermittelt auf denen die folgenden Kapitel aufbauen. Der Aufbau der *Rendering Pipeline*, die Schnittstellen zur progammierbaren Grafikhardware und die eingesetzte 3D-Entwicklungsumgebung *Unity* wurden beschrieben.
Im folgenden Kapitel wird ein Konzept erstellt, welches unter Einbeziehung der Ziele dazu dient eine Softwarelösung zur automatischen Erzeugung von 3D-Stadtmodellen zu entwickeln. Anschließend wird die Realisierung des Konzepts beschrieben.

# Kapitel 4

# Konzeptionierung

In diesem Kapitel wird die Konzeptentwicklung für eine Software zur prozeduralen Erzeugung Echtzeit-optimierter 3D-Stadtmodelle beschrieben. Dazu werden zunächst die Anforderungen an die Softwarelösung definiert und mögliche Optimierungsmethoden, die zur prototypischen Erstellung der Software beitragen, sowie deren Anwendung aufgezeigt. Im darauf Folgenden Kapitel wird die Entwicklung der Software beschrieben. Dafür werden die notwendigen vorbereitenden Maßnahmen erläutert und anschließend die softwareseitige Implementierung der Anwendung im Hinblick auf die vorgangegangen Schritte vorgenommen.

## 4.1 Einleitung

Eine Stadt, im Duden beschrieben als *"größere, dicht geschlossene Siedlung, [...] große Ansammlung von Häusern und öffentlichen Gebäuden [...]"*. Städte und Ortschaften stellen darüber hinaus strukturierte, in verschiedene Bereiche unterteilte Gebiete dar, die je nach örtlichen Begebenheiten untereinander variieren können. Markante geografische, infrastrukturelle oder architektonische Begebenheiten geben einer Stadt ihr einzigartiges Erscheinungsbild. Meist stellen Gebirge, Gewässer, Wahrzeichen, bestimmte Gebäude, Häfen oder Brücken solche eindeutigen Merkmale einer Stadt dar. Abbildung 4.1 zeigt wie der offensichtliche Eindruck einer einzigartigen und wiedererkennbaren Skyline der Stadt Köln ohne markante Bauwerke erlischt. Der Blick eines Betrachters fällt unweigerlich auf den Kölner Dom, das wohl berühmteste und nicht minder dominante Wahrzeichen der Stadt. In der Regel genügt ein kurzer Blick auf das exponierte Objekt im Bild, um das Bauwerk als Kölner Dom und dementsprechend die Stadt Köln zu identifizieren. Die übrigen Bauten im Hintergrund, wie der Fernsehturm oder der *Mediapark* mit dem prägenden *Kölnturm*, werden einem oftsfremden Betrachter nicht zur Identifizierung der Stadt ausreichen. Nach der Fotomontage ist es offensichtlich, dass der größte Teil der Stadt eine *anonyme Masse* an Gebäuden darstellt. Sofern die markanten städtischen Begebenheiten erhalten bleiben, kommt bei Austausch dieser Häuser kaum der Eindruck der Veränderung auf. Für eine automatische Stadtgenerierung kann

dieser Umstand genutzt werden, um innerhalb eines angemessenen Zeitrahmens ohne Anspruch auf Echtheit den Großteil der Stadtgebäude automatisiert zu erstellen.

**Abbildung 4.1**: Oben: bearbeitetes Bild, Stadt ohne Wahrzeichen kaum erkennbar. Unten: Markante Skyline der Stadt Köln[1]

Mit der Umsetzung eines prozeduralen Verfahrens zur Stadtgenerierung lassen sich Großteile einer Stadt automatisiert und zeitsparend erstellen. Damit das Verfahren uneingeschränkt für verschiedene Städte oder Gebiete verwendbar ist, kann die Generierung örtlicher oder spezifischer Merkmale für eine Lösung nicht berücksichtigt werden. Hervorstechende Bauwerke und Begebenheiten werden im Projekt des *Schiffsimulators* manuell erstellt.

[1] Bild: Raimond Spekking (CC-BY-SA-3.0)

## 4.2 Anforderungen an die Softwarelösung

Besonders im Bereich der Echtzeitgrafik ist eine exakte Definition der Anforderungen an die Softwarelösung erforderlich. Durch viele leistungsbedingte Einschränkungen innerhalb einer Echtzeitgrafikanwendung muss oft eine zweckmäßige spezifische Problemlösung entwickelt werden. Trotzdem soll eine Anwendung flexibel, erweiterbar sowie unabhängig von projektspezifischen Gegenheiten nutzbar sein. Darüber hinaus soll sich der notwendige Arbeitsaufwand in einem angemessenen Rahmen bewegen.

Aus diesen Ansprüchen resultieren demnach folgende Anforderungen:

- Optimierung des Arbeitsaufwandes
  Für die Herstellung komplexer 3D-Modelle innerhalb eines adäquaten Produktionszeitraums soll eine Softwarelösung entwickelt werden. Sie soll die prozedurale Erstellung sich wiederholender 3D-Assets realisieren und so den erforderlichen Arbeitsaufwand optimieren.

- Generierung von 3D-Assets
  Die Softwarelösung zur verfahrensmäßigen Erstellung von 3D-Daten soll innerhalb der 3D-Entwicklungsumgebung *Unity3D* erfolgen. Die Software ist nicht durch die Generierung bestimmter Modelle festgelegt. Unabhängig von Stadtmodellen ist die prozedurale Erstellung beliebiger anderer Assets umzusetzen, wodurch das Tool flexibel eingesetzt werden kann.

- Echtzeitgrafikfähigkeit der generierten Modelle
  Bei der Verwendung der erstellen 3D-Modelle ist eine performante Echtzeitgrafikverarbeitung zu gewährleisten. Resultierend aus parallel zur Softwareentwicklung druchzuführenden Performancetests ist eine permanente Optimierung der Modelle vorzunehmen.

- Steuerbarkeit der räumlichen Verteilung der Einzelmodelle
  In Bezug auf Stadtmodelle ist die Kontrolle über die Platzierung und Verteilung der einzelnen Assets innerhalb des Gesamtstadtmodells zu ermöglichen. Die räumliche Einteilung in verschiedene Stadtgebiete ermöglicht eine realitätsnahe Nachbildung bestimmter Städte. Die Platzierung der Gebäude und Anlagen soll zufällig oder anhand bestimmter Vorgaben, wie beispielsweise einem Straßennetzwerk, erfolgen.

- Projektunabhängige Verwendbarkeit
  Die Softwarelösung ist so zu realisieren, dass sie jederzeit ohne Weiteres in anderen Projekten innerhalb der *Unity Engine* Anwendung finden kann. Eine modular aufgebaute Struktur soll die die Einbindung von projektabhängigen Inhalten zulassen. Gleichzeitig soll anhand der Modularität gewährleistet sein, dass die Software den Gegebenheiten anderer Projekte angepasst werden kann und die Generierung von 3D-Daten auch unabhängig von projektspezifischem Inhalt erfolgen kann.

- Erweiterbarkeit
  Durch die einfache Einbindung von 3D-Assets soll ein Benutzer das Tool auf einfache Weise um beliebigen Inhalt erweitern können. Dabei soll dem Benutzer die Anzahl, die Komplexität oder die Art der zur Erweiterung heranzuziehenden 3D-Modelle offen stehen.

- Benutzerfreundlichkeit
  Die zweckmäßige Erstellung komplexer Modelle soll mittels einfacher Bedienung des Tools und unkomplizierter Konfiguration der Assets sichergestellt werden. Die Konfiguration zur prozeduralen Erstellung soll dem Benutzer eine angemessene und effektive Bearbeitung ermöglichen. Die Anwendung soll in die grafische Benutzeroberfläche des *Unity Editor* eingebunden werden sowie die in *Unity* übliche Bedienung umsetzen.

## 4.3 Optimierungsmethoden

Um die zuvor definierten Anforderungen an die Echtzeitgrafik erfüllen zu können werden im Folgenden eine Auswahl in Frage kommender und notwendiger Optimierungsmethoden untersucht. Es werden umsetzbare Methoden in Bezug auf die Entwicklung einer entsprechenden Softwarelösung für die Umsetzung evaluiert und das in Frage kommen der Implementation dieser Methoden zusammenfassend beschrieben.

### 4.3.1 Culling-Verfahren

Das Verfahrensweise des *Culling* (dt. Auslese, Wegwerfen) beschreibt in der Computergrafik das Weglassen von für die Kamera nicht sichtbaren Objekten. Die drei Hauptverfahren des *Culling* teilen sich in *Backface Culling*, *Frustum Culling* und *Occlusion Culling* auf. Andere Verfahren können den jeweiligen Hauptverfahren größtenteils untergordnet werden [Rus08]. Alle genannten Methoden können in der *Untiy Engie* Anwendung finden und werden im Folgenden näher betrachtet.

#### Backface Culling

Das Verfahren des *Backface Culling (BC)* stellt eine einfache Technik dar, welche sich auf einzelne Polygone bzw. Teilflächen von Objekte bezieht. *BC* ist als Standardverfahren in jeder Renderengine enthalten. Hierbei werden in Abhängigkeit zum Betrachtungswinkel einzelne Polygone vom Rendern ausgeschlossen. Polygone deren Flächennormale der Kamera abgewandt ist werden nicht gerendert [Rus08].
Ist der Winkel zwischen Flächennormale des Polygons zum Punkt der Betrachtung größer als 90 Grad wird das Polygon vom Rendervorgang ausgeschlossen. Mit diesem Verfahren wird die Anzahl der zu rendernden Polygone um etwa 50% reduziert.

Das Polygon ist nicht sichtbar, solange $n \cdot (e - p) \leq 0$. Wobei $n$ der Flächennormale, $p$ einem Punkt des Polygons und $e = [e^x, e^y, e^z]^T$ dem Betrachtungspunkt in Weltkoordinaten entspricht [JC98].

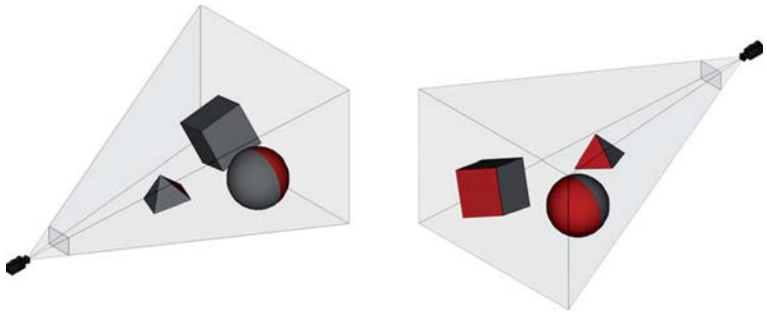

**Abbildung 4.2**: Backface Culling, rot-markierte Polygone werden nicht gerendert

### Frustum Culling

Anders als beim *Backface Culling* bezieht sich das Verfahren des *Frustum Culling (FC)* oder *View Frustum Culling* auf ganze Objekte bzw. Gruppen von Polygonen [AMHH08]. Es gehört ebenfalls zu den Standardverfahren jeder Renderengine. *FC* beschreibt das Entfernen aller Objekte außerhalb des Kamerafrustums, also dem Sichtbereich der Kamera, die keinen Einfluss auf das gerenderte Bild haben. Spiegelungen bzw. Reflektionen stellen dabei eine Ausnahme dar. Hier kann durch das Hinzufügen einer zusätzlichen virtuellen Kamera, durch das Erstellen eines geeigneten Sichtvolumens für jede Spiegelung, Abhilfe geschaffen werden. Dabei kommt das Verfahren erneut zum Einsatz [Rus08].
*FC* ist effizient ausführbar und führt zum Ausschluss großer Teilbereiche der gesamten Szene vom Rendervorgang, woraus ein großer Performancegewinmn resultiert. Es wird aufgrund der einfachen Berechnung und der hohen Entfernungsquote stets als eines der wichtigsten Verfahren implementiert.

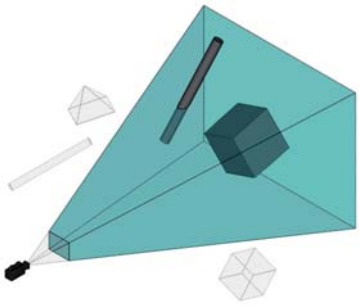

**Abbildung 4.3**: View Frustum Culling, Objekte außerhalb des Kamerafrustums werden nicht gerendert

Zur weiteren Vereinfachung der Berechnung und damit zur Performancesteigerung werden *Bounding Volumes (BV)* eingesetzt. Dabei werden komplexe Objekte mit einfachen geometrischen Primitiven umschlossen. Diese *BV* werden hierarchisch aufgebaut. Ein *BV* eines Objekts schließt auf diese Weise alle dem Objekt unterordneten Teilobjekte mit ein. Durch diese *Bounding Volume Hierarchy (BVH)* wird das *Frustum Culling* effektiv umgesetzt, da die Nichtsichtbarkeit gesamter Teilbäume nicht weiter untersucht werden muss [Rus08]. Vorm Rendern müssen alle *BV* im *Scene Graph* auf Position untersucht werden. Befindet sich ein *BV* mit allen Unterobjekten komplett innerhalb des *View Frustum* wird es gerendert. Analog dazu wird es außerhalb des *View Frustum* mit all seinen *Children* verworfen. Ein *BV* kann jedoch so im Raum liegen, dass es mindestens eine Fläche des Frustums schneidet. Handelt es sich bei dem schneidenden Objekt um das hierarchich obere Objekt wird der gesamte Teilbaum gerendert. Schneidet lediglich ein untergeordnetes Objekt das Frustum wird der gesamte Teilbaum nicht zum Rendervorgang herangezogen [AM99].

### Occlusion Culling

Als *Occlusion Culling (OC* werden Verfahren bezeichnet, die zur Erkennung von Verdeckungen dienen. Damit kann vermieden werden, dass verdeckte und somit ohnehin nicht sichtbare Objekte bzw. Polygone gerendert werden. Das ermöglicht eine weitere erhebliche Leistungssteigerung. Im Gegensatz zum *Backface* oder *Frustum Culling* ist diese Methode relativ aufwändig.

Für die Entstehung eines Bildes müssen die Pixel aller Objekts gezeichnet werden. Dabei werden zunächst die von der Kamera am weitesten entfernten Objekte gerendert, welche schließlich durch die nahesten Objekte verdeckt werden. Befinden sich aus Sicht des Betrachtungspunktes viele Objekte hintereinander müssen diese Pixel mehrmals gezeichnet werden, der soganennate *Overdraw*. Lediglich das letztmalige Zeichnen, des Pixels eines opaken Objekts, ist für das entstehende Bild relevant. Dieser *Overdraw* wird minimiert, indem die Tiefenkomplexität einer Szene mit Hilfe des *Z-Buffer* ermittelt wird. Damit ist bekannt welche Polygone verdeckt werden und nicht gerendert werden müssen.
Ein besonderer Fall besteht bei transparenten Objekten. Dazu wird mittels *Transparency Culling* versucht Objekte zu entfernen, welche aufgrund der Verdeckung transparenter Objekte kaum zu erkennen sind. Mit einem durch den Alphawert definierten Schwellwert wird die Durchsichtigkeit festegelegt. Bei Unterschreitung des Schwellwerts wird das verdeckte Objekt nicht gerendert. Bei dieser Methode müssen leichte Bildfehler in Kauf genommen werden [Rus08].

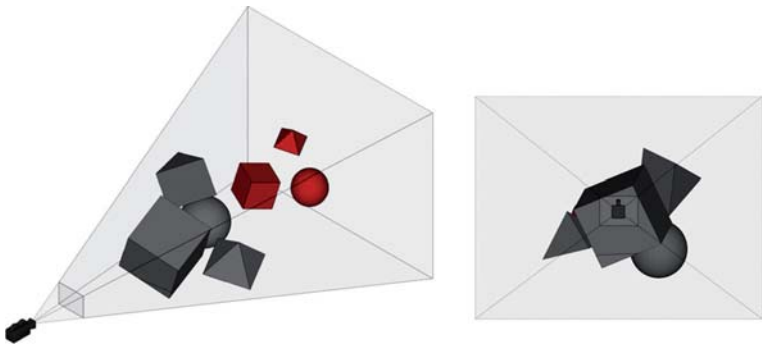

**Abbildung 4.4**: Occlusion Culling, rot-markierte Objekte werden nicht gerendert, rechts: Blick aus Sicht der Kamera

Angesichts des aufwändigen Verfahrens kommt die *Unity Engine* nicht ohne eine Berechnung vor der Laufzeit für das *OC* aus. Mit Hilfe einer virtuellen Kamera erstellt die *Engine* bei der Vorberechnung eine Hierarchie potentiell sichtbarer Objekte für jede Kameraposition. Dieser Vorgang muss für jede Kamera durchgeführt werden, die zur Laufzeit verwendet werden soll. Zur Laufzeit wird diese Hierarchie von jeder Kamera auf sichtbare Objekte abgefragt und die einzelnen Objekte jeweils ein- oder ausblendet.

Dadurch wird die Anzahl der notwendigen *Draw Calls* in der Szene stark reduziert. Die Daten in der Hierarchie sind in Zellen unterteilt. Jede Zelle stellt ein Unterobjekt eines gesamten *Bounding Volumes* der Szene dar, ähnlich einer Baumstruktur. *Unity OC* nutzt zwei dieser Baumstrukturen. Eine zum Speichern der *View Cells* der statischen Objekte. Die zweite für die *Target Cells* aller beweglichen Objekte. Die *View Cells* werden mit einem Sichtbarkeitsstatus indiziert, welcher die exakte Ein- oder Ausblendung der Objekte regelt[2].
Dieses Verfahren ist in *Unity* nur unter bestimmten Voraussetzungen möglich. Zum einen müssen alle Objekte, die für das *OC* in Frage kommen sollen, statisch sein. D.h. sie dürfen unter keinen Umständen bewegt werden. Und zum anderen ist es dadruch nicht möglich *Meshes* von Objekten zusammenzufassen, da sie einzeln ein- bzw. ausgeblendert werden müssen.

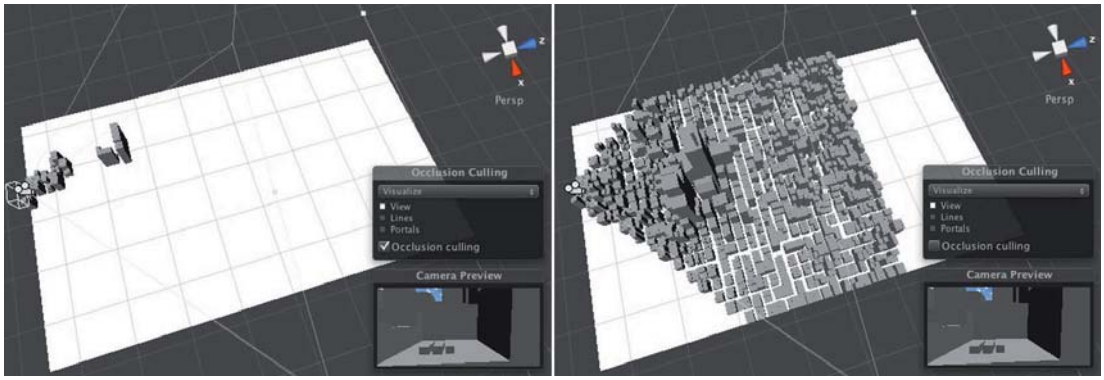


**Abbildung 4.5**: Vergleich in *Unity*; links: gerendete Szene mit *Occlusion Culling*, rechts: gerendete Szene ohne *Occlusion Culling*[3]

Aufgrund dieser Voraussetzungen ist die Anwendung des Verfahrens im *Schiffsimulator* aus folgenden Gründen nicht möglich. Für das Spiel wird ein extrem großes Terrain von über 300 km Länge verwendet. Das Gelände wurde in einzelne quadratische *Terrain Tiles* unterteilt, die aneinander gereiht das Gesamtterrain bilden. In diesem Terrain sind große Postionswerte im 3D-Raum notwendig, die die 100.000 weit überschreiten. Die Positionswerte werden als 4 Byte Fließkommazahl gespeichert. Je größer ein Postionswert wird, desto ungenauer wird er. Für sämtliche beweglichen Objekte im *Schiffsimulator* bedeutet dies eine ungenaue und fehlerhafte Platzierung oder Steuerung. Ähnliche Fehler treten bereits bei der Erstellung und Bearbeitung des Spiels im *Editor Mode* auf. Um diese Fehler zu umgehen werden alle Objekte inklusive Terrain dynamisch zur Laufzeit stets auf Position $(0, 0, 0)$ gesetzt. Für die Bearbeitung im *Unity Editor* wird dieser Vorgang ebenfalls ausgeführt. Da resultierend aus diesem Fehler alle in den Szenen befindlichen Objekte bewegt werden müssen, kann keines der Objekte statisch gesetzt werden.

[2] http://unity3d.com/support/documentation/Manual/Occlusion%20Culling.html
[3] Bilder: http://unity3d.com/support/documentation/Manual/Occlusion%20Culling.html

Hinzu kommt, dass die für das *OC* notwendigen einzelnen Objekte bei der Menge der Gebäude und der Größe der Städte eine unverhältnismäßig hohe Anzahl der entstehenden *Draw Calls* bedeutet.

### 4.3.2 Draw Call Batching

Damit ein Objekt auf dem Bildschirm gezeichnet wird, muss die 3D-Engine den *Draw Call* an das Grafik-Interface senden. Ein *Draw Call* bezeichnet einen Aufruf an die Grafikkarte. Jeder einzelne *Draw Call* benötigt eine bestimmte Verarbeitungszeit um innerhalb der Grafik-API durchgeführt zu werden. Daher nimmt jeder *Draw Call* einen bestimmten Teil CPU-Leistung in Anspruch[4].

Innerhalb der *Unity Engine* beansprucht ein *Game Object* oder ein *Material* je einen *Draw Call*. In der *Engine* kann mit Hilfe des *Batching* eine große Anzahl einzelner *Game Objects* zu einem bzw. wenigen Objekten zusammengefasst werden. Je mehr *Game Objects* zusammengefasst werden, desto geringer die Anzahl der notwendigen *Draw Calls* und desto größer der Performancegewinn. Bei der steigenden Komplexität einer Szenerie sind oft zusätliche Aufrufe erforderlich, beispielsweise für Echtzeitschatten [Wag11].
*Unity* sieht für das *Draw Call Batching* mehrere Methoden vor. Mit dem im *Standard Assets Package* mitgelieferten *CombineChildren*-Skript können *Game Objects*, die sich innerhalb der *Scene Hierarchy* einem *Parent Object* unterordnen, zusammengefasst werden. Das Skript fasst dabei jeweils alle Objekte zusammen, die das selbe *Material* verwenden. Demnach kann die Zahl der *Game Objects* innerhalb einer Szene theoretisch auf die Menge der *Materials* reduziert werden. Übersteigt jedoch die Zahl der *Vertices* bzw. *Triangles* des *Combined Mesh* eine bestimmte Größe kann der Gewinn an Peformance nicht aufrecht erhalten werden. Analog dazu erzeugt eine große Anzahl von Objekten mit wenigen *Vertices* ein hohes Aufkommen von *Draw Calls* und führt gleichermaßen zu Leistungseinbußen.

Mit einem *Batch* werden *n Vertices*, eines oder mehrerer Objekte, in einem *Array* zusammengefasst. Jedes dieser *Vertex Arrays* muss von der *CPU* an die *Grafik Pipeline* übertragen werden. Dazu wird der *Batch* im Speicher der Grafikkarte abgelegt. Sobald sich der erste *Batch* vollständig im Grafikspeicher befindet, beginnt die *GPU* mit dessen Verarbeitung. Ist die Anzahl der zu übertragenden einzelnen *Batches* zu groß, entsteht ein Performanceengpass in der Übertragung durch die *CPU*, da sie die *Batches* nicht so schnell übertragen kann, wie sie von der *GPU* verarbeitet werden. Wird der *Grafik Pipeline* hingegen ein zu großes *Array* übergeben kommt diese mit der Berechnung der Geometrie aufgrund der zu vielen *Vertices* nicht nach, sodass auch hier ein Performancengpass entsteht. Diese Performanceengpässe werden als *Bottleneck* bezeichnet.
Für die Optimierung gilt es also die Verarbeitungschritte gleichmäßig zu verteilen um *Bottlenecks* zu vermeiden. Für die *CPU* ist die Anzahl der zu übertragenen *Batches*

[4] http://unity3d.com/support/documentation/Manual/iphone-DrawCall-Batching

entscheidend. Die *GPU* ist hingegen abhängig von der Menge der in einem *Array* enthaltenen *Vertices*. Daraus ergibt sich, dass für die Optimierung nicht die Anzahl der *Triangles* bzw. *Vertices* pro *Batch* entscheidend ist, sondern vielmehr die Menge der *Batches* pro *Frame* die durch das Zusammenspiel von *CPU* und *GPU* berechnet werden kann [Wlo03]. Die ausgeglichende Verteilung der Aufgaben an die Hardware und die daraus resultierende Verarbeitungsgeschwindigkeit sind demnach abhängig von der Leistungsfähigkeit der einzelnen Komponenten.

Bewegliche Objekte, die das selbe *Material* nutzen, werden bis zu einer *Vertex*-Anzahl von insgesamt 900 automatisch in der *Unity Engine* mittles *Dynamic Batching* zusammengefasst. Das *Static Batching* in *Unity* erlaubt es anderseits Geometrien nicht beweglicher Objekte mit weitaus mehr *Vertices* zusammenfassen. Dazu werden die einzelnen Objekte in der Szene ausgeblendet und ein zusätzliches *Combined Mesh* angelegt, welches jeweils auf die ursprünglichen *Meshes* der Objekte verweist. Bei einer sehr großen Menge von Objekten kann das einen immensen Speicheraufwand zur Folge haben und gleichzeitig die Renderperformance negativ beeinflussen[5].
Wie viele *Vertices* in einem *Batch* zusammengefasst werden ist auch abhängig von der benötigten Qualität und der Plattform auf die die Anwendung ausgerichtet ist[6]. Bei der *Unity Engine* liegt der Schwellenwert der maximalen Anzahl an *Vertices* für PC-Anwendungen bei 60.000 pro *Mesh*. Liegt die Zahl darüber verhindert die *Engine* das Rendern des Modells und blendet es bereits im *Editor Mode* aus. Ab dieser Anzahl kann eine angemessene Verarbeitungszeit durch die *Engine* nicht mehr gewährleistet werden. Die Größe zusammengefasster *Meshes* und der nötige Speicheraufwand ist also für die Erstellung komplexer Stadtmodelle und deren Optimierung zu berücksichtigen.

### 4.3.3 Level of Detail

Besonders in der Echtzeitgrafik finden zur Performanceoptimierung oft *Level of Detail*-Systeme (LOD-Systeme) Anwendung. Mit ihnen ist es möglich die Anzahl der *Vertices* und damit die Anzahl der *Draw Calls* zu reduzieren. LOD-Systeme beschreiben die Abstufung des wahrnehmbaren Detailgrads von Modellen in Abhängigkeit zur Kameraentfernung. Dazu werden komplexe Modelle durch simplere Modelle ersetzt. Objekte im Nahbereich sind gut und groß sichtbar und deshalb detailliert modelliert sowie texturiert darzustellen. Bei weit entfernten Objekten sind Details durch den Betrachter kaum erkennbar und können aus diesem Grund vernachlässig werden.
Ein gebräuchlicher Weg für die Auswahl der entsprechenden LOD-Modelle ist die distanzbasierte Selektion. Dazu werden die Modelle bei einer vorher festgelegten Kameradistanz durch die jeweils nächste LOD-Stufe des Modells ausgetauscht. Alternativ zur distanzabhängigen Auswahl kann die LOD-Stufe auch durch die Bestimmung der Abbildungsgröße des Objektes auf dem Bildschirm erfolgen [Feu10]. Dieses Verfahren ermöglicht eine genaue Einteilung der LOD-Stufen, kommt aber aufgrund des relativ

[5] http://unity3d.com/support/documentation/Manual/iphone-DrawCall-Batching
[6] http://unity3d.com/support/documentation/Manual/Modeling%20Optimized%20Characters.html

hohen Berechnungsaufwandes jedes einzelnen Modells in dieser Arbeit nicht in Frage.

*Unity* sieht in der Version 3.4 kein integrietes LOD-System vor. Im Projekt des *Schiffsimulators* wird ein Skript verwendet, welches zur Selektion der LOD-Modelle das distanzabhängige Verfahren umsetzt. Die Modelle werden dafür dem jeweiligen LOD-Layer zugewiesen. Die Einteilung erfolgt in drei Stufen. So stehen ein Layer *LodNear* für den Nahbereich, *LodMid* für mittlere Entfernung und für große Distanzen ein *LodFar*-Layer zur Verfügung. Zusätzlich und speziell für den *Schiffsimulator* werden Objekte in unmittelbarer Flussnähe einem LOD-Layer zugewiesen, welcher auch deren Reflektionen im Fluss abstandsabhängig rendert. Flussnahe Objekte wie bspw. Brücken, Bäume oder Hafenkräne müssen zur Laufzeit doppelt gerendert werden. Gerendert wird das eigentliche Objekt und die Reflektion des Objektes im Fluss. Durch die maximale Vereinfachung der Reflektionsobjekte auf eine grobe Silhouette kann der Reflektionseffekt beibehalten und gleichzeitig weitere Polygone eingespart werden [Wag11]. Für die Städte ist der Mindestabstand zum Fluss der Modelle entscheidend. Kann die Grafikleistung bei doppeltem Rendervorgang aufrecht erhalten werden, können zumindest die flussnahen Häuserreihen als Flussreflektionen gerendert werden.

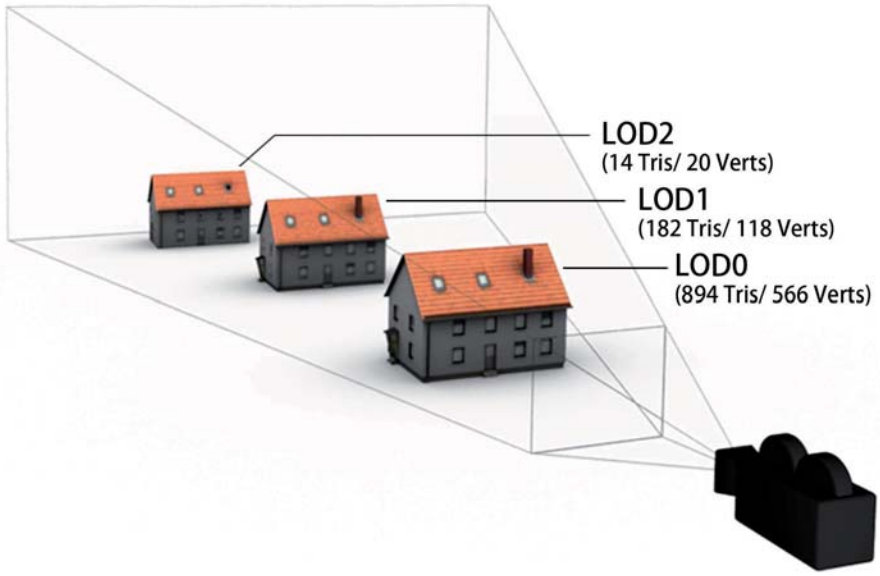


**Abbildung 4.6**: Beispiel eines Level of Detail

Die Verwendung verschieden aufgelöster Texturen für die verschiedenen LOD-Modelle kann unter Umständen die Inanspruchnahme des Texturspeichers der Grafikkarte zusätzlich verringern. Bei Objekten einer niedrigeren LOD-Stufe, also der größeren Entfernungen, kann auf Details in der Textur verzichtet werden. Werden Texturen kleiner als in der ursprünglichen Größe dargestellt, beschleunigt das die Berechnung des Bildes. Allerdings können dabei *Aliasing*-Effekte auftreten, die die Bildqualität verschlechtern.

Das Bild wird dabei lediglich um entsprechend viele Pixelreihen und -spalten reduziert, was unschöne Kanten zur Folge haben kann. Das Verfahren des *Mip Mapping* versucht das Auftreten solcher Effekte zu vermeiden. Dafür wird je *MipMap*-Stufe eine halb so große Versionen des Bildes gespeichert, bei dem das Pixelraster durch *Downsampling* neu berechnet wurde. Der benötigte Texturspeicher für die gesamte Textur erhöht sich dabei um maximal ein Drittel im Vergleich zum ursprünglichen Bild [NVI12]. In der *Unity Engine* ist dieses Verfahren standardmäßig integriert.

### 4.3.4 Texturen

Durch Texturen wird versucht das Detail eines 3D-Modells zu erhöhen. Bei sehr einfachen Modellen, die aus wenigen Flächen bestehen, vermittelt erst die Textur das gewünschte Erscheinungsbild. Dabei bilden Texturen nicht nur farbliche Details als *Diffuse Color Map* ab, sondern ermöglichen auch die Nachbildung geometrischer Details in Form von *Normal Maps* oder visualisieren glänzende oder leuchtende Flächen in Form von *Specular* bzw. *Illumination Maps*. In der Echtzeitgrafik wird der vermittelte Tiefeneindruck eines einfachen Modells oft durch Schattenbildungen unterstützt, die durch *Ambient Occlusion Maps* realisiert werden können. Ein Modell kann zur Darstellung demnach mehrere Texturen benötigen.

Die Auflösung und Farbtiefe einer Textur sind für die Echtzeitverarbeitung maßgeblich entscheidend. Aus ihnen resultieren die Anzahl der zu verarbeitenden Pixel und der notwendige Speicherbedarf: $Pixel_x \cdot Pixel_y \cdot Bit_{Farbtiefe} = Byte_{Speicherbedarf}$
Zur Verarbeitung wird die Textur an die Grafikkarte übertragen, wo sie im *Texture Cache* abgelegt und durch die *GPU* bei Bedarf abgerufen und verarbeitet wird. Mit der Auflösungs- und Speichergröße einer Textur steigt auch die Verarbeitungszeit. Je größer die Auflösung einer Textur ist, desto mehr Pixel müssen zur Darstellung abgetastet werden. Analog dazu dauert die Übertragung einer Textur im hardwareinteren Bussystem mit steigender Speichergröße länger. Hinzu kommt dass die Größe des Texturspeichers der Grafikkarte begrenzt ist. Ist er voll müssen die Texturen in den Hauptspeicher ausgelagert werden, was die Verarbeitungszeit erheblich verlängert.
Um diesem Problem entgegen zu treten muss die Zahl der Texturen minimiert und deren Auflösungsgröße optimal ausgenutzt werden. Oft können Polygone eines Modells mit den selben Teilen einer Textur versehen werden, sodass große oder sich wiederholende Flächen mit einem kleinen Texturteil abgedeckt werden können. Die Texturen müssen dazu bei wiederholen *kachelbar* sein, also nahtlos aneinander passen. So kann viel Platz auf einer Textur eingespart werden.

Um farbige Texturen anzuzeigen müssen die Bilddateien mit einer Farbtiefe von 24 Bit gespeichert werden. Eine solche Datei setzt sich aus den Farbkänalen Rot, Grün und Blau zu je 8 Bit zusammen. Durch Verwendung des Alphakanals steht ein vierter zusätzlicher 8 Bit Farbkanal zur Verfügung. Dieser wird oft zur Regulierung von Transparenz zu hilfe genommen. Für die Darstellung von Graustufen ist lediglich ein 8 Bit Farbkanal erforderlich. Die Notwendigkeit des Einsatzes einer 32 Bit RGBa Bilddatei

mit einem erhöhten Speicherbedarf ist von der Verwendung des Modells abhängig.
An jedem *Mesh* eines Modells lassen sich *Vertex Colors* mit einem 24 Bit Farbraum und einem zusätzlichen 8 Bit Alphakanal speichern. In Verbindung mit einer Graustufen-Textur, mit der diese Farben verrechnet werden können, besteht die Möglichkeit durch diese speicherfreundlichen Texturen trotzdem farbige Modelle zu rendern.

Mit dem *Direct Draw Surface*-Format (DDS-Format) steht ein speziell für die Speicherung von Texturen entwickeltes Dateiformat zur Verfügung. Mit diesem Format ist die Speicherung von Bilddateien ab einer Farbtiefe von 1 Bit (2 Farben) möglich. *DDS*-Dateien können schneller in den Grafikspeicher geladen und dort verarbeitet werden. Das Format wird oft zum Speichern von *MipMaps* oder *Cube Maps* verwendet, da aufgrund der sparsamen Speicherung alle Bilddaten in einer statt in verschiedenen Dateien gespeichert werden können [NVI12]. Für *Unity* wird offiziell keine Unterstützung des Formats angegeben[7]. Intern nimmt die *Engine* allerdings die Speicherung von Bilddateien im *DDS*-Format vor.

Anwendung finden die Texturen eines Modells durch ein *Material* innerhalb der *Unity Engine*. Pro *Material* ist ein *Draw Call* zur Verarbeitung notwendig. Um die Anzahl der Aufrufe an die Grafikkarte zu minimieren, können die einzelnen Texturen zu sogenannten *Texture Atlases* zusammengefasst werden. Auf einem Atlas sollten Texturen von Modellen zusammengefasst werden, deren gemeinsame Auftrittswahrscheinlichkeit in der 3D-Szene am größten ist. Die Auflösung eines solchen Atlases und die damit verbundene Qualität ist abhängig vom Einsatz der Modelle. Üblich sind Auflösungen von 1024x1024 oder 2048x2048 Pixeln bei einer Farbtiefe von 24 Bit. Werden auf einem Atlas mehrere optimal gekachelte Texturen von vielen Modellen gespeichert (vgl. Abb. 4.7) kann der Texturspeicherbedarf, sowie die Anzahl der *Draw Calls* auf ein Minimum reduziert werden.

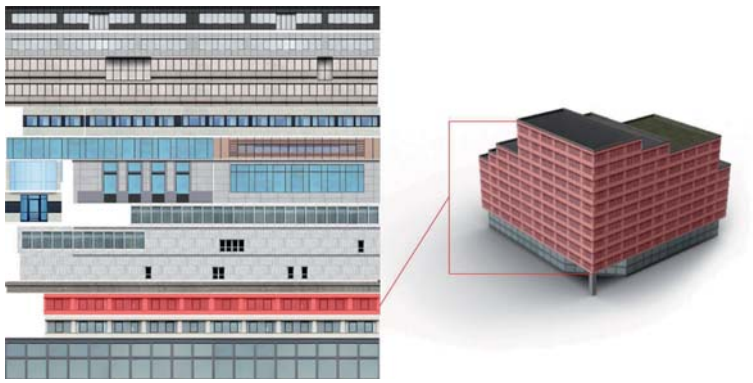

**Abbildung 4.7**: links: Texture Atlas mit gekachelten Texturen; rechts: texturiertes Modell

[7] http://unity3d.com/unity/editor/importing

### 4.3.5 Optimaler Shader

Ein *Shader* beeinflusst die Verarbeitung sämtlicher Daten durch die Grafikkarte maßgeblich. Die Programmierung eines *Shaders* ermöglicht so den Einfluss auf das letztendlich entstehende Erscheinungsbild der 3D-Szenerie.

Bei den beschriebenen Möglichkeiten zur Optimierung der 3D-Modelle und Texturen ist ein spezieller *Shader* zur Umsetzung dieser Maßnahmen erforderlich. Erst durch den *Shader* ist die Verarbeitung der *Vertex Colors* der einzelnen *Meshes* in Verbindung mit den entsprechenden Texturen möglich. Hierfür werden die einzelnen Farbkänale der Bilder ausgelesen. Dadurch können aus einer 32 Bit Bilddatei bis zu vier 8 Bit Texturen ausgelesen und verarbeitet werden. Auf diese Weise können die ohnehin in Graustufen gespeicherten Texturen, wie *Specular*, *Illumination* oder *Ambient Occlusion Map* innerhalb einer Datei zusammengefasst werden. Das erspart unter Umständen Speicher und Verwaltungsaufwand.

### 4.3.6 Zusammenfassung

Die zuvor beschriebenen Methoden zur Minimierung der Verarbeitungszeit und zur Sicherstellung der Grafikleistung zeigen ein großes Optimierungspotential innerhalb *Unitys* auf. Einige dieser Methoden sind bereits standardmäßig in der *Engine* implementiert, wie die Verfahren des *Backface* bzw. *Frustum Culling*. Das *Occlusion Culling*-Verfahren kann angesichts der genannten Gründe speziell für den *Schiffsimulator* nicht eingesetzt werden. Die Umsetzung aller weiteren Optimierungsmethoden ist möglich und notwendig für die Sicherstellung der Gesamtgrafikleistung der Anwendung. Deren zielgerichtete Implementierung in das Projekt des *Schiffsimulators* wird im folgenden Kapitel beschrieben. Dabei wird zunächst das Verfahren ausgewählt, welches zur Stadtgenerierung dienen soll. Anschließend wird auf die daraus resultierenden notwendigen vorzubereitenden Schritte eingegangen und letztlich die Entwicklung des Softwarekonzepts sowie dessen Implementierung beschrieben.

# Kapitel 5

# Konzeptrealisierung

Im vorangegangen Kapitel der Konzeptionierung wurden Anforderungen definiert sowie die Anwendung der Optimierungsmethoden zur Erfüllung der Anforderungen ausführlich beschrieben und bewertet.
In diesem Kapitel erfolgt die Umsetzung des Konzepts. Nach dem eine Verfahrensauswahl getroffen wird, erfolgt die Entwicklung der Software zur automatischen Stadtgenerierung. Dafür werden die notwendigen vorbereitenden Maßnahmen erläutert und anschließend die softwareseitige Implementierung der Anwendung im Hinblick auf die vorgangegangen Schritte vorgenommen. Im darauf folgenden Kapitel werden die Ergebnisse dieses Kapitels aufgezeigt und durch Performancetests evaluiert.

## 5.1 Verfahrensauswahl

Die im letzten Teil des Kapitels detailliert beschriebenen in Frage kommenden Optimierungsmethoden innerhalb der eingesetzten *Unity Game Engine* ermöglichen die Sicherstellung der Echtzeitgrafikleistung der Anwendung, den definierten Anforderungen entsprechend. Damit diese Maßnahmen angewendet werden können und eine zielgerichte Lösung entwickelt werden kann muss vor Umsetzung die weitere Vorgehensweise festgelegt werden.

Für die Erstellung der einzelnen Assets innerhalb des Stadtmodells stehen zwei grundverschiedene Erstellungsmethoden zur Verfügung. Einerseits ist es möglich vormodellierte sowie texturierte Einzelmodelle oder Modellteile zur Stadterzeugung heran zu ziehen. Diese werden nach bestimmten Parametern automatisch erzeugt, passend zusammengesetzt und innerhalb einer festgelegten Umgebung platziert. Anderseits besteht die Möglichkeit sämtliche *Meshes* der Modelle parameterabhängig dynamisch während des Erstellungsprozess zu erzeugen und einer vorgegebenen Umgebung anzupassen. Ausmaße, Formen und Texturen der Einzelmodelle stehen dann erst nach dem Erstellungsprozess fest.

Beide Verfahren erfordern die Ausführung des prozeduralen Erzeugungsprozesses im *Editor Mode* der *Unity Engine*. Der *Editor Mode* wird zur Erstellung und Bearbeitung der 3D-Szenerie benötigt, unabhängig von der Laufzeit des Programms. Sämtliche Erstellungs-, Veränderungs- und Optimimierungsmaßnahmen, wie sie im folgenden konzeptionell beschrieben werden, finden im *Unity Editor Mode* statt. Die aus dem Konzept resultierenden Ergebnisse sollen Echtzeit-optimierte Stadtmodelle sein, die erst nach Anwendung sämtlicher beschriebenen Methoden Einsatz zur tatsächlichen Laufzeit eines Programms finden können. Für die Entwicklung einer Echtzeitgrafikanwendung ist es besonders wichtig bestimmte Faktoren stets unter Kontrolle zu halten. Bei der prozeduralen Erzeugung von 3D-Inhalt entstehen schnell übermäßig komplexe Modelle, mit denen die geforderte Echtzeitperformance nicht eingehalten werden kann. Für die bestmögliche Kontrolle über Polygonzahlen und Texturen des zu erstellenden Stadtmodells, wird deshalb die Methode der vorgefertigten Modelle näher betrachtet. Bei dieser Methode kann offensichtlich eine maximale Performanceoptimierung erfolgen, bei dennoch ansprechenden visuellen Ergebnissen. Darüber hinaus steht es dem Benutzer bei dieser Methode frei bereits vorhandene, gewöhnliche oder auch individuell angefertige Modelle für die Generierung zu verwenden, ohne einen Mehraufwand an Vorbereitung oder Konfiguration.

Vor der Softwareentwicklung und -implementation müssen also sämtliche prototypischen 3D-Assets, welche für die Stadtgenerierung erfordlich sein sollen, vorhanden sein. Diese Assets bilden die Grundlage für alle folgenden Maßnahmen für die Umsetzung des Konzepts. Im Folgenden wird zunächst die zielgerichtete Erstellung der benötigten Assets beschrieben. Unter Berücksichtigung aller Anforderungen soll mit den 3D-Modellen und Texturen die Anwendung der zuvor beschrieben Optimierungsmethoden ermöglicht werden.

## 5.2 Modellierung

Die Gestaltung der einzelnen 3D-Modelle bzw. Modellteile, welche später in der Stadt zu sehen sein werden, spielen eine zentrale Rolle. Typische Charakteristiken einzelner Assets verleihen dem Gesamtmodell durch ihre Ausmaße und Formen einen bestimmten Stil. Mit der Erstellung der 3D-Modelle werden die grundlegenden stilistischen Eigenschaften der Stadt geschaffen.
Bei Modellen die tausendfach instanziiert werden sollen bestehen mehrere Probleme. Die Anzahl der verwendeten *Triangles* und *Vertices* ist zu minimieren. Dem gegenüber steht der Wunsch nach einem Erscheinungsbild, welches eine gewisse Grundästhetik und einen bestimmten Stil vermittelt. Eine hohe Varianz der Modelle untereinander sorgt für Abwechlungsreichtum.
Grundsätzlich ist der genaue Verwendungszweck der Modelle entscheidend. Der notwendige Detailgrad der Modelle ist abhängig von Entfernungen oder Blickwinkeln der Kamera zum jeweiligen Modell. Soll beispielsweise eine Fahrt durch die Straßen einer

Stadt simuliert werden ist es unter Umständen empfehlenswert den Häuserfassaden mehr Detail zu verleihen als den Dächern. Anders bei einem Simulationsflug über eine Stadt. Hier gilt die Aufmerksamkeit eines Betrachters mehr den Dächern als den Fassaden. Der charakteristische Stil eines Modells innerhalb einer Szenerie kann ebenfalls über den Modelldetailgrad entscheiden. Bei Architekturvisualisierungen wird das Stadtgebiet um das betreffende Objekt oft abstrakt oder sehr vereinfacht dargestellt. Dies dient vorwiegend der Herausstellung des eigentlichen Objekts, vermittelt darüber hinaus aber auch einen bestimmten Stil.

Für die Verwendung der Stadtmodelle im *Schiffsimulator 2012 - Binnenschifffahrt* ist ein Kamerasichtbereich von 5 bis 50 Metern Höhe bei einer Entfernung von 10 bis maximal 2000 Metern vorgesehen. Die Kamerabewegung ist abhängig von der Position des Schiffes auf dem Fluss, d.h. die Kamera wird sich nie im rechten Winkel oberhalb eines Stadtgebiets oder innerhalb eines Straßenzugs befinden.

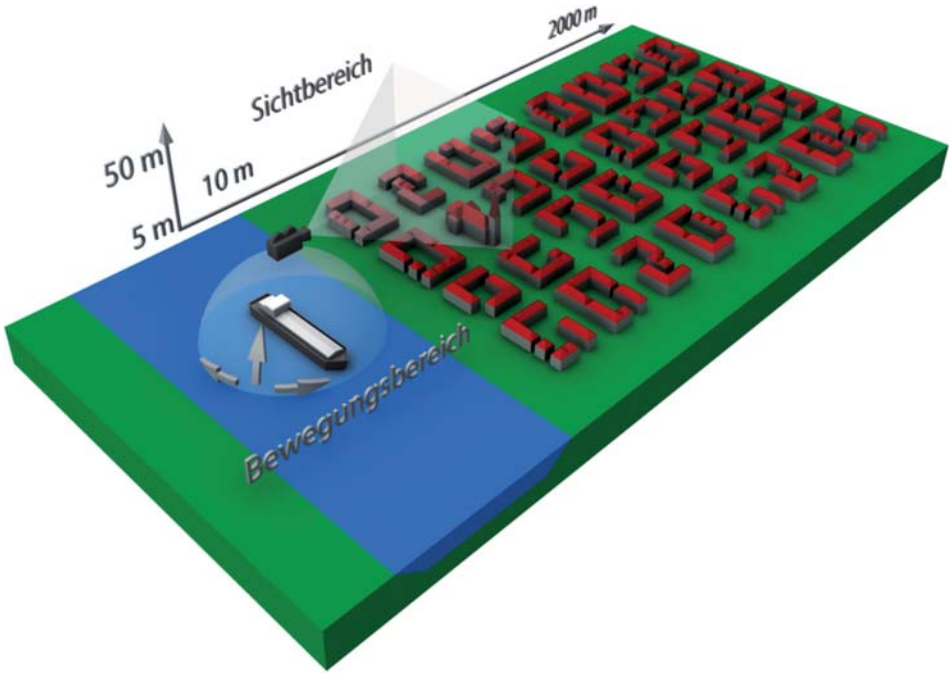


**Abbildung 5.1**: Bewegungs- und Sichtbereich der Kamera

Daraus ergibt sich, dass für die Gebäudefassaden für den Großteil der Stadt kein detailliertes *Mesh* erforderlich ist. Lediglich für die erste Reihe Häuser in unmittelbarer Flussnähe machen detaillierte Fassaden unter Umständen Sinn. Anders bei den Dächern. Fast alle sind sichtbar und tragen größtenteils zum visuellen Eindruck der Stadt bei. Viele verschiedene Dachteile, wie Gauben, Schornsteine, Apparaturen oder Dachfenster, sorgen für Varianz (vgl. Abb. 5.2).

**Abbildung 5.2**: Stadtmodell mit einfachen Fasseden und detailierten Dächern

Um die Assets in einem *Level-of-Detail*-System verwenden zu können ist es notwendig die Modelle mit verschiedenen Detailstufen anzufertigen. Dies betrifft insbesondere die Dächer mit allen Auf- und Anbauten. Die verwendeten Fassadenteile sind wie erwähnt bereits auf ein Minimum an Polygonen herunter gebrochen. Die Teile die bei großer Entfernung auszublenden sind werden als einzelne Objekte gespeichert und als *Child* dem *Parent Object* angehängt. Sie werden später in der *Unity Engine* einem seperaten *LOD-Near-Layer* zugewiesen, der die Objekte nur ab einer geringen Kameraentfernung anzeigt. Die *Parent Objects* werden hingegen dem *LOD-Far-Layer* zugewiesen, welcher Objekte bereits bei hoher Kameradistanz rendert.
Im Rahmen der Arbeit wurden pro LOD-Stufe rund 220 einzelne Gebäudeteile erstellt, woraus sich 150 verschiedene Gebäude eingeteilt in 23 Kategorien zusammensetzen lassen. Angesichts der bereits angewandten *Low Poly*-Modellierung wurden die Modelle in lediglich zwei LOD-Stufen erstellt. Die Anzahl der Stockwerke, die Wahl der jeweiligen Einzelteile, Texturen und Farben soll bei der späteren Generierung zufällig geschehen um ein Höchstmaß an Varianz der Gebäude zu erhalten. Kategorisiert wurden die Gebäude in verschiedenen Größen als Dorf-, Kleinstadt-, Großstadt-, Appartment-, Büro- und Gewerbegebäude sowie Industrieanlagen und Kirchen.

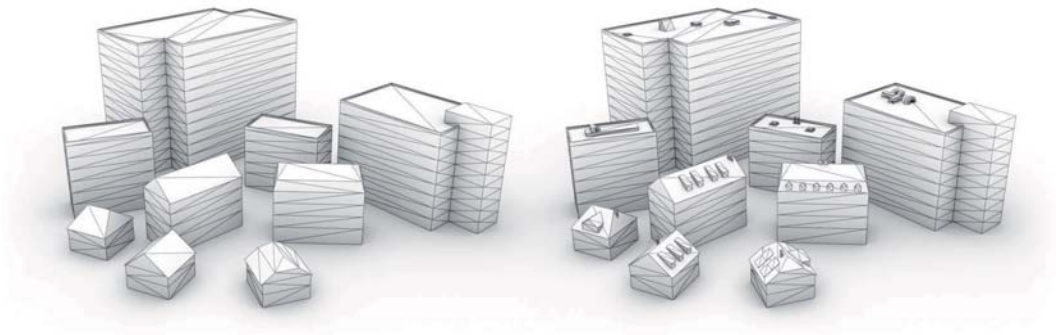

**Abbildung 5.3**: *LOD-Modelle*, links *LOD-Far* (706 Vertices, 722 Triangles), rechts *LOD-Near* (1503 Vertices, 1698 Triangles)

## 5.3 Texturierung

Die Texturierung von Oberflächen bestimmt das *Look and Feel* grafischer Elemente. Die richtige Wahl der Farben, deren Intensität und die Einbindung passender Einzelelemente einer Textur verleihen den 3D-Modellen einen stimmigen Wiedererkennungswert.

Die *Diffuse Color Map* stellt die grundlegenden Texturelemente dar. Bei Häusern sind das hauptsächlich Mauerwerk, Dachziegel, Fenster, Türen und diverse spezifische Details. Eine *Diffuse Color Texture* repräsentiert somit detailliert die farblichen Gegebenheiten einer Oberfläche. Geometrische Details, Oberflächeneigenschaften oder das Verhalten bei verschiedenen Lichtsituationen werden dadurch nicht behandelt.
Die in dieser Arbeit verwendeten sehr einfach gehaltenen Modelle bestehen zumeist lediglich aus wenigen planaren Flächen. Um diesen Modellen mehr Detail zu verleihen stehen verschiedenen Methoden zur Verfügung. Mit Hilfe von *Normal Mapping* ist es möglich den geometrischen Detailreichtum eines Modells durch Schattierungen bei der Beleuchtung zu steigern, ohne dabei die Anzahl der Polygone erhöhen zu müssen. So entsteht auf planaren Flächen unter anderem für Mauerwerk, Fenster- oder Türöffnungen ein Tiefeneindruck. Eine *Ambient Occlusion Map* kann den detaillierten Tiefeneindruck eines einfachen Modells zusätzlich unterstützen. Durch entstehende Verdeckungsschatten an Kanten oder den simulierten geometrsichen Details innerhalb der *Normal Map* wird der Eindruck des Detailreichtums weiter intensiviert (vgl. Abb. 5.4).

Auf einer Textur werden viele verschiedene Materialien dargestellt. Mit der Verwendung einer *Specular Map* können die verschiedenen Glanzeigenschaften der jeweiligen Oberflächen kontrolliert werden. Für die Darstellung in Tag-/Nachtsituationen empfielt sich der Einsatz einer *Illumination Map*. In Abhängigkeit des *Shaders* (siehe Kapitel 5.5.3) sorgt sie für die Regulierung der Helligkeit bestimmter Flächen auf der Textur. So kann beispielsweise der Eindruck von beleuchteten Fenstern oder Lichtern erzielt

werden.

**Abbildung 5.4**: Vergleich *Texture Maps*; links nur mit *Diffuse Color Map*, rechts mit *Diffuse Color, Specular* und *Normal Map*

Bei den Texturen für Echtzweitanwendungen besteht die Anforderung darin, so wenig Texturspeicher wie möglich in Anspruch zu nehmen und gleichzeitig ein ansprechendes und erkennbares Erscheinungsbild des Objekts zu schaffen. Der Speicherbedarf einer Textur berechnet sich aus der Texturgröße in Pixeln und der Farbtiefe in Bit.

Der Gesamtspeicherbedarf aller für ein Modell benötigten Texturen in einer für Echtzeitgrafik üblichen Größe von jeweils 1024 Pixeln Höhe und 1024 Pixeln Breite setzt sich aus den folgenden einzelnen *Texture Maps* zusammen:

*Diffuse Color Map* und *Normal Map*, je 24 Bit (RGB)
jeweils $1024Pixel \cdot 1024Pixel \cdot 24Bit = 3.071kB \approx 3MB$

*Specular*, *Ambient Occlusion* und *Illumination Map*, je 8 Bit (Graustufen)
jeweils $1024Pixel \cdot 1024Pixel \cdot 8Bit = 1.024kB = 1MB$

Ohne Optimierung ergibt sich daraus ein Gesamtspeicherbedarf für die Texturen von rund 9 MB pro Modell. Bei 150 verschiedenen Häusern entspricht das einem Texturspeicherbedarf von etwa 1.350 MB bei der jeweils einzelnen Verwendung aller *Texture Maps*.

Aufgrund der großen Anzahl an Modellen und der relativ hohen Entfernung der Stadt zur Kamera ist es nicht erforderlich jedem einzelnen Modell einen unverhältnismäßig großen Texturplatz in Höhe und Breite von 1024 Pixeln für maximalen Detailgrad einzuräumen. Die *Diffuse Color Maps* von Modellen einer Kategorie werden auf einem *Texture Altas* der Größe 1024x1024 Pixel zusammengefasst. Viele der 3D-Modelle nutzen gleiche oder ähnliche Texturen. Durch die Überlagung der *UV-Layouts* ausgwählter *Faces* der Modelle kann der Raum auf dem *Texture Atlas* effektiver genutzt werden.
Die automatische Auswahl zur prozeduralen Generierung der Häuser soll später per Kategorie erfolgen. Mit den zusammengefassten Texturen befinden sich nur die *Texture Atlases* der Hauskategorien im Grafikspeicher, die auch tatsächlich Anwendung in der Szene finden und ohnehin der Wahrscheinlichkeit nach gemeinsam auftreten werden. Somit wird der Texturspeicher der Grafikkarte stets effektiv ausgenutzt. Nicht verwendete Texturen können aus dem Grafikspeicher entfernt werden und stellen so Speicher für andere Texturen der 3D-Szene zur Verfügung. Weiterhin verbessert sich die Ladezeit, da nur wenige speicherunintensive *Texture Atlases* in den Grafikspeicher geladen werden müssen. Da für jeden *Texture Atlas* nur je ein *Material* in der *3D-Engine* verwendet werden muss wird gleichzeitig die Anzahl der *Draw Calls* an die *GPU* reduziert.
Schlussendlich können die *Diffuse Color Maps* sämtlicher verwendeter Häuser aller Kategorien zu insgesamt 8 *Texture Atlases* zusammengefasst werden.

Zur Optimierung des Texturspeicherbedarfs wird eine vergleichbare Zusammenfassung aller weiteren *Texture Maps* durchgeführt.
Da bei der großen Kameradistanz nahezu keinerlei Details der einzelnen Texturen auffallen und der Speicherbedarf aufgrund der 24 Bit Farbtiefe relativ hoch ist wird zunächst auf die Verwendung von *Normal Maps* für alle Modelle verzichtet. Dadurch ergibt sich nach der Zusammenfassung aller *Texture Maps* eine Ersparnis von 3 MB pro *Texture Atlas* bzw. Hauskategorie.
Insgesamt kann durch die Zusammenfassung der Texturen und das Ausschließen der *Normal Maps* die Inanspruchnahme des Gesamttexturspeichers um ca. 96,5% von 1,3 GB auf rund 48 MB reduziert werden. Bei Anwendung der *Normal Maps* um ca. 94,6% auf rund 72 MB.

Für die 3D-Modelle stehen damit insgesamt 32 Texturen bzw. *Texture Atlases* bereit. Davon entfallen 8 auf *Diffuse Color Maps* zu je 24 Bit Farbtiefe (RGB) und 24 auf *Specular, Illumination* und *Ambient Occlusion Maps* zu je 8 Bit Farbtiefe (Graustufen).
Tests in *Unity* haben gezeigt, dass die *3D-Engine* für jede Textur, unabhänig von der Farbtiefe, stets mindestens eine 24 Bit RGB-Textur im Speicher reserviert. Das bedeutet dass auch den 8-Bit-Graustufentexturen ein Speicher von jeweils rund 3 MB, anstelle von nur 1 MB, bereit gestellt wird. Daraus folgt ein Gesamtspeicherbedarf von 96 MB, wovon 50% nicht genutzt werden. Um dem entgegen zu wirken ist eine weitere Optimierung der Texturen erforderlich.

Die Verwendung des kompletten 24 Bit Farbraums für jede der *Diffuse Color Maps* dient der farblichen Gestaltung der Modelle. So wird jedes Haus die selbe Farbe aus

der Textur erhalten. Um unterschiedlich farbige Häusermodelle zu erhalten werden die *Vertex Colors* der *Meshes* zu hilfe genommen. Dazu kann ein 32-Bit-Farbwert (RGBA) in jedem einzelnen *Vertex* gespeichert werden, ohne den Speicherbedarf für *Mesh* oder *Textur* zu erhöhen. Dieser Farbwert kann mit dem entsprechenden *Shader* (siehe Kapitel 5.5.3) beispielsweise mit der *Diffuse Color Map* multipliziert werden und sorgt so für farbliche Unterschiede beim Rendern der *Meshes*. Für die Multiplikation von Farben empfiehlt sich die *Diffuse Color Map* in Graustufen (8 Bit) zu speichern, wodurch sich der Speicherbedarf der *Diffuse Color Map* von je 3 auf 1 MB verringert. Das Multiplizieren einer Farbe mit einem Grauwert verändert die Helligkeit der Farbe, jedoch nicht den Farbton. Das Multiplizieren zweier Farbwerte kann zu unerwünschten Ergebnissen führen.
Zunächst bewirkt die Multiplikation der *Vertex Colors* mit der *Diffuse Color Map* die Einfärbung der gesamten Textur. Damit nur bestimmte Stellen, wie Fassade oder Dach, eingefärbt werden, ist eine *Maske* notwendig. Diese Maske kann ebenfalls in Graustufen angefertigt werden. Der *Shader* multipliziert den Farbwert der *Vertex Colors* im *Mesh* mit dem Graustufenwert der Maske.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}_{vertex} \cdot v_{mask} = \begin{pmatrix} R \\ G \\ B \end{pmatrix}_{color}$$

wobei $R, G, B \notin [0, 255]$ und $v_{mask} \notin [0, 1]$

Für jede Hauskategorie sind demnach folgende Texturen erforderlich:

- *Diffuse Color Map*, 8 Bit, Graustufen
- *Specular Map*, 8 Bit, Graustufen
- *Illumination Map*, 8 Bit, Graustufen
- *Ambient Occlusion Map*, 8 Bit, Graustufen
- *Vertex Color Mask*, 8 Bit, Graustufen

Mit der Erweiterung einer 24 Bit RGB Textur um einen weiteren 8 Bit Farbkanal auf 32 Bit RGBA können vier dieser jeweils 8 Bit *Texture Maps* in einer Datei gespeichert werden. Der *Shader* (siehe Kapitel 5.5.3) greift dann auf den jeweils benötigten Farbkanal zu. So wird der durch *Unity* ohnehin für eine 24 Bit Textur reservierte Speicher optimal ausgenutzt und gleichzeitig der Verwaltungsaufwand von vorher 40 auf 16 Texturdateien verringert. Da das Detail der *Ambient Occlusion Map* bei der vorliegenden Kameraentfernung im *Schiffsimulator* nur bedingt sichtbar ist, wurde diese *Texture Map* als einzelne Textur gespeichert. Alle anderen Texturen tragen wesentlicher zum Gesamtbild der Modelle bei. Bei einer möglichen notwendigen Performanceoptimierung kann diese Textur dadurch als erste außen vor gelassen werden. Je nach Notwendigkeit kann die Aufteilung der fünf Texturen auch anders erfolgen.

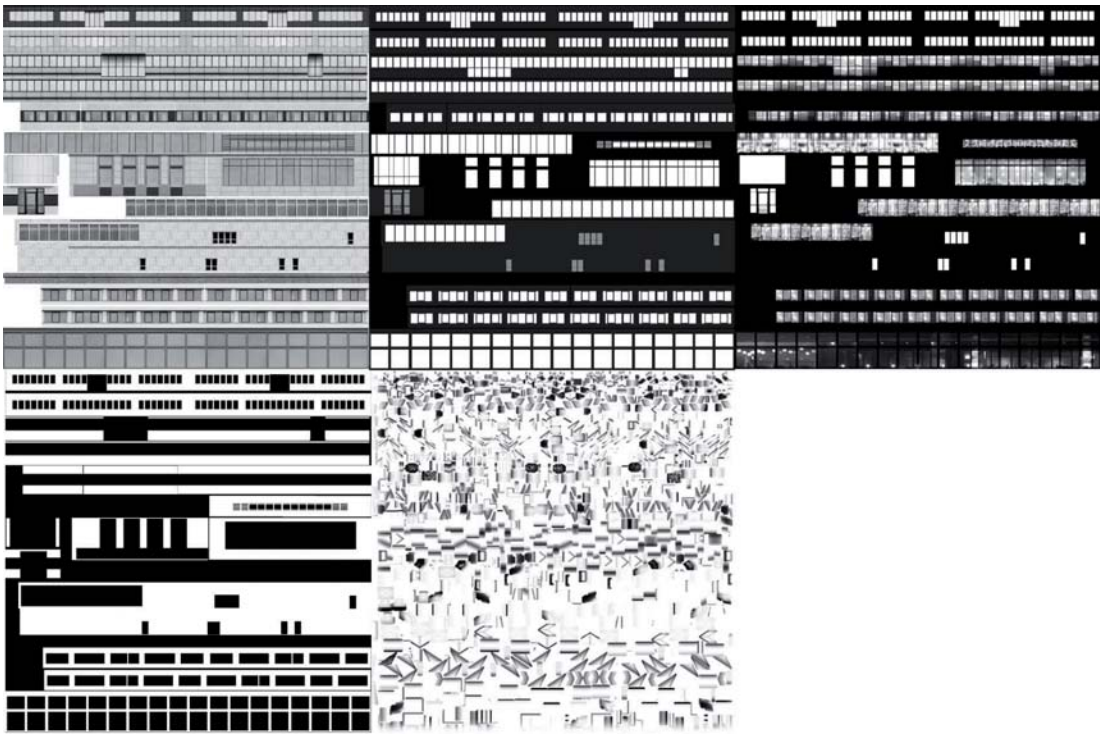

**Abbildung 5.5**: Beispiel für *Texture Maps* einer Hauskategorie; v.l.n.r. Texturen für *Diffuse Color*, *Specular*, *Illumination*, *Vertex Colors*, *Ambient Occlusion* in *UV-Set 2*

Letztendlich besteht ein Speicherbedarf von 5 MB pro Hauskategorie. D.h. es ist ein Gesamttexturspeicher für alle Stadtmodelle von maximal 40 MB erforderlich. Somit kann die Speichernutzung von 1,3 GB um 97% reduziert werden.

## 5.4 Einbindung in Unity3D

Sind alle Modelle und Texturen fertig gestellt können die Assets in Unity eingepflegt und vorkonfiguriert werden. Durch die Verwendung des standardisierten *FBX*-Dateiformats kann der Datenimport in Unity plattform- und softwareunabhängig gewährleistet werden. Der Workflow zum Import von 3D-Daten in *Unity* ist nicht für eine große Anzahl von Modellen vorgesehen und über den konventionellen Weg sehr mühsam. Dazu kommt ein großer Aufwand für eventuelle Massenänderung an den verwendeten Assets. Aus diesem Grund wurde im Rahmen dieser Arbeit ein *Unity Editor* Skript erstellt, welches den Aufwand für Import und Reimport wesentlich verringert.
Das Skript ermöglicht es aus den im *FBX*-Dateiformat vorliegenden einzelnen 3D-Modellen automatisch *Prefabs* zu generieren, diese kategorisch innerhalb des Projekts abzulegen, das richtige Material zuzuweisen sowie ein erforderliches Skript zur späteren Stadterstellung zuzuordnen. Für eine einfache Kategorisierung müssen die einzulesenden Dateien lediglich in Ordnern vorsortiert vorliegen.

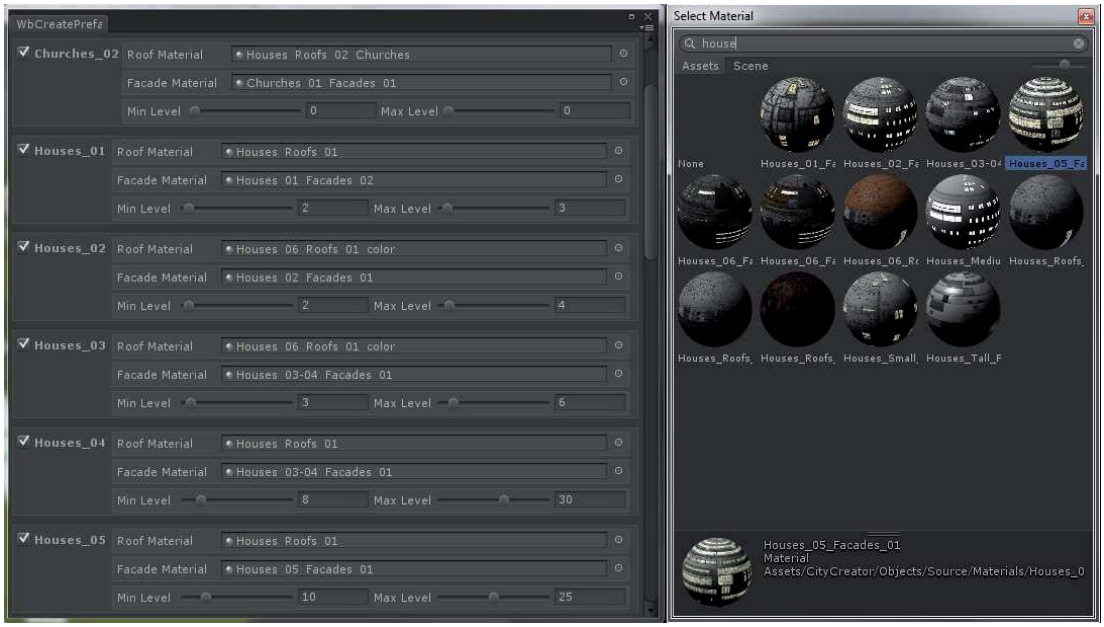


**Abbildung 5.6**: *Unity Editor* Skript für automatischen FBX-Import

Über diesen Weg kann benutzerfreundlich beliebig viel weiterer 3D-Content, welcher für die Stadtgenerierung berücksichtigt werden soll, in ein *Unity*-Projekt integriert werden. Gleichzeitig wird über das Skript sichergestellt, dass alle benötgiten 3D-Daten korrekt vorkonfiguriert für die weitere Verarbeitung vorliegen.

## 5.5 Implementation

In diesem Teil des Kapitels werden die vorangegangen Überlegungen des Konzepts aufgegriffen. Aufbauend auf den in den Optimierungsmethoden und den vorbereiteten Maßnahmen entstandenen Ergebnisse kann die softwareseitige Umsetzung vorgenommen werden.
Dazu wird die prototypische Lösung des modular aufgebauten Softwarekonzepts im Folgenden aufgezeigt und dessen Implementierung erläutert. Parallel zur Entwicklung werden permanent Performancetests durchgeführt um jederzeit auf die Leistungsoptimierung eingehen zu können. Im darauf folgenden Kapitel werden Ergebnisse des Konzepts durch abschließende Tests analysiert und bewertet.

### 5.5.1 Terrain-Datenbasis

Für die Nachbildung der 3D-Szenerie anhand realer Vorgaben sind grundlegende Geodaten notwendig. Für die Erstellung des 3D-Terrains wurden Geodaten der US-amerikanischen Luft- und Raumfahrtbehörde NASA verwendet. Mit dem *NASA World Wind*-Projekt[1] stehen sämtliche globalen Satellitendaten unter *NOSA*-Lizenz[2] (NASA Open Source Agreement) zur Verfügung. Unter anderem liegen damit genaue Geländedaten mit Höhenangaben vor, anhand derer das 3D-Terrain für den *Schiffsimlator* erstellt wurde. Es umfasst den Flussverlauf des Rheins sowie dessen Umland von Mainz bis zur holländischen Grenze und den Flussverlauf des Mains von Frankfurt-Höchst bis zur Rheinmündung. Dabei wurde ein Terrain von insgesamt ca. 2.750 $km^2$ Größe nachgebildet. Das 3D-Terrain wurde zur weiteren Nutzung in 110 seperate 5x5 km größe *Terrain Tiles* unterteilt. Die hochaufgelösten NASA-Terraindaten müssen auf echtzeitkonforme Größen runtergebrochen werden. Dazu wurde eine adäquate kleinste Größeneinheit für die Terrainunterteilung bei 10 Metern festgelegt. Diese Unterteilung bietet das ideale Verhältnis von *Mesh*-Auflösung zu ansprechender Grafik. Daraus ergibt sich pro *Terrain Tile* ein *Mesh* von 260.000 *Vertices* bzw. 520.000 *Triangles*.
Weiterhin sind Geoinformationen in Form von Kartendaten notwendig um Städe und Ortschaften inklusive Straßen und Schienennetzen nachzubilden. Hierfür kamen Kartendaten des *OpenStreetMap*-Projekts (OSM-Projekt) zum Einsatz. OSM-Daten stehen unter *Creative Common*-Lizenz und sind frei verwendbar[3]. Daten dieser Karten können zur Bearbeitung innerhalb des *Unity*-Projektes mittels Skript auf das 3D-Terrain projiziert werden (vgl. Abb. 5.8).
Die Verwendung tatsächlicher Kartendaten und real nachempfundenen Terrain ermöglicht eine realtitätsnahe Erstellung von Stadt- und Geländemodellen. Zusätzlich werden Satelliten-, Luft- und weitere Bilder als Referenzen zur Nachbildung der ortlichen Gegebenheiten herangezogen.

[1] http://worldwind.arc.nasa.gov/
[2] http://goworldwind.org/about/, Lizenz: http://worldwind.arc.nasa.gov/worldwind-nosa-1.3.html
[3] http://openstreetmap.de/, Lizenz: http://www.openstreetmap.org/copyright

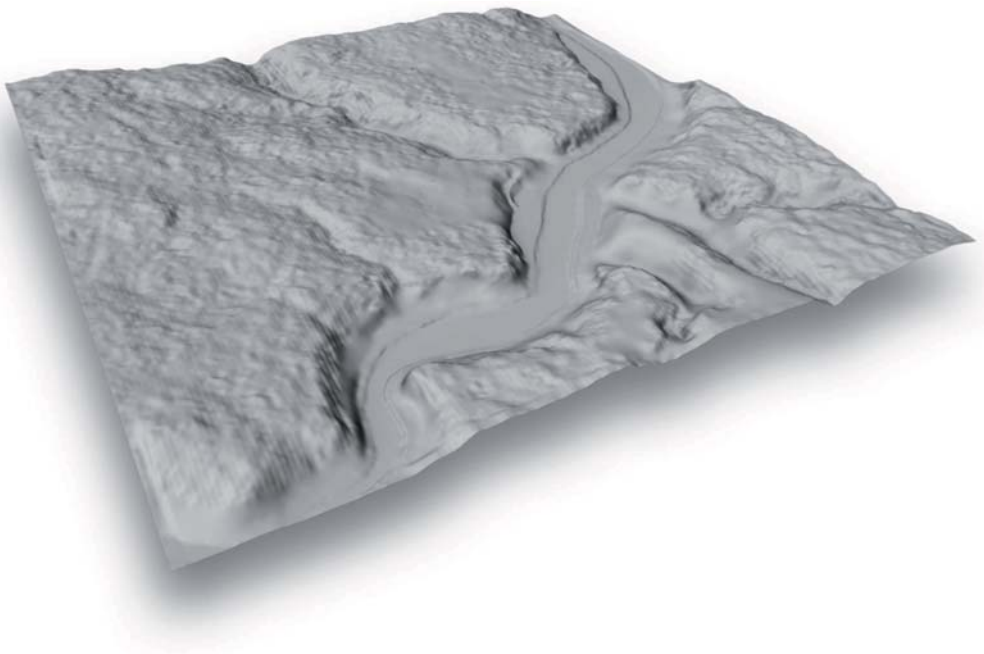

**Abbildung 5.7**: *Terrain Tile* um die *Loreley* auf Basis von NASA-Daten

### 5.5.2 Ablauf des Level-Designs eines urbanen Gebiets

Mit den Ergebnissen aus den vorangegangenen Schritten sind alle Vorbereitungen abgeschlossen und es liegen sämtliche notwendigen *Assets*, die zur Implementierung benötigt werden, vor. Das Terrain stellt die Grundlage für alle weiteren Generierungsschritte dar. Es definiert die Bereiche in denen die verschiedenen *Assets* platziert werden können [CRMW10]. Im Folgenden wird zunächst der grundsätzliche automatisierte Ablauf der Stadterstellung beschrieben. Die softwareseitige Umsetzung wird im Anschluss an diesen Teil des Kapitels erläutert.

Die prozedurale Content-Generierung bietet ein hohes Potential für das Level-Design in Bezug auf die schnelle und zeitsparende Erstellung ansprechender Modelle. Gegenüber zur manuellen Erzeugung fehlt oft eine ausreichende Kontrolle über den Erstellungsprozess. Unnatürlich gleichmäßige oder unwahrscheinliche Verteilung der Modelle kann die Folge sein [CRMW10]. Der Prozess soll trotzdem einfach und in einer angemessenen Zeit konfigurierbar sein. Mit der Vorgabe globaler Einstellungen, die oft wiederkehrend oder ähnlich sind, kann ein Großteil des Setupaufwandes für den Benutzer bereits abgefangen werden. Das globale Setup muss jederzeit anpassbar bleiben. Die manuelle Konfiguration prozessspezifischer Einstellungen ist nicht zu vermeiden. Um einen schnellen Arbeitsablauf zu erreichen sind die spezifischen Einstellungen auf die Wesentlichsten zu reduzieren. Der Benutzer behält dabei ein Mindestmaß an Kontrolle über den Erstellungsprozess.

Für eine zielgerichtete Konzeptionierung der Algorithmik muss zunächst analysiert werden wie ein realitätsnahes urbanes Gebiet aufgebaut werden kann. Grundsätzlich ist der Verteilung von Gebäuden und Anlagen in städtischen Gebieten, neben dem Gelände, abhängig vom Verkehrsinfrastrukturnetz. Der Standort und die Ausrichtung eines Hauses ist oft an eine Straße oder an einen Weg gebunden, kaum ein Haus steht gänzlich allein. Dadurch ist eine struktierte Platzierung der Gebäude anhand eines Straßen- und Schienennetzwerks möglich [CRMW10].
Der Aufbau einer urbanen Infrastruktur ist abhängig von demographischen Gegebenheiten. Genauso ist der Aufbau der Gebiete abhängig vom Infrastrukturnetz. Beispielsweise sind Bahnhöfe, Häfen oder Industriegebiete innerhalb eines Stadtgebiets stets mit entsprechender Infrastruktur ausgestattet [Jak06]. Für ein realitätsnahes Ergebnis empfiehlt sich die selbe Vorgehensweise wie beim Level-Design. Für die strukturierte örtliche Platzierung ergibt sich daraus im ersten Schritt das notwendige Verkehrsinfrastrukturnetz zu generieren. Desweiteren können durch die vorangehende Platzierung sämtlicher markanter Bauwerke, wie Wahrzeichen oder Brücken, die Bereiche definiert werden, in denen keine Gebäude automatisiert platziert werden sollen [Teo08].

Für die Erzeugung von Verkehrsnetzen anhand realer Vorgaben kommt im *Schiffsimulator* eine flexibel eingesetzbare Lösung zum Einsatz. Über die Erstellung von Straßen- und Schienennetzen wird damit auch die Erstellung der Flussgeometrie mit sämtlichen benötigten Randobjekten anhand des Flusslaufs vorgenommen. Mit Hilfe der Netzwerke sollen später automatisch KI-Fahrzeuge (Künstliche Intelligenz), wie Schiffe, Züge und Autos auf Flüssen, Schienen bzw. Straßen fahren.

Für die Erstellung eines Verkehrsnetzes kann der Benutzer im *Unity Editor* ein *Street Network* manuell hinzufügen oder verändern. Eine Straße besteht dabei aus jeweils zwei *Nodes* zwischen denen ein *Edge* aufgespannt wird. Dabei stellen die *Nodes* den Anfang der Straße bzw. eine Kreuzung dar. Ein *Edge* entspricht einer Straße. Daraus entsteht ein Netzwerk bestehend aus Straßen und Kreuzungen. Für den *Schiffsimulator* sind die verschiedenen Verkehrswege kategorisierst in Landstraßen, Straßen innerorts, Autobahnen, spezielle Bergstraßen sowie Schienenwege und spezielle Schienenwege am Berg. Jede Straße hat verschiedene Eigenschaften, wie Typ oder Breite. Für die Häuserplatzierung ist hauptsächlich die Straßenbreite relevant um die Modelle jeweils exakt an den Rand der Straße oder des Fußweges zu setzen. In Anbetracht der Kameraperspektiven innerhalb des Spiels sind viele Straßen jenseits der ersten Häuserreihen nicht mehr sichtbar. Um für die nicht sichtbaren Straßenzüge keine neuen Straßengeometrien tessellieren zu müssen können diese Straßen von der Geometrie-Tessellierung ausgeschlossen werden, um weitere Polygone zu sparen.

Sobald das Verkehrsnetz definiert ist erfolgt die Einteilung der Stadtgebiete. Dadurch wird festgelegt welche Kategorien von Häusern strukturiert an den Straßen platziert werden können. In der Realität werden Stadtgebiete anhand von Flächennutzungsplänen bzw. Bebauungspläne eingeteilt. Gebäude die innerhalb eines solchen Gebietes entstehen

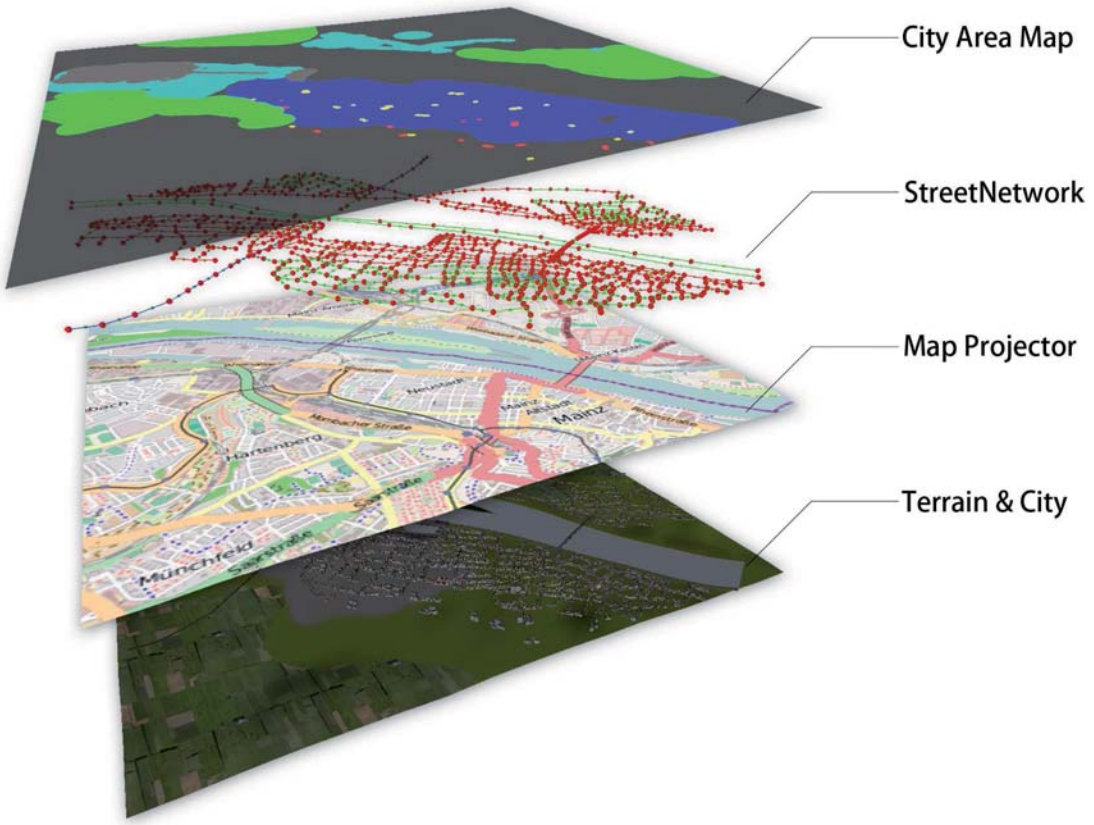


**Abbildung 5.8**: Bearbeitungsprozesse für die Stadtgenerierung am Beispiel der Stadt Mainz

müssen einer bestimmten Nutzung entsprechen [Het98]. Daran angelehnt ist die Definition und Einteilung der einzelnen Stadtgebiete mittels *City Area Map*. Dabei handelt es sich um eine farbige Maske über dem Terrain mit der die Gebiete anhand bestimmter Farben eingeteilt werden. Jedem Haustyp ist mindestens eine Farbe zugewiesen. Für die automatisierte Platzierung kommen in den farbigen Bereichen nur die Haustypen infrage, die mit der entsprechenden Farbe versehen sind. Ein weißer Farbwert wird für die zufällige Auswahl aus allen verfügbaren Haustypen verwendet. Ein schwarzer Farbwert wird für Bereiche verwendet, in denen keine Gebäude erstellt werden sollen.

Nach der Generierung der Verkehrswege, der Platzierung der Wahrzeichen und der Einteilung der Stadtgebiete erfolgt die automatisierte Erzeugung von Gebäuden und Anlagen. Damit die erzeugten Modelle strukturiert an den Straßen aufgestellt werden können wird über alle *Edges* des Straßennetzwerks iteriert und zufällige Häuser werden platziert. Dabei werden vorher konfigurierte Mindest- und Maxmimalabstände zwischen den Häusern berücksichtigt und ein Zufallswert für den Abstand gewählt. Um die Hausobjekte auf die jeweils korrekte Terrainhöhe zu setzen wird mittels *Raycast* der ent-

sprechende Wert pro Haus ermittelt. Die in den vorbereitenden Maßnahmen erstellten einzelnen Gebäudeteile können nach konfigurierten Vorgaben passend zusammengebaut werden und ergeben letztlich ein komplettes, nahezu einizgartiges Haus. Jedem Teil ist ein Skript zugeordnet, welches alle notwendigen Eigenschaften des Gebäudeteils speichert. Alle Teile die dem selben Haustyp zugewiesen sind passen zusammen. Darüber hinaus gibt die Art des Gebäudeteils Auskunft darüber ob es sich um ein komplettes Einzelteil, Fundament, Mittel- oder Dachteil handelt. Ein weiteres Attribut speichert ob es sich um ein gerades oder um ein Eckteil handelt. Bei der Generierung werden die Teile passend für gerade Straßen bzw. Straßenecken zusammen gebaut. Die als Attribut gespeicherte minimale und maximale Anzahl der Stockwerke beeinflusst die Bauhöhe des Modells zufällig. Als einen der letzten Schritte werden alle *Meshes* mit *Vertex Colors* versehen. Dabei erhalten Fassadenteile zufällig eine von 350 geeigneten Fassadenfarben. Dächer werden mit einer entsprechend passenden Dachfarbe eingefärbt. Ähnlich wie in der Realität sind die Gebäude einer Stadt oft einem oder weniger Stile zuzuordnen. Viele der Häuser ähneln sich, jedes ist für sich dennoch einzigartig durch Form und Farbe.

### 5.5.3 Softwarekonzept

Die Entwicklung der Softwarelösung zur Generierung der Städte stellt einen Teil einer Gesamtlösung dar. Sie beinhalted in dieser Arbeit nicht die prozedurale Erstellung der Verkehrsnetze. Die Erzeugung von Straßen und Schienen gehört dennoch zum Prozess der Stadtgenerierung dazu. Durch die modulare Aufbauweise der Gesamtsoftware ist der Zugriff und Austausch der Daten unter den einzelnen Modulen sicher gestellt.
Der im letzten Teil des Kapitels beschriebene Ablauf des Level-Designs gibt Aufschluss über die notwendigen Schritte und deren Abhängigkeit untereinander. Jeder der Schritte stellt einen in sich vollendeten Einzelprozess dar. Die Einzelprozesse entsprechen somit je einem Modul innerhalb der Gesamtsoftwarelösung. Dabei ist für das entstehende Endergebnis die Verarbeitungsreihenfolge der Module entscheidend. Die einzelne aufeinander folgende Abarbeitung von Teilaufgaben wird als *Pipelining* bezeichnet [AN88]. Der Gesamtprozess der Stadtgenerierung kann vereinfacht als *City Creation Pipeline* dargestellt werden (vgl. Abb. 5.9). Idealerweise sollen die Module unabhängig voneinander sowie einzeln einsetzbar sein. Der Aufbau der *Pipeline* kann an beliebiger Stelle um Module ergänzt oder reduziert werden.

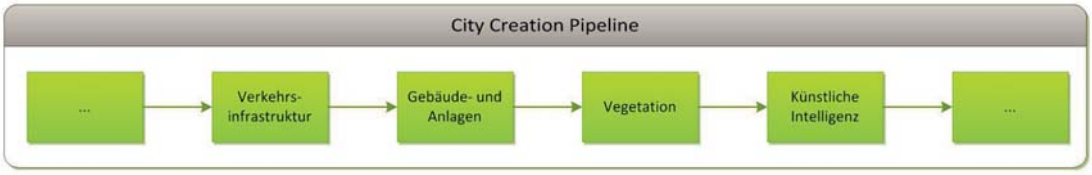


**Abbildung 5.9**: City Creation Pipeline

Das Modul der Verkehrsnetzgenerierung stellt laut Level-Desgin allerdings die zwingend erforderliche Grundlage aller weiteren Prozesse dar. Durch die Straßen- und Schienennetze erlangt die Stadt die notwendige Struktur und erhält dadurch ein realistischeres Erscheinungsbild. Um dennoch die Flexibilität, durch den unabhängigen und einzelnen Einsatz der Module, zu gewährleisten, ist es mit der Software möglich die Modelle ohne die Struktur der Verkehrswege zu erstellen und zu platzieren. Die Stadterzeugung ohne Straßennetzwerk kann beispielsweise für Orte in weit entfernten Bereichen zur Kamera Anwendung finden. Dort ist die Struktur aufgrund der hohen Distanz kaum erkennbar. Gleichzeitig kann der notwendige Arbeitsaufwand zur Erstellung solcher Stadtmodelle verringert werden, da keine Bearbeitung der Infrastruktur erforderlich ist. Die Einteilung der Stadtgebiete und Auswahl der entsprechenden Häusertypen erfolgt nach dem selben Prinzip mittels *City Area Plane*. Die Platzierung und Ausrichtung der einzelnen Modelle geschieht dabei zufällig.

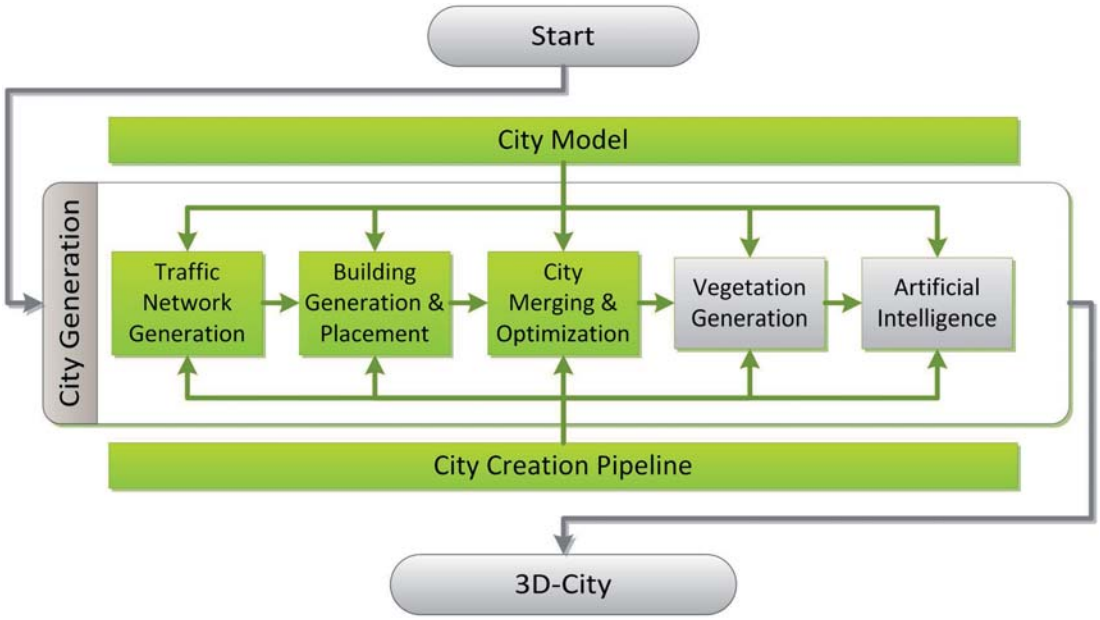


**Abbildung 5.10**: Softwareseitiger Prozess der Stadtgenerierung

Die *City Creation Pipeline* stellt die Modularität und Flexibilität der Softwarelösung sicher. Jeder aktive Schritt innerhalb der *Pipeline* wird verarbeitet und liefert ein Teilergebnis. Damit aus der Summe aller Teilergebnisse letztlich als Gesamtergebnis ein komplettes 3D-Stadtmodell geliefert werden kann ist es erforderlich, dass die Module im Quellcode untereinander kommunizieren können. Soll die Stadterstellung anhand eines Verkehrsnetzes erfolgen benötigt beispielsweise der Prozess der Gebäudeerstellung Informationen, welches Straßennetzwerk für die Platzierung zu grunde liegt. Analog ist für die Umsetzung von KI-Logik für Fahrzeuge ebenfalls das Netz von Straßen und Schienen erforderlich.

Jedes Modul entspricht im Quellcode je einer Klasse. Für den gemeinsamen Zugriff globaler Eigenschaften der Stadt erbt jede dieser Klassen von der Oberklasse *WbCityModel*. Über diese Oberklassen erhalten alle erbenden Klassen den Zugriff untereinander auf Methoden und Objekte. Die Klasse *WbCreationPipeline* stellt die grafische Benutzeroberfläche für alle Module zur Verfügung. Weiterhin werden in der Klasse alle *Pipeline Steps* initialisiert und der eigentliche Stadtgenerierungsprozess ausgeführt. Ein Modul entspricht einem *Pipeline Step*, sie erben dazu die Eigenschaften der abstrakten Klasse *WbPipelineStep*.

Nachdem mit Ausführung der Stadtgenierung tausende einzelne *Meshes* für Straßen und Häuser entstanden sind kann die notwendige Performanceoptimierung beginnen. Die *City Creation Pipeline* ermöglicht durch die einfache Erweiterbarkeit die Umsetzung der Optimierung durch ein weiteres Modul. Durch die automatische Optimierung nach dem Erstellungsprozess verringert sich der Arbeitsaufwand erheblich. Außerdem wird so ein benutzerfreundliches und einfaches Optimierungsverfahren realisiert. Bei den notwendigen Maßnahmen werden mit Hilfe des bei den Optimierungsmethoden beschriebenen *CombineChildren*-Skriptes (siehe Kapitel 4.3.2) alle *Meshes* mit dem selben Material zusammengefasst. Ein *Mesh* wird bis maximal 60.000 *Vertices* zusammengefasst. Gleichzeitig werden die Schattenoptionen für die *Combined Meshes* deaktiviert. Die zusammengefassten *Meshes* werfen somit keinen Schatten. Da die Städte nicht unmittelbar in der Nähe der Kamera stehen sollen und Echtzeit-Schatten sehr zu Lasten der Performance gehen wird auf das Werfen von Schatten für die Stadtmodelle verzichtet. Weiterhin werden im Optimierungsschritt alle deaktivierten einzelnen Straßen und Häuserobjekte aus der *Unity Hierarchy* gelöscht. Dadurch verringert sich die Ladezeit der einzelnen Szenen, sowohl im *Play* als auch im *Editor Mode*. Darüber hinaus wird die Bearbeitungszeit im Editor reduziert und es verhindert den unnötigen Verbrauch von Festplattenspeicher.
Auf weitere Module wird im Rahmen dieser Arbeit nicht eingegangen. Innerhalb der *Creation Pipeline* können weitereführende Schritte zum Beispiel die prozedurale Erstellung von Vegetation, die Anpassung des Terrain oder die Erstellung von Logiken für künstliche Intelligenz von Fahrzeugen sein.

**Abbildung 5.11**: Prozedural erstelltes Stadtgebiet im Großraum Düsseldorf im *Unity Scene View*, verteilt auf sechs *Terrain Tiles*

### Shader

Beim Erstellungsprozess der einzelnen Häuser werden den *Meshes* der Modelle *Vertex Colors* zur Darstellung von Fassaden- und Dachfarben zugewiesen. Alle verwendeten Texturen, einschließlich die *Diffuse Color Map* sind in Graustufen abgespeichert (siehe Kapitel 4.3.4). Damit die *Vertex Colors* und sämtliche Texturen die richtige Anwendung finden ist ein spezieller *Shader* notwendig. Umgesetzt werden muss die Einfärbung der *Diffuse Color Map*, die Verwendung der *Specular*, *Ambient Occlusion* sowie der *Illumination Map*.

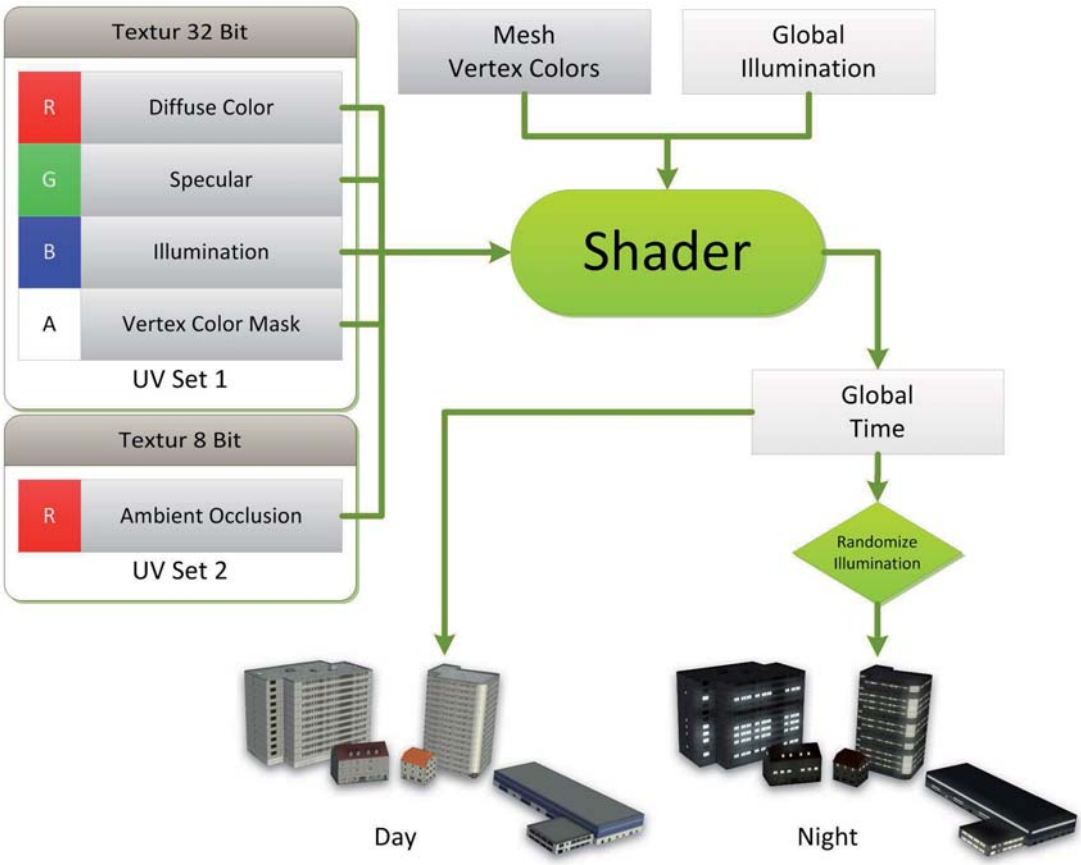


**Abbildung 5.12**: Verarbeitung durch den *Shader*

Im *Shader* wird zunächst definiert welche *Texture Map* aus welchem Farbkanal der Textur gelesen und welchem *UV-Set* des Modells sie zugeordnet werden soll. Um die *Diffuse Color Map* aus dem ersten Farbkanal (rot) einzufärben werden die Farbwerte der *Vertex Colors* auf die Werte der *Vertex Color Mask* aus dem vierten Farbkanal (alpha) multipliziert (siehe Kapitel 5.3). Anhand der *Vertex Color Map* werden gezielt nur bestimmte Texturbereiche der *Diffuse Color Map* eingefärbt. Die *Specular Map* wird aus dem zwei-

ten Farbkanal und die *Ambient Occlusion Map* aus einer seperaten Bilddatei gelesen und auf üblichem Wege vom *Shader* ausgewertet. Die *Ambient Occlusion Map* wird auf das zweite *UV-Set* der Modelle angewendet. In diesem *UV-Set* sind alle Polygone *unique* angeordnet, d.h. durch die Anordnung kann eine Überlappung der Polygone verhindert werden. Die *Illumination Map* soll in Abhängigkeit zum Tag-/Nachtsystem nur bei Dunkelheit ausgewertet werden. Flächen wie Fenster, Türen oder Lampen sollen in Verbindung mit einem *Glow*-Effekt den Nachteindruck der Modelle wesentlich prägen. In den *Vertex Colors* können Werte in vier Farbkanälen zu je 8 Bit gespeichert werden. Zur Einfärbung sind lediglich drei der Kanäle nötig um Farben einer 24 Bit Farbtiefe darzustellen. In den vierten Farbkanal (alpha) wurde ein Zufallswert gespeichert aus dem eine Zeitangabe umgerechnet werden kann. So wird verhindert, dass die Lichteffekte aller Modelle gleichzeitig aktiviert werden. Vielmehr werden die Lichter zufällig nacheinander an- und ausgeschaltet. Dadurch wirkt die Stadt lebendig. In Abhängigkeit vom globalen Zeitsystem und der Zufallszeit übernimmt der *Shader* die glühenden Lichteffekte für die in der *Illumination Map* markierten Flächen. Das globale Zeitsystem steuert den Zeitrahmen in dem Lichteffekte über *Illumination Maps* aktiviert werden können.
Das im Spiel eingesetzte *Skylight* verändert je nach Tageszeit die Farbtemperatur. Morgens und abends färbt es sich rötlich, mittags weißlich und nachts wird die Farbe stark entsättigt. Der *Shader* steuert zusätzlich die Veränderungen der Farben, die von Zeit und globalem Lichtsystem abhängig sind.

### 5.5.4 Zusammenfassung

In diesem Kapitel wurde die Konzeptionierung und prototypische Realisierung einer Softwarelösung zur prozeduralen Stadterstellung vorgenommen. Besonderes Augenmerk fiel dabei auf die Methoden der Optimierung, um zu gewährleisten, dass die Anwendung denen zu anfang des letzten Kapitels definierten Anforderungen nachkommt. Bis auf das Verfahren des *Occlusion Culling* konnten alle Optimierungsmethoden umgesetzt werden. Aufgrund projektspezifischer Besonderheiten konnte dieses Verfahren nicht mit der *Unity Engine* umgesetzt werden.
Die entwickelte Anwendung ermöglicht einem Benutzer die zeitsparende Erstellung urbaner Gebiete auf einfache Weise innerhalb *Unitys*. Für den Erstellungsprozess können weitere individuelle Modelle einbezogen werden. Durch den modularen Aufbau kann die Software beliebig und flexibel um Funktionalitäten erweitert werden. Die Entwicklung der Software erfolgte speziell und zielgerichtet für den *Binnenschiffsimulator*. Die Überlegungen und Vorgehensweisen zur Umsetzung können dennoch problemlos auf andere Projekte übetragen werden.
Im nächsten Kapitel werden die Ergebnisse der Realisierung aufgezeigt und untersucht. Anhand von Tests soll die geforderte Echtzeitperformance analysiert und ggf. Anpassungen vorgenommen werden.

# Kapitel 6

# Ergebnisse

In den vorangegangenen Kapiteln wurde eine Softwarelösung zur prozeduralen Erstellung von Echtzeit-optimierten 3D-Stadtmodellen konzipiert, entwickelt und implementiert. In diesem Kapitel wird sichergestellt, dass bei Einsatz der Stadtmodelle innerhalb einer Echtzeitanwendung die erforderliche Rechenleistung zur Verfügung steht. Dazu wird zunächst die Echtzeitperformance analysiert und ggf. werden Optimierungen vorgenommen und beschrieben. Anschließend werden die Ergebnisse des Konzepts präsentiert.

## 6.1 Performanceanalyse

Bei allen im Konzept verwendeten *Assets*, die zum Prozess der Stadterstellung herangezogen werden, wurden die Modelle und Texturen bereits bei der Erstellung möglichst effizient gestaltet. Zusätzliche Optimierungsmaßnahmen wurden zur Verbesserung der Grafikleistung vorgenommen. Um gewährleisten zu können, dass zur Programmlaufzeit tatsächlich die erforderliche Leistung für eine flüssige Darstellung im Spiel zur Verfügung steht ist es notwendig das Leistungsverhalten zu analysieren und auszuwerten. Kann hierbei die Leistung nicht erreicht oder aufrecht erhalten werden sind weitere Optimierungsmaßnahmen durchzuführen.

### 6.1.1 Auswahl der Software

Speziell für 3D- und Spieleentwickler stehen verschiedene Anwendungen zur detaillierten Performanceanalyse zur Verfügung. *NVIDIA* bietet mit *PerfHUD*[1] ein umfangreiches Echtzeitanalyse-Tool für *Direct3D*-Anwendungen an. Das Tool bietet allerdings nur Unterstützung für proprietäre Grafikhardware des Herstellers. Analog dazu bietet auch *AMD* (ehem. *ATI*) mit *GPU PerfStudio*[2] ein Echtzeitanalyste-Tool für herstellereigene Hardware an. Zusätzlich unterstüzt dieses Programm neben *Direct3D* auch

[1] http://developer.nvidia.com/nvidia-perfhud
[2] http://developer.amd.com/archive/gpu/perfstudio/Pages/default.aspx

*OpenGL*-Anwendungen. Für eine herstellerunabhängige Analyse steht das freie Programm *gDEBugger*[3] zur Verfügung. Die Software bietet einen hohen Funktionsumfang beschränkt sich allerdings auf *OpenGL*-Anwendungen. Mit dem *Graphics Performance Analyzer*[4] der Firma *Intel* (Intel GPA) kann unter anderem die Performanceanalyse von *Direct3D*-Anwendungen unabhängig von Grafikhardware vorgenommen werden. Dieses Programm beschränkt sich bei bestimmten Analyse-Modulen auf die Verwendung *Intel Core-basierter* Prozessoren.
Um die Leistungsverhalten mit verschiedenen Grafikkarten zu analyisieren und die Funktionalität sicherzustellen kommt keine herstellerspezifische Performance-Software nicht in Frage. Darüber hinaus nutzt der *Schiffsimulator* als *Grafik-API Direct3D*. Folglich wird zur Performanceanalyse *Intel GPA* verwendet.

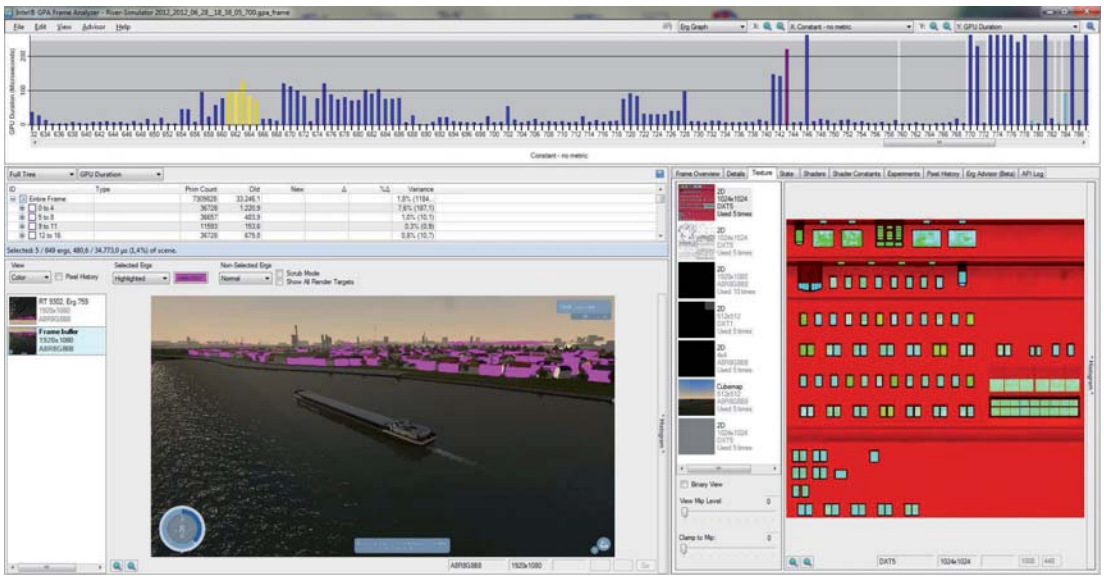

**Abbildung 6.1**: Analyse einzelner *Frames* mit *Intel GPA Frame Analyzer*

### 6.1.2 Durchführung der Performanceanalyse

Der *Intel GPA* bietet verschiedene Module zur Analyse von hardware-seitig verarbeiteten Daten. Für die Analyse der Grafikleistung wird der *Intel GPA Frame Analyzer* verwendet. Sämtliche durch die *Rendering Pipeline* (siehe Kapitel 3.2) verarbeitete Grafikdaten der Anwendung können detailliert pro *Frame* untersucht werden. Dazu muss jedes zu untersuchende *Frame* einzeln mit Hilfe der *Capturing*-Funktion zur Laufzeit erfasst werden. Für jedes erfasste Einzelbild sind Informationen der gesamten Szene und

[3] http://www.gremedy.com/
[4] http://software.intel.com/en-us/articles/vcsource-tools-intel-gpa/

allen in der Szene befindlichen Einzelobjekten verfügbar. Zu den wichtigsten Informationen der Gesamtszene zählen die Berechnungsdauer eines *Frames* (in $\mu s$), die Anzahl aller *Draw Calls* und der benötigte Gesamttexturspeicher. Die Einzelobjekte entsprechen je einem *Draw Call*, zu dem jeweils weitere ausführliche Informationen gespeichert sind.

Damit eine zielgerichtete Performanceanalyse stattfinden kann müssen die Mindestanforderungen an die Hardware definiert werden. Für Systeme, die den Mindestanforderungen entsprechen, ist es nicht zwingend erforderlich die Lauffähigkeit in bester Grafikqualität sicherzustellen. *Unity* bietet dazu die Möglichkeit verschiedene Qualitätseinstellungen[5] zu konfigurieren. Die Qualität lässt sich im späteren Programm umstellen. Für die Reduzierung der Grafikqualität wird überlicherweise auf eine hohe Bildschirmauflösung, Schatten, Reflektionen, diverse Lichteffekte, *Anti-Aliasing* oder größere Kamerasichtweiten verzichtet. Neben den Mindestanforderungen empfiehlt sich die Definition der Hardwarespezifikationen eines konkreten Zielsystems. Die Lauffähigkeit der Anwendung in bester verfügbarer Grafikqualität ist für das Zielsystem zwingend zu gewährleisten.

Für den *Schiffsimulator* wurden folgende wesentliche Anforderungen definiert:

**Tabelle 6.1**: Definition der Systemanforderungen.

| | **Minimum** | **Empfohlen** |
|---|---|---|
| **Prozessor** | 2,0 GHz | 2 x 2,0 GHz<br>oder besser |
| **Arbeitsspeicher** | 2.048 MB | 3.072 MB |
| **Grafikkarte** | 512 MB<br>DirectX 9.0 fähig | 1.024 MB<br>DirectX 9.0 fähig |

[5] http://docs.unity3d.com/Documentation/Components/class-QualitySettings.html

Zur Performanceanalyse der Grafikleistung werden zu den Systemanforderungen verschiedene Rechnersysteme mit entsprechend ausgestatter Grafikkarte unterschiedlicher Hersteller herangezogen.

**Tabelle 6.2**: Testsysteme.

| **System**<br>Komponente | **Hardware** |
|---|---|
| **Rechner 1** | |
| Prozessor | AMD Phenom II X4 965<br>4 x 3,4 GHz |
| Arbeitsspeicher | 8.192 MB |
| Grafikkarte | ATI Radeon HD 4600 Series<br>512 MB |
| **Rechner 2** | |
| Prozessor | AMD Phenom II X4 940<br>4 x 3,0 GHz |
| Arbeitsspeicher | 4.096 MB |
| Grafikkarte | ATI Radeon HD 4800 Series<br>1.024 MB |
| **Rechner 3** | |
| Prozessor | Intel Core i5-2500<br>4 x 3,3 GHz |
| Arbeitsspeicher | 8.192 MB |
| Grafikkarte | NVIDIA GeForce GTX 460<br>1.024 MB |
| **Rechner 4** | |
| Prozessor | Intel Core2 Quad<br>4 x 2,7 GHz |
| Arbeitsspeicher | 8.192 MB |
| Grafikkarte | NVIDIA GeForce GTX 480<br>1.536 MB |

### 6.1.3 Ergebnisse der Performanceanalyse

Damit die Ergebnisse der verschiedenen System für die Auswertung vergleichbar sind erfolgte Performanceanalyse im *Schiffsimulator* unter selben Parametern. Als Umgebung wurde der Terrainbereich mit dem größten zusammenhängenden Stadtgebiet ausgwählt. Im Spiel werden zusätzlich zur aktuell geladenen *Terrain Tile* auch alle benachbarten *Terrain Tiles* geladen. Es können also bis zu neun *Tiles* gleichzeitig geladen sein. Das Terrain des Großraum Düsseldorf erfordert die höchsten Ansprüche an die Grafikverarbeitung. Hier sind neben dem größten zusammenhängenden Stadtgebiet zusätzlich zwei detailliert nachbebildete Häfen zu finden. Außerdem stellen die zu ladenden benachbarten *Terrain Tiles* ausschließlich Stadtgebiet dar.

Parallel zur Umsetzung des Konzeptes wurden Performancetests durchgeführt um stets bei Problemen beim Leistungsverhalten eingreifen zu können. So traten anfangs Probleme bei der Berechnung der komplexen Geometrien der Stadtmodelle im Zusammenspiel mit der Berechnung der restlichen 3D-Szenerie auf. Die Menge der Geometriedaten großer Stadtmodelle konnte durch die *Rendering Pipeline* (siehe Kapitel 3.2) der Grafikkarte nicht in erforderlicher Zeit verarbeitet werden, ein Engpass der Geometrieberechnung ist entstanden. Pro *Terrain Tile* enthielt ein Modell ca. 9 Mio. *Vertices* bei relativ moderater Anzahl von *Draw Calls*. Bei Laden aller benachbarten *Terrian Tiles* mussten bis zu 80 Mio. *Vertices* verarbeitet werden. Viele dieser Gebäudemodelle einer Stadt waren aufgrund des begrenzten Kamerasicht- und bewegungsbereichs für den Spieler zur Laufzeit des *Schiffsimulator* nicht sichtbar. Außerdem waren hintere Bereiche der Stadt oft durch Häuser im Vordergrund verdeckt. Um dem Engpass entgegen zu treten wurden die Städte anhand das Sichtfelds der Kamera um die irrelevanten Modelle drastisch reduziert. Weiterhin konnte die Dichte der Häuser aufgrund des spitzen Sichtwinkels verrinert werden. Dadurch konnte die Menge der *Vertices* um ein zehnfaches vermindert werden. Gleichzeitig konnte das Stadtbild aus Sicht des Spielers überwiegend aufrecht erhalten werden. Die Anzahl der *Vertices* konnte auf ca. 1 Mio. pro *Terrain Tile* bzw. maximal 9 Mio. *Vertices* inklusive aller benachbarten *Tiles* minimiert werden. Bei erneuten Performancetests konnten keine nennenswerten Probleme durch Engässe festgestellt werden.

Damit die Darstellung bewegter Bilder einem Betrachter flüssig erscheint muss eine Bildwiederholfrequenz von mindestens 25 Hz gewährleistet werden. D.h. die Berechnung eines *Frames* darf die Dauer von 40 ms nicht überschreiten. Die abschließenden Performancetests wurden zunächst auf jedem System bei bester Grafikqualität durchgeführt. Konnte dabei die erforderliche *Frame Rate* nicht erreicht werden, wurde die Qualität mit Hilfe der vorkonfigurierten Qualitätseinstellungen verringert und erneut ein Test durchgeführt. Mit drei der vier angegebenen Testsysteme kann eine flüssige Darstellung bei bester Grafikqualität gewährleistet werden. Mit dem System, welches die Mindestanforderungen erfüllt, konnte keine Berechnung von 25 *Frames* pro Sekunde (FPS) bei bester und mittlerer Grafikqualität erreicht werden. Erst bei minimalen Qualitätseinstellungen kann auch hier die flüssige Darstellung gewährleistet werden (vgl. Tabelle 6.3). Bei ma-

ximalen Qualitätseinstellungen sind durch die Grafikkarte zwischen 800 und 1.200 *Draw Calls* zu verarbeiten, bei minimalen Einstellungen mindestens 470. Die Zahl der *Draw Calls* ist abhängig von Lichtern, Effekten, anderen beweglichen Objekten in der Szene durch zufällige *KI-Logik* sowie von den Objekten innerhalb des aktuellen Kamerafrustums (siehe Kapitel 4.3.1). Davon entfallen unabhängig von Grafikqualität zwischen 50 und 80 *Draw Calls* auf die Stadt. Die Berechnundsdauer der Stadtmodelle schlägt im Durchschnitt mit 8 bis 11% der Gesamtberechnungsdauer eines *Frames* zu buche. Dabei treten keine nennenswerten Unterschiede zwischen den verschiedenen Testsystemen auf. Der erhöhte Anteil der Stadtberechnung bei mittlerer und niedriger Grafikqualität ist auf die geringere Anzahl anderer *Draw Calls* zurückzuführen, wodurch sich der Anteil der Stadt bei gleichbleibend vielen Stadtojekten erhöht.
Die Inanspruchname des Texturspeichers der Grafikkarte für die Stadtmodelle lag nie höher als bei der erwarteten Größe von ca. 40 MB (siehe Kapitel 5.3). Bei bester Texturqualität liegt der Gesamtspeicherbedarf für Texturen im Spiel mit 250 bis 400 MB absolut im Rahmen der Mindestanforderungen. Hier besteht kein Handlungsbedarf.

**Tabelle 6.3**: Performanceanalyse der Gesamtszenerie

| System | Berechnung durch GPU | Anteil Stadt | FPS | Grafik-qualität |
|---|---|---|---|---|
| **Rechner 1** | 72,4 ms - 126 ms | 10,3% | 8 - 14 fps | Maximal |
| **Rechner 1** | 46,24 ms - 63,5 ms | 16,7% | 16 - 22 fps | Mittel |
| **Rechner 1** | 32,2 ms - 35,36 ms | 21,2% | 29 - 32 fps | Minimal |
| | | | | |
| **Rechner 2** | 35,48 ms - 39,46 ms | 9,8% | 26 - 29 fps | Maximal |
| | | | | |
| **Rechner 3** | 31,67 ms - 36,71 ms | 8,1% | 29 - 32 fps | Maximal |
| | | | | |
| **Rechner 4** | 27,31 ms - 30,41 ms | 8,5% | 34 - 38 fps | Maximal |

## 6.2 Ergebnisse

In der Konzeptionierung wurden die Anforderungen an die Softwarelösung definiert, die es mit der Realisierung des Konzepts umzusetzen galt. Folgende Anforderungen wurden definiert.

- Optimierung des Arbeitsaufwandes für das Level-Design
- Prozedurale Generierung von 3D-Inhalt innerhalb *Unitys*
- Echtzeitgrafikfähigkeit der generierten Modelle
- Steuerbarkeit der räumlichen Verteilung der Einzelmodelle
- Projektunabhängige Verwendbarkeit
- Erweiterbarkeit
- Benutzerfreundlichkeit

Mit der im Konzept entwickelten prototypischen Softwarelösung werden alle definierten Anforderungen erfüllt. Mit der Software wird innerhalb der *Unity Game Engine* ein effizientes Level-Desgin urbaner Gebiete durch prozedurale Modellerzeugung ermöglicht. Die Software erlaubt darüber hinaus auch die Erstellung anderer Gebiete durch die beliebige Einbindung von weiterem individuellem 3D-Inhalt. Für den vereinfachten Import von neuen Modellen oder Modellteilen wird zusätzlich ein Tool bereit gestellt, welches die Grundkonfiguration und strukturierte Speicherung der *Assets* automatisch vornimmt. Gleichzeitig wird damit eine Möglichkeit zum Massenimport bereitgestellt. Das gesamte Tool kann projektunabhängig in *Unity* eingebunden werden und dort Anwendung finden. Der mit der *Creation Pipeline* geschaffene modulare Aufbau gewährleistet eine einfache Erweiterbarkeit der Software um weitere Module und Funktionen. Im Rahmen des Konzepts dieser Arbeit wurde ein Modul für die Erstellung von Verkehrinfrastrukturnetzen eingebunden. Dadurch kann eine grundlegene Struktur in ein urbanes Gebiet übernommen werden. Die Platzierung von *Assets* kann innerhalb eines Straßen- oder Schienennetzes statt finden. Zusätzlich stellt das Netzwerk die Basis für weitere Funktionen dar, wie zum Beispiel die Entwicklung eines *KI-Systems* zur Automatisierung von Straßenverkehr. Mit einem weiteren Modul wurden abschließende Optimierungsmaßnahmen automatisch eingebunden.
Mittels *City Area Map* kann auf einfache Weise die Verteilung der prozedural erstellten Modelle erfolgen. Beim Erstellungs- und Platzierungsvorgang werden nur die *Assets* gewählt, die für das über die farbige Maske gewählte Gebiet in Frage kommen. Dazu ist jedem *Asset* mindestens eine Farbe zugewiesen. Über den Alphawert der *City Area Map* wird zusätzlich die Dichte der Platzierung reguliert. Für die gänzlich zufällige Auswahl der zur Verfügung stehenden Modelle kann das Gebiet weiß markiert werden. Soll keine Erstellung stattfinden wird der Bereich auf der *City Area Map* schwarz markiert. Mit

dieser simplen Lösung kann das Level-Design auf schnellsten Weg bei verhältnismäßig großen Gebieten vorgenommen werden. Gleichzeitig ist ein Mindestmaß an Kontrolle über den Erstellungsprozess sichergestellt. Oft werden ähnliche Parameter für die automatische Erstellung benötigt. Deshalb können die wichtigsten Einstellungen global konfiguriert werden, auf die beim automatischen Prozess zurück gegriffen wird. Spezifische zusätzliche Parameter ermöglichen eine individuelle Erstellung.
Die Performanceanalyse bestätigt die Echtzeitfähigkeit der Stadtmodelle innerhalb einer komplexen Gesamtszenerie.

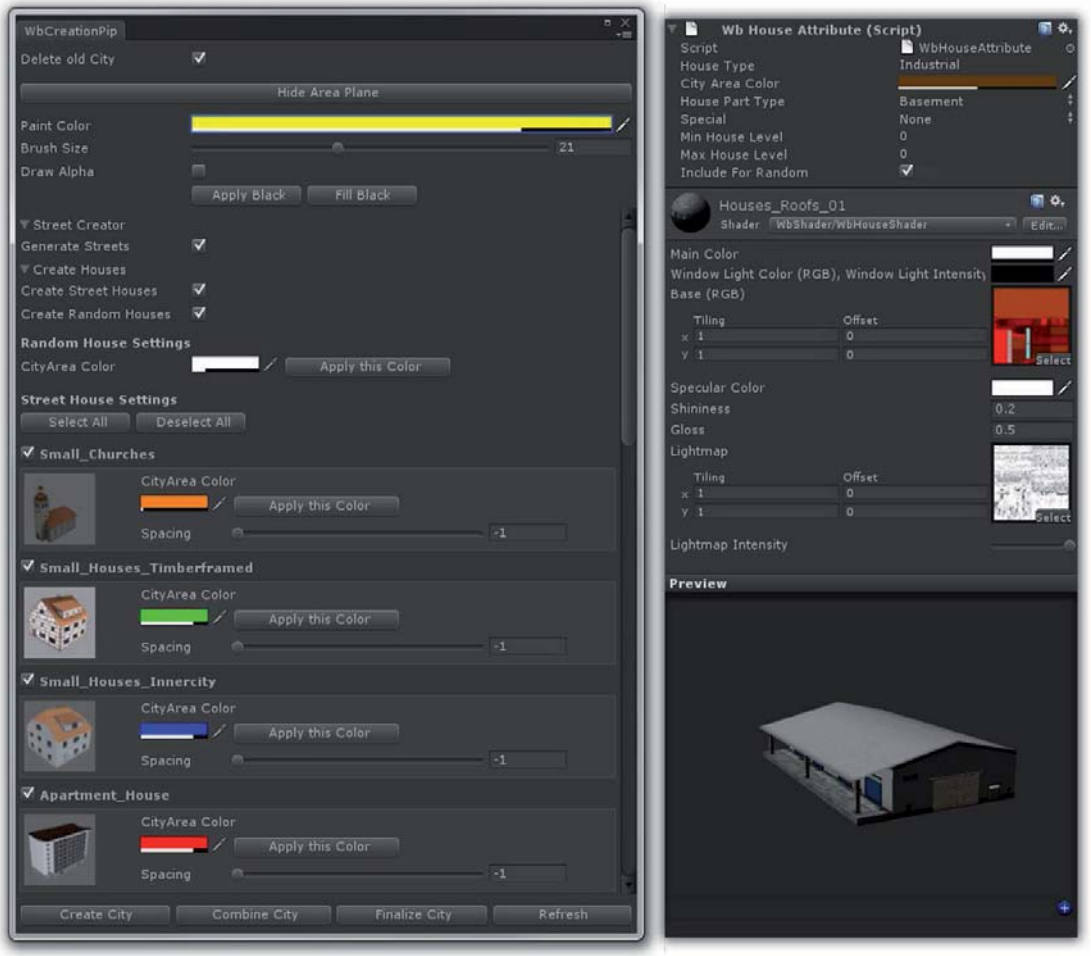


**Abbildung 6.2**: Prototypische Bedienoberfläche der Softwarelösung in *Unity3D*. Rechts: Konfiguration der globalen Parameter und des *Shaders*

.

**Abbildung 6.3**: Prozedural erstellte Städte im Schiffsimulator. Von oben nach unten: Düsseldorf mit Medienhafen, Deutsches Eck bei Koblenz, Kölner Kranhäuser im Vordergrund und Dom im Hintergrund, Mainzer Zollhafen und Stadt im Hintergrund

.

# Kapitel 7

# Zusammenfassung und Ausblick

Im Vordergrund dieser Arbeit stand die Entwicklung einer Softwarelösung mit der es möglich ist 3D-Stadtmodelle prozedural zu erzeugen. Die automatisch erzeugten Stadtmodelle sind für den Einsatz in Echtzeitgrafikanwendungen optimiert. Durch eine modulare Aufbauweise ist die Software flexibel um Funktionen erweiterbar. So sind während der Arbeit Module für die Erzeugung von Verkehrsinfrastrukturnetzen oder für die automatisierte Optimierung entstanden. Der Erzeugungsprozess kann individuell parametrisiert werden. Dabei werden grundlegende Einstellungen global und spezifische Einstellungen manuell bei der Erstellung gesetzt. Um möglichst reale Einschätzungen zur Performance innerhalb einer komplexen Echzeitanwendung treffen zu können, wird die Software bei der Umsetzung des Konzepts innerhalb des kommerziellen Computerspiels *Schiffsimulator 2012 - Binnenschifffahrt* entwickelt und eingesetzt.

Im ersten Kapitel wurde eine kurze Einführung in die Thematik geliefert, die Problematik beschrieben und die Ziele dieser Arbeit definiert. Anschließend wurden mit dem Stand der Technik Konzepte und Lösungsansätze beschrieben an denen geforscht wird um virtuelle 3D-Umgebungen durch prozedurale Generierung von Modellen nachzubilden. Im speziellen wurden Methoden zur Erzeugung urbaner Gebiete erläutert. Darüber hinaus wurden aktuelle kommerzielle Softwarelösungen beschrieben, die eine prozedurale Stadterstellung ermöglichen. Projekte die das Ziel verfolgen die Visualisierung von 3D-Inhalt für Internet und mobile Geräte zu realisieren lassen darauf schließen, dass sich in diesem Bereich zukünftig weitere Projekte und Lösungen entwickeln werden. Neue Technologien ermöglichen die schrittweise Erweiterung von Echtzeit-, Browser- und Internetanwendungen, sodass sich deren Leistungsfähigkeit immer mehr denen von Desktop-Applikationen annäheren wird. Dennoch exisitiert zur Zeit keine ganzheitliche Lösung um alle genannten Bereiche abzudecken.
Im nächsten Kapitel wurden die Grundlagen heutiger Echtzeitgrafik und die Verarbeitung von Grafikdaten durch die Hardware anhand der *Rendering Pipeline* beschrieben. Der Einfluss auf die Verarbeitung der Daten durch programmierbare Schnittenstellen und *Shader*-Programme hat weitere Möglichkeiten für Lösungsansätze aufgezeigt.
Nach der Vermittlung des Basiswissens erfolgte die Konzeptionierung zur Entwicklung

einer Softwarelösung für die automatisierte Erzeugung von Stadtmodellen unter Einbeziehung der gesetzten Ziele. Dazu wurden zunächst die Anforderung an die Software definiert und in Frage kommende Optimierungsmethoden zur Erreichung und Einhaltung der Echtzeitperformance ausführlich beschrieben. Daraufhin erfolgte die Umsetzung des zuvor entwickelten Konzeptes, beginnend mit der Auswahl einer von mehreren möglichen Verfahrensweisen zur prozeduralen Erstellung von 3D-Modellen. Daraus resultierten die weiteren Schritte, die zur Vorbereitung der Umsetzung notwendig waren. Anhand eines Konzepts zur Softwareentwicklung konnte die Implementation der Anwendung erfolgen. Das Ergebnis der Entwicklung ist die prototypische Softwarelösung zur Stadterstellung, die unter Einbeziehung der zuvor gesetzten Ziele und den definierten Anforderungen evaluiert wird. Die Echtzeitfähigkeit der mit der Software prozedural erstellten Stadtmodelle wird durch eine Performanceanalyse bewertet und schließlich bestätigt.

Nachdem die Anwendung erfolgreich prototypisch umgesetzt ist und die Echtzeitperformance der automatisch erstellten Städte gewäherleistet werden kann, wird die Software wie beschrieben zunächst im *Schiffsimulator* eingesetzt. Die visuelle Evaluation der Stadtmodelle kann in weiteren Schritten durch Kundenrezessionen oder Befragungen durchgeführt werden. Bereits während der Konzeptrealisierung und weit vor dem Release-Termin des Spiels erfolgte der Aufbau einer Community. Einige vorab veröffentlichte Screenshots wurden bereits positiv durch die zukünftigten Benutzer bewertet. Darauf aufbauend kann nach Veröffentlichung des Spiels eine gezielte Evaluation durch Benutzer erfolgen. Im *Schiffsimulator* sind Funktionen für *Modding* durch Spieler und Community vorgesehen. Um die Benutzer mehr einzubinden ist eine Funktion zum Import weiterer Modelle denkbar. So könnten Nutzer spezielle Wahrzeichen, Brücken oder markante Bauwerke selbst einbringen. Durch eine Vielzahl markanter Bauwerke mit einem großem Wiedererkennungswert steigt sowohl das Realismusempfinden im Spiel, als auch die Aktzeptanz aller Benutzer.

Bei der Performanceanalyse sind Engpässe bei der Verarbeitung komplexer Geometrien der anfänglich verwendeten Stadtmodelle deutlich geworden. Daraufhin mussten die Modelle wesentlich reduziert werden. Bei der jetzigen Optimierung erfolgt das Zusammenfassen der *Meshes* bis zu einer Anzahl von maximal 60.000 *Vertices* pro *Mesh* (siehe Kapitel 5.5.3). Dadurch entstehen verhältnismäßig wenige *Draw Calls* die gleichzeitig durch die Grafikkarte verarbeitet werden können. Allerdings kann ein *Draw Call* bei dieser Grenze sehr komplexe Geometrien erhalten. Dies kann zur Folge haben, dass die Verarbeitung eines *Draw Calls* zuviel Zeit in Anspruch nimmt und ein Engpass beim *Vertex Shader* innerhalb der *Rendering Pipeline* entsteht. Die gezielte Analyse dieser Problematik ist empfehlenswert. Unter Umständen kann ein besseres Gleichgewicht von *Draw Calls* und zu verarbeitenden Geometriedaten eine weitere Performancesteigerung bewirken.

Weiterhin sollte die zusätzliche Verwendung dynamisch angepasster *Meshes* zur Reduzierung von Geometriedaten und zur Verbesserung des Stadtbilds untersucht werden. Bei der Verfahrensauswahl (siehe Kapitel 5.1) ist die Wahl auf die Verwendung vor-

modellierter Modelle bzw. Modellteile gefallen, da hierbei offensichtlich eine maximale Performanceoptimierung erfolgen kann. Die zuvor beschriebene Problematik zu vieler Geometriedaten ist dennoch aufgetreten. Je größer und detaillierter eine virtuelle Umgebung sein soll, desto schwieriger wird es das Aufkommen komplexer Geometrien zu vermeiden. Bei der Kombination aus beiden Vorgehensweisen können unter Umständen Polygone gespart und gleichzeitig Häuser passgenau verbunden werden. Auch die Einbringung in das Gelände oder an den Straßen kann durch dynamische Anpassung der *Meshes* zum Erstellungsprozess besser geregelt werden.

Auch untersucht werden sollte die Auslagerung von Verarbeitunsprozessen auf die *GPU*. Die relativ neue programmierbare Komponente des *Geometry Shaders* (siehe Kapitel 3.3) innerhalb der *Rendering Pipeline* der Grafikkarte stellt eine interessante Möglichkeit zur Performancesteigerung dar. Mit Hilfe des *Geometry Shader* können einfache Primitive direkt durch die *GPU* parallel zur Vearbeitung erzeugt und um ein vielfaches instanziiert werden. Durch die direkte Erzeugung auf der *GPU* steht ein enormes Potential für Performancesteigerungen zur Verfügung. Für die Erstellung von Stadtmodellen können so unter Umständen einfache Würfel oder Flächen erstellt, rudimentär texturiert und tausendfach als Füllobjekte für weite Entfernungen genutzt werden. Die Programmierung des *Geometry Shader* ist ab *Direct3D 10* möglich. *Unity 3.4*, wie in dieser Arbeit verwendet, bietet Unterstützung bis *Direct3D 9*. Während der Erstellung dieser Arbeit wurde *Unity 4*[1] angekündigt. Es bietet Unterstützung bis *Direct3D 11*. Aufgrund des enormen Potentials ist die Untersuchung der Performancesteigerung durch Programmierung von *GPU*-Komponenten sehr empfehlenswert.

[1] http://unity3d.com/?unity4

# Glossar

| | |
|---|---|
| CPU | CPU (*Central Processing Unit*) bezeichnet die zentrale Verarbeitungseinheit (Hauptprozessor) in einem Computersystem. |
| GPU | GPU (*Graphical Processing Unit*) bezeichnet den Prozessor für Grafikverarbeitung innerhalb eines Computersystems. Oft unterstützt die GPU die CPU bei der Verarbeitung von Grafikdaten. |
| RGB / RGBa | RGB bezeichnet ein additives Farbmodell, bei dem die Menge der wahrnehmbaren Farben durch die drei Grundfarben Rot, Grün und Blau (RGB) definiert wird. |
| Editor Mode | auch *Scene View*, bzeichnet den Szenen-Bearbeitungsmodus von *Unity*. Im *Scene View* können alle *Game Objetcs* selektiert und bearbeitet werden. Es ist die zentrale Ansicht zur Bearbeitung von Szenen innerhalb der *Unity Engine*. |
| Play Mode | auch *Game View*, bezeichnet in *Unity* die Sicht aus der festgelegten Kamera zur Laufzeit des Programms. Das Programm bzw. Spiel wird ausgeführt. Während des Play-Mode sind sämtliche Änderungen im Programm nur temporär gültig. Bei Verlassen des Play Mode werden die Einstellungen zurückgesetzt. |
| Prefab | Ein Prefab in Unity3D ist ein wiederverwendbares GameObject, welches im ProjectView innerhalb von Unity gespeichert wird. |
| Draw Call | Damit ein Objekt auf dem Bildschirm gezeichnet wird, muss die 3D-Engine den sogenannten Draw Call (Aufruf zum Zeichnen) an das Grafik-Interface senden. |
| Bottleneck | Zu dt. Flaschenhals. Bezeichnet einen Engpass innerhalb eines dynamischen Systems bzw. dynamischen Prozessablaufs [Wag11]. |

| | |
|---|---|
| Mip Map | Eine *Mip Map* stellt eine Folge von Rasterbildern desselben Motivs dar. Die Kantenlänge der Bilder nimmt jeweils um die Hälfte ab. Durch *Mip Mapping* wird versucht die Kantenbildung (*Aliasing*) die bei Verkleinerung eines Bildes durch die Pixel entstehen zu vermeiden. Dafür werden die jeweiligen Einzelbilder jeweils neu berechnet (*Down-Sampling*). |
| Cube Map | In einer *Cupe Map* werden sechs 2D-Texturen gespeichert, deren Anordnung der Seiten eines Würfels entsprechen. *Cube Maps* werden meist zur Darstellung der Umgebungspiegelungen verwendet. |
| Z-Buffer | auch *Depth Buffer*, speichert die Tiefenkomplexität einer Szene, d.h. die Entfernungen der Objekte zur Kamera werden gespeichert. |

# Literaturverzeichnis


[AM99] Assarsson, Ulf ; Möller, Tomas: *Optimized View Frustum Culling Algorithms*. März 1999

[AMHH08] Akenine-Möller, Tomas ; Haines, Eric ; Hoffman, Naty: *Real-Time Rendering*. Third Edition. Peters, Wellesley, 2008. – ISBN 978–1–56881–424–7

[AN88] Aiken, Alexander ; Nicolau, Alexandru: Perfect Pipelining: A new loop Parallelization Technique. In: *Lecture in Computer Science* Bd. 300, 1988, S. 221–235

[BB06] Bender, Michael ; Brill, Manfred: *Computergrafik - Ein anwendungsorientiertes Lehrbuch*. 2. Auflage. Carl Hanser Verlag München, 2006. – ISBN 3–446–40434–1

[BFN03] Bundrock, Jan ; Fruggel, Olaf ; Nitschke, Dirk: *Die Rendering-Pipeline*. September 2003. – Humboldt-Universität zu Berlin. Institut für Informatik.

[Bie08] Biedert, Tim: *Die Render-Pipeline*. April 2008

[CRMW10] Cooper, Simon ; Rhalibi, Prof. Abdennour E. ; Merabti, Prof. M. ; Wetherall, Jon: Procedural Content Generation and Level Design for Computer Games, 2010

[DFM+] Dylla, Kimberly ; Frischer, Bernard ; Mueller, Pascal ; Ulmer, Andreas ; Haegler, Simon: *Rome Reborn 2.0: A Case Study of Virtual City Reconstruction Using Procedural Modeling Techniques.*. – Virtual World Heritage Laboratory, University of Virginia, USA.

[Ebs05] Ebser, Robin: *Die programmierbare Hardware-Pipeline und ihre Verwendung für glaubhafte künstliche Charaktere. Fachhochschule Stuttgart - Hochschule der Medien.*, Diplomarbeit, 2005

[Feu10] Feuerstein, Florian: *Entwicklung eines datenbankbasierten Level of Detail Systems zur Beschleunigung von Echtzeitszenen mit zahlreichen Geometriemodellen. B.Sc. Thesis, Fachbereich IEM und MND, Fachhochschule Gießen Friedberg*. September 2010

[FK03] Fernando, Randima ; Kilgard, Mark J.: *The Cg Tutorial - The Definite Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003. – ISBN 0–321–19496–9. – Copyright by NVIDIA Corporation

[Gab11] Gabriel, Paul: *WebGL basierte visuelle Analyse von 3D Geomodellen Technische Universität Bergakademie Freiberg. Institut für Geophysik und Geoinformatik*, Diplomarbeit, September 2011

[HB98] Haala, Norbert ; Brenner, Claus: *Extraction of buildings and trees in urban environments*. September 1998. – Institute for Photogrammetry. University of Stuttgart

[Het98] Hettinga, D.: *Allgemeines zum Baurecht*. 1998

[Jak06] Jakubowski, Peter: Stadt ohne Infrastruktur heißt Stadt ohne Zukunft - Zur Agenda kommunaler Infrastrukturpolitik. In: *Informationen zur Raumentwicklung, Bundesinstitut für Bau-, Stadt- und Raumforschung* Heft 2006 (2006), Nr. 5, S. 237–248

[JC98] Johannsen, Andreas ; Carter, Michael B.: Clustered Backface Culling. In: *Journal of Graphics Tools* 3 (1998), Nr. 1, S. 1–14

[Lie08] Lieckfeldt, Peter: *Herstellung von 3D-Stadtmodellen*. April 2008. – GTA Geoinformatik GmbH. 9. Norddeutsche Fachtage, Hochschule Neubrandenburg

[LJ02] Lewis, Michael ; Jacobson, Jeffrey: Game Engines in Scientific Research. In: *Communications Of the ACM* 45 (2002), Nr. 1, S. 27–31

[MP01] Müller, Pascal ; Parish, Yoav I H.: Procedural Modeling of Cities. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, S. 301–308. – Siggraph 2001

[Nol05] Nold, René: *Entwicklung einer interaktiven 3D-Echtzeitapplikation zur Simulation von Wetterflügen. Fachbereiche IEM, MND und MNI. Fachhochschule Gießen Friedberg.*, Diplomarbeit, Juli 2005

[NVI03] NVIDIA, Corporation: *Using Vertex Buffer Objects (VBOs)*. Santa Clara, California, USA, Oktober 2003

[NVI12] NVIDIA, Corporation: *NVIDIA Developer Zone*. Santa Clara, California, USA : `http://developer.nvidia.com`, Juni 2012

[Rus08] Rusdorf, Dipl.-Inf. S.: *Aspekte der Echtzeit-Interaktion mit virtuellen Umgebungen*, Diss., März 2008

[Sch12] Schüler, Peter: Landkarten aus dem Web - Schöne neue Welt-Bilder. In: *c't magazin* 2012 (2012), Nr. 11, S. 138–145

[Sud05] Sudmann, Oliver: *Interaktive Visualisierung der speziellen Relativitätstheorie auf programmierbarer Grafikhardware Institut für Informatik. Universität Paderborn.*, Diplomarbeit, November 2005

[Teo08] Teoh, Soon T.: *Algorithms for the Automatic Generation of Urban Streets and Buildings.* 2008. – Department of Computer Science, San Jose State University, California, USA

[TNAK03] Takase, Y. ; N., Sho ; A., Sone ; K., Shimiya: Automatic Generation of 3D-City Models and related Applications. In: *Journal of the Visualization Society of Japan* 23 (2003), Nr. 88, S. 21–27

[Wag11] Wagner, Matthias: *Untersuchung zur Performance-Optimierung von 3D-Fahrzeugmodellen im Bereich der fotorealistischen Bildgenerierung für Fahrerassistenzsystemtests im Automotive Bereich. B.Sc. Thesis, Fachbereich IEM, Fachhochschule Gießen Friedberg.* Januar 2011

[Wei] Weidner, Uwe: *Digital Surface Models for Building Extraction.* – Institut für Photogrammetrie. Rheinische Friedrich–Wilhelms-Universitat Bonn

[WFRK] Wells, Sarah ; Frischer, Bernard ; Ross, Doug ; Keller, Chad: *Rome Reborn in Google Earth.* – Virtual World Heritage Laboratory, University of Virginia, USA.

[Wlo03] Wloka, Matthias: *Batch, Batch, Batch: What Does It Really Mean?* März 2003. – NVIDIA Corporation. Game Developers Conference