Entwicklung eines datenbankbasierten „Level of Detail“ Systems zur Beschleunigung von Echtzeitszenen mit zahlreichen Geometriemodellen

Fachbereiche IEM und MND der
Fachhochschule Gießen-Friedberg

Bachelorarbeit

vorgelegt von

Florian Feuerstein
geb. in Bad Nauheim

Referent der Arbeit:	Prof. Dr.-Ing. Cornelius Malerczyk
Korreferent der Arbeit:	Dipl.-Inf.(FH) Christian Rathemacher

Fachbereiche

Informationstechnik-Elektrotechnik-Mechatronik IEM
und
Mathematik, Naturwissenschaften und Datenverarbeitung MND

Friedberg, 09.09.2010

Danksagung

Verschiedene Personen haben zum Gelingen dieser Arbeit beigetragen, bei denen ich mich an dieser Stelle ganz herzlich bedanken möchte. Vielen Dank an Prof. Dr.-Ing. Cornelius Malerczyk, der mich während meines Studiums immer begleitet hat. Bei meinem Betreuer Dipl.-Inf.(FH) Christian Rathemacher möchte ich mich für die guten Ratschläge und die vielen beantworteten Fragen besonders bedanken. Des Weiteren danke ich Daniel Pielok für die informativen Gespräche auf dem Nachhauseweg und Réne Nold für die Aufnahme in das Weltenbauer-Team.

Lieben Dank an meine Freunde, die auch in schwereren Zeiten hinter mir stehen. Bei meiner Freundin Julia Olusoga möchte ich mich für die Unterstützung und sämtlichen Ansporn bedanken. Vor allem danke ich vielmals meiner Familie, meinen Eltern und Großeltern, die mich während meines ganzen Lebens in jeder Hinsicht unterstützt haben und ohne die ein Studium erst gar nicht möglich gewesen wäre.

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.

Friedberg, September 2010

Florian Feuerstein

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Interaktive Anwendungen zur Visualisierung bestimmter Inhalte sind heute ein Standard, der sich über die letzten Jahre erfolgreich durchgesetzt hat. Die Rede ist hier von virtuellen Welten oder der virtuellen Realität, wobei der Begriff genauso viele Definitionen besitzt, wie es Autoren dafür gibt. Der Brockhaus [Bro97] definiert virtuelle Realität als eine mittels Computer simulierte Wirklichkeit oder künstliche Welt, in die Personen mithilfe technischer Geräte sowie umfangreicher Software versetzt und interaktiv eingebunden werden. Damit grenzt sich die Definition von computergenerierten Bildern und Filmen ab, da diese Medien noch keine Interaktion mit Personen erlauben.

Die aktive Position des Benutzers, auf der die Anwendungen basieren, ist genau die Ursache für die steigende Beliebtheit [Hee05]. Für die Interaktion zwischen Mensch und Maschine läutet sie sogar ein neues Zeitalter ein [Dis03]. Statt dem besten Freund ein Bild des letzten Urlaubsortes zu geben, wäre es für ihn viel anschaulicher und einprägender sich selbst virtuell durch den Ort zu bewegen. Er wird die Informationen schneller aufnehmen und verarbeiten, als bei der Betrachtung eines Bildes. Sogar der Wissenserwerb kann mithilfe dieser Methode gefördert werden [Pri03]. Der Grund ist die Kommunikation zwischen Mensch und Maschine, die sich auf mehreren Kanälen unterschiedlicher Intensität abspielt. Eine perfekte virtuelle Welt würde den Menschen auf gleiche Art und Weise wie in der wirklichen Welt auf allen fünf Sinnen ansprechen [Cho02]. Davon ist die Technik aber noch weit entfernt und in den meisten Fällen beschränkt sich die Kommunikation auf die audiovisuelle Ebene. Doch diese Sinneskommunikation reicht aus, um einer Person das Gefühl zu geben tatsächlich in der computergenerierten Welt präsent zu sein. Das Fachwort für diese Empfindung nennt sich Immersion und beschreibt das Eintauchen in die fiktive Welt [Hee05]. Nun ist die Immersion für eine interaktive, virtuelle Welt viel größer als die eines Bildes oder Filmes, weshalb das Ausmaß der Anwendungsgebiete ständig wächst.

Medium	Darstellungsdimensionen	Interaktivität	Visualisierung abstrakter Konzepte	Präzision der Darstellung abstrakter Konzepte	Individualisierung des Aneignungsprozesses
Lehrbuch	2D	Kaum möglich	Primär durch 2D-Graphiken	Mäßig	Eher Selektion möglich, da lineare Präsentation
Multimedia und Hypermedia	2D	In begrenztem Umfang möglich	Primär durch 2D-Graphiken, interaktive Graphiken sind möglich	Gut	Individuell möglich
Virtuelle Umgebung	3D	i.d.R. vorhanden; sehr umfassend möglich	Interaktive 3D-Graphiken sind möglich	Gut bis sehr gut	Individuell möglich

Abbildung 1.1: Formen der Wissensvermittlung mit verschiedenen Medien. Entnommen aus [Hee05].

1.1 Motivation

Ursprünglich wurden virtuelle Umgebungen vor allem für den industriellen und militärischen Bereich entwickelt [Hee05]. Die Anwendungen dienten zur Simulation, zum Beispiel von Fahr- und Flugzeugen. Heute werden damit hauptsächlich Videospiele in Verbindung gebracht. Das Internet ist überflutet von kostenlosen Angeboten, in denen sich tausende Spieler in virtuellen Welten treffen und agieren können[1]. Konfiguratoren für Autos, Küchen oder den kompletten Familienwohnsitz sind ebenfalls beliebte Anwendungsmöglichkeiten[2]. Die Benutzer können nach ihren persönlichen Vorlieben Inhalte verändern und bekommen sofort einen realistischen Eindruck des Ergebnisses zurück. Visualisierungen von Innenarchitektur oder Gebäudeplänen können dem Kunden schon im Voraus einen Eindruck der Räumlichkeiten geben. Aber auch komplette Stadtmodelle oder sogar vergangene Zeitalter lassen sich darstellen. Viele historische Gebäude sowie Städte werden virtuell wieder zum Leben erweckt und als Wissensmedium gebraucht [Hee05]. Eine virtuelle Welt lässt sich also auch als Lernplattform anwenden. Durch den dreidimensionalen Aufbau, der anschaulichen Darstellung und der Interaktivität werden zum Beispiel komplizierte Zusammenhänge von Molekülen besser aufgefasst. Gerade wenn es um Größen, Distanz, Zeit, Raum und Veränderung geht, ist eine interaktive, virtuelle Umgebung eine gute Wahl. Abbildung 1.1 zeigt einige Vorteile auf.

[1] kostenloses Onlinespiel für Kinder: http://fusionfall.cartoonnetwork.com/
[2] Konfigurator für Gebäudeszenen: http://www.scenecaster.com/web/home.php

Abbildung 1.2: Screenshots aus verschiedenen Computerspielen. Von links nach rechts: Crysis, Crysis2, Red Dead Redemption.

Um solche Anwendungen zu realisieren müssen die entsprechenden geometrischen Modelle der Umgebung vorhanden sein. Diese werden dann durch Algorithmen mit einem Computerprogramm auf dem Bildschirm angezeigt und mit Interaktivität ausgestattet. Hierbei ist das Ziel oft die höchstmögliche Immersion für den Benutzer zu erzeugen, denn umso mehr Kanäle miteinander kommunizieren und umso realistischer deren Umsetzung, desto stärker fühlt man sich in die virtuelle Welt versetzt. Das beste Beispiel dafür sind aktuelle Computerspiele[3] (Abbildung 1.2), welche die Hardware voll ausreizen. Ist es jedoch möglich die Realität auf der visuellen Ebene perfekt nachzubilden oder gibt es Grenzen?

1.2 Problemstellung und Zielsetzung

Für die Realitätstreue bei computergenerierten Bildern in interaktiven, virtuellen Welten gibt es Grenzen und diese hängen von der Leistung des Computers ab. Solch eine Welt wird in Echtzeit berechnet und angezeigt [Bri09]. Das bedeutet, die Anwendung muss innerhalb einer bestimmten Zeitspanne ein Ergebnis für die Verarbeitung von anfallenden Daten zurückgeben. Erfüllt das Programm die Bedingung, redet man von einem Echtzeitsystem oder einer Echtzeitanwendung [Zöb08]. In einer virtuellen Welt besteht ebenfalls eine zeitkritische Bedingung. Die Voraussetzung ist eine flüssige Darstellung ohne Verzögerungen des Bildes zu erreichen. Um eine Bewegung als fließend zu erkennen werden mindestens 25 Bilder pro Sekunde benötigt, was eine Zeitspanne von ungefähr 40 Millisekunden pro Bild zur Folge hat. Die zeitliche Bedingung des Echtzeitsystems wird also von der menschlichen Physiologie bestimmt [Bri09]. Der Computer muss innerhalb dieser Dauer das nächste Bild berechnen. In der Computergrafik wird ein Bild auch als Frame bezeichnet. Um zu wissen, was berechnet werden soll, fängt man zuerst die Benutzereingaben ab und anhand dieser Daten wird das Bild generiert. Drückt man beispielsweise die Taste für die Vorwärtsbewegung, wird die virtuelle Kamera in der Szene nach vorne versetzt und das nächste Bild berechnet. Je mehr Geometriemodelle in der Szene vorhanden sind und je komplexer sich die Interaktivität

[3] Bilder aus http://www.ea.com/games/crysis und http://www.sosnewyork.com/ und http://www.rockstargames.com/reddeadredemption/agegate/ref=/.

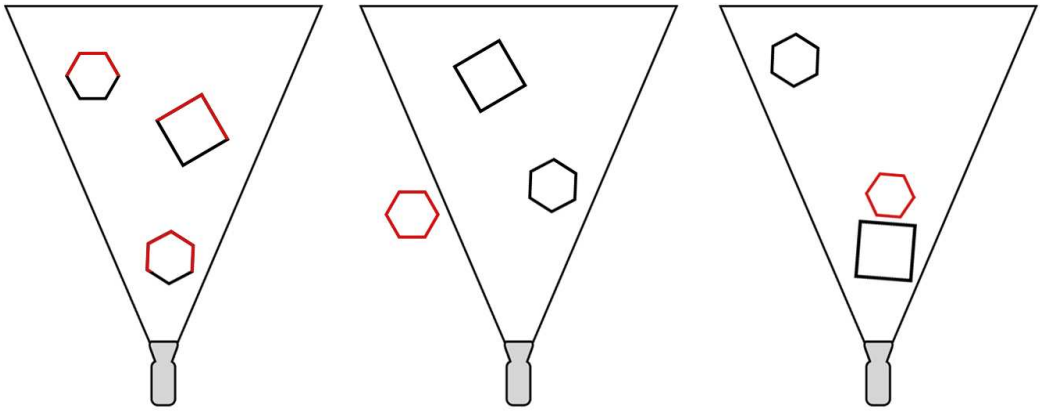

Abbildung 1.3: Von links nach rechts: Back Face Culling, View Frustum Culling und Occlusion Culling. Die in rot markierten Geometrien werden in der Bildberechnung nicht berücksichtigt.

gestaltet, desto höher ist die Rechenlast des Systems. Damit gerenderte Bilder eine hohe Realitätsnähe erreichen, müssen Modelle und Texturen alle Details enthalten, wie sie auch in der Natur vorkommen. Diese Genauigkeiten führen einmal zu einer großen Datenmenge, die ebenso viel Speicher verbraucht, und zu einer längeren Renderzeit für den Computer, da mehr Informationen bearbeitet werden müssen.

In den letzten Jahren wurden jede Menge Algorithmen mit unterschiedlichen Ansätzen zur Verringerung der Renderzeit entwickelt [TAMH08]. Dabei werden immer nicht sichtbare Geometrien gesucht und vom Renderprozess ausgeschlossen. Zum Beispiel kann eine Fläche, deren Normale nicht zur Kamera zeigt (Back Face Culling), oder ganze Objekte, die sich nicht im Sichtbereich der Kamera befinden (View Frustum Culling), bei der Bildberechnung wegfallen [AM99]. Objekte, die vollständig hinter anderen verschwinden (Occlusion Culling), können ebenfalls ignoriert werden [SC97] (Abbildung 1.3). Eine weitere Methode nennt sich „Level of Detail“(LOD) und besteht darin unwichtige sowie nicht sichtbare Informationen wegzulassen oder zu vereinfachen. Bei weit entfernten Objekten kann das Auge viele Details nicht mehr auflösen. Warum sollte man sich dann die Mühe machen sie zu berechnen? Innerhalb des LOD-Bereichs gibt es erneut verschiedene Lösungen zu spezifischen Problematiken. Die LOD-Verfahren basieren alle darauf, im Vorfeld des Renderprozesses noch mehr Daten zu halten, um dann nur gezielt ausgewählte davon an den Renderprozess zu delegieren, oder durch weitere Berechnungen die Details soweit zu minimieren, dass die Rechenzeit des LOD-Algorithmus ausschlaggebend kleiner als des Renderprozesses ist.

Umfasst eine Szene zahlreiche Objekte, so hat das einen negativen Einfluss für den Speicher als auch auf die Performance der Anwendung. Für das Verwalten von großen Mengen an Daten in Echtzeitvisualisierungen, auch Szenenmanagement genannt, gibt es ebenfalls Methoden, die sich vor allem auf die Speicher- und Performanceoptimierung von stark ver-

deckten Szenen, wie Innenräume, konzentrieren. In solchen Fällen werden viele Objekte durch geschlossene Wände verdeckt und brauchen nicht geladen zu werden. Durch die Verwendung einer Datenverwaltung, wie es das Szenenmanagemen darstellt, ist es möglich auch komplexere Szenen ohne Ladeunterbrechungen anzuzeigen.

Eine weitere spezielle Problematik ergibt sich bei Szenen, die zur Visualisierung von Städten oder Ähnlichem dienen und mit enorm vielen Geometriemodellen sowie Texturen ausgestattet sind. Einige besondere Eigenschaften kennzeichnen diese Art von Szenen. Erstens umfassen sie aufgrund der weiten Dimensionen noch mehr Daten, sodass es für einen heutigen Computer unmöglich ist den ganzen Umfang gleichzeitig im Speicher zu halten. Zweitens ist es eine sehr offene Welt mit wenig geschlossenen bzw. sichtgeschützten Orten, aber viele Höhen und Tiefen, sodass sich die Aussichten stark voneinander unterscheiden. Drittens hat jede Stadt ihren eigenen Charakter, dass heißt viele Häußer ähneln sich in ihrer Erscheinung, was die Grundlage für eine Optimierung bildet. Gleichen sich verschiedene Objekte in einem gewissen Maß, bedeutet es, dass sie auch teilweise gleiche Informationen tragen. Diese redundanten Daten kann man mithilfe einer Optimierung beseitigen und Speicherplatz sparen.

Ziel dieser Bachelorarbeit ist es mit der Basis eines Stadtmodells ein System zu entwickeln, welches ein „Level of Detail“ System mit einer Datenverwaltung verknüpft. Somit wird die Renderzeit für zahlreiche Geometrien beschränkt und nur die aktuell relevanten Objekte der Szene abhängig vom Kamerastandort geladen. Hierfür sollen zuerst die Rohdaten der Stadtszene untersucht und optimiert in einer Datenbank abgelegt werden. Das datenbankbasierte LOD-System soll dann zur Laufzeit die hohe Anzahl an 3D-Objekten performant sowie speichersparend verwalten und anzeigen. Hierbei gilt die Voraussetzung, dass für das berechnete Bild nur so wenige Informationen verloren gehen, dass der Qualitätsverlust für das menschliche Auge kaum wahrnehmbar ist. Das System muss anhand von definierten Regeln eigenständig erkennen können, welches Objekt in welchem Detailgrad geladen oder zum Rendern freigegeben wird. Die Eigenschaftswerte der Regeln sollen veränderbar sein, sodass eine individuelle Anpassung für jedes Projekt gewährleistet wird. Welche Methoden für die grundlegenden Modelle am besten geeignet sind, gilt es in dieser Arbeit herauszufinden.

Das Stadtmodell, welches dabei in eine interaktive, virtuelle Welt umgesetzt werden soll, entstand in dem Projekt Rome Reborn[4]. Es stellt das antike Rom von der ersten Besiedlung in der späten Bronzezeit (ca. 1000 v. Chr.) bis zur Verödung der Stadt im frühen Mittelalter (ca. 550 n. Chr.) dar. Man begann die Modellierung für das Jahr 320 n. Chr., da Rom zu diesem Zeitpunkt die Spitze der Population erreicht hatte und die ersten christlichen Kirchen erbaut wurden (Abbildung 1.4). Mittlerweile umfasst das Projekt alleine über 20.000 Gebäude, darunter historische Bauwerke, wie das Forum Romanum, das Colosseum und der Circus Maximus.

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

Abbildung 1.4: Bild aus dem Rome Reborn Projekt.

Der ursprüngliche Zweck von Rome Reborn ist all das Wissen über die Stadt hinsichtlich der Infrastruktur und Topografie zusammenzufassen, um Informationen als auch Theorien über das Erscheinungsbild des alten Roms zu präsentieren [Fri08]. Dieser Gedanke soll auch in der virtuellen Welt als Lernplattform aufgegriffen werden. Die Verknüpfung der Informationen mit den dreidimensionalen Objekten passiert in einer Datenbank, die gleichzeitig als Informationsbasis der Anwendung dient. Als zusätzliche Informationen könnten beispielsweise historische Fakten oder Metadaten des Objekts gespeichert werden. Diese werden dann in der Echtzeitanwendung vom Benutzer abgerufen und vermitteln ihm das entsprechende Wissen. Besonders wichtig ist die Trennung der Objektdatenbank vom LOD-System, damit beide Teile unabhängig voneinander ausgetauscht werden können und somit auch wiederverwendbar sind. Die Objektdatenbank soll ein einfaches Bearbeiten einzelner oder mehrerer Objekte erlauben, sodass zu jedem Zeitpunkt eine Veränderung der Datenbasis möglich ist, ohne parallel das System zu verändern.

1.3 Organisation der Arbeit

Die vorliegende Arbeit ist folgendermaßen strukturiert: Das 2. Kapitel behandelt die Grundlagen und den Stand der Technik mit den bekannten LOD- und Szenenmanagementsystemen, welche Methoden sie benutzen und für welche Anwendungen sie bevorzugt zum Einsatz kommen. Desweiteren werden Techniken beschrieben, die oft in Verbindung mit diesen eingesetzt werden.

Im 3. Kapitel befinden sich die Grundlagen über die Echtzeitberechnung von computergenerierten Bildern. Es wird die Pipeline dargestellt, welche die Daten auf dem Weg zum Bildschirm durchlaufen und die Software vorgestellt, mit der sich interaktive, virtuelle Welten

entwickeln lassen.

Das 4. Kapitel beschreibt die besondere Datenbasis, auf der das LOD-System aufsetzen soll sowie deren Erstellung.

Nach allen Grundlagen wird in Kapitel 5 zuerst das Konzept des datenbankbasierten LOD-Systems erläutert, dass heißt, wie die Datenbasis erstellt, in der Objektdatenbank organisiert und optimiert auf dem Bildschirm angezeigt wird. Danach behandelt es die konkrete Implementierung in der Entwicklungsumgebung und deren Eigenheiten. Dabei wird nochmals auf alle Arbeitsschritte vom Export der ursprünglichen Daten bis zum Anzeigen der optimierten Daten eingegangen.

Im 6. Kapitel wird die Implementierung des datenbankbasierten LOD-Systems anhand einer Laufzeitanalyse hinsichtlich der Leistung, des Speicherbedarfs und der Bildqualität bewertet sowie die Auswirkungen auf die Anwendung mit verschiedenen Einstellungen verglichen.

Zuletzt fasst das 7. Kapitel die wesentlichen Aspekte dieser Arbeit zusammen und hebt die Ergebnisse der Evaluation hervor. Außerdem werden Anregungen zur weiteren Entwicklung und Optimierung der Anwendung aufgeführt.

1.4 Zusammenfassung der wichtigsten Ergebnisse

Im Rahmen dieser Arbeit wird auf der Grundlage eines Stadtmodells mit zahlreichen Objekten ein „Level of Detail“ mit einem Szenenmanagementsystem verknüpft, um erstens die Darstellung der vollständigen Szene ohne Ladeunterbrechungen zu ermöglichen und zweitens den Renderprozess zu beschleunigen. Durch die gewonnene Zeit kann man zusätzliche Modelle darstellen, die wiederum die Bildqualität verbessern. Dabei wird das Kriterium der Interaktivität, welches man hier auf durchschnittlich 25 Bilder pro Sekunde setzt, nicht verletzt. Außerdem wird zum Zweck der Wissensvermittlung eine Datenbank aufgebaut, in der sich diverse Informationen zu einzelnen Objekten aus der fertigen Anwendung heraus hinterlegen, verändern und abrufen lassen.

Das Stadtmodell des alten Roms, welches dafür zum Einsatz kommt, entstand überwiegend durch die prozedurale Generierung von Gebäudemodellen. Bei dieser Erstellung von insgesamt über 22.000 Objekten wird jeweils zuerst die Grundform, also die äußeren Umrisse, des Hauses mit wenigen Vertexpunkten erzeugt. Darauf bestimmen benutzerdefinierte Regeln die Erstellung von drei weiteren Modellen für das selbe Objekt. Dieser prozedurale Vorgang teilt die Grundformen, insbesondere die Hauswände, in Kacheln und füllt diese mit Texturen von Wänden, Fenstern und Türen. Dabei wird für alle Modelle aus dem selben Texturvorrat gegriffen. Diese Eigenschaften der Datenbasis werden im entwickelten System berücksichtigt und führen zur Optimierung des Speichers sowie der Performance. Die pro

Objekt generierten Modelle sind bereits LOD Stufen des Hauses in drei unterschiedlich hohen Auflösungen. Sie werden für ein kollisionsbasiertes, diskretes LOD System verwendet, dass die Modelle bei verschiedenen Distanzen zur virtuellen Kamera anzeigt. Dabei wird jedoch nicht der aktuelle Abstand aller Objekte zur Kamera berechnet, da dies zu einem frühzeitigen oder verspäteten LOD Austausch führen kann, den der Benutzer schneller wahrnehmen würde. Das entwickelte LOD System, wie auch das Szenenmanagement, basieren auf Kollisionsereignissen mit Platzhaltern für die Szenenobjekte.

Das LOD und Szenenmanagementsystem besteht aus den Phasen der Objektdatenbankerstellung, Szenenvorbereitung und dem Laufzeitpozess. Während die ersten beiden Phasen nur einmalig ausgeführt werden müssen, findet der Laufzeitprozess bei jedem Start der Anwendung statt. Bei der Erstellung der Objektdatenbank werden redundante Daten, hier die Materialien und Texturen, von den Modellen getrennt und in separate Dateien mit einer eindeutigen ID gespeichert. Mit den einzelnen LOD Modellen wird ebenso verfahren. Dabei werden die benötigten IDs der Materialien oder Texturen in der davon abhängigen Datei gehalten. Zur Laufzeit der Anwendung kann man so alle Daten wieder zusammenbringen. Im Abschnitt der Szenenvorbereitung wird die leere Szene neben dem Gelände mit den Grundformen aller Gebäude gefüllt. Sie werden in der Anwendung nicht gerendert, sondern dienen als Kollisionsobjekte. Zusätzlich werden drei Sphären mit unterschiedlichen Radien als Kollisionsobjekte um die Kamera gehängt, die sich analog zu ihr bewegen. Die Sphären definieren die Bereiche, in dem die jeweiligen LOD Stufen geladen plus angezeigt werden und besitzen auch eine entsprechende ID. Ein Nebeleffekt der virtuellen Kamera lässt Objekte hinter der größten Sphäre vollständig verschwinden. So brauchen zur Laufzeit auch keine Modelle außerhalb der Sphären geladen und angezeigt werden. Zuletzt wird eine hierarchische Datenstruktur für alle Kollisionsobjekte angelegt, welche die benötigten Schnittberechnung wesentlich beschleunigt.

Die Grundidee des Laufzeitverfahrens ist das Laden und Anzeigen des entsprechenden LOD Modells in der Objektdatenbank bei der Kollision einer Sphäre mit einer Grundform. Sobald eine Kollision statt findet, wird eine Ladeanfrage mit der ID des zu ladenden LOD Modells an einen Queue geschickt. Dieser verarbeitet und verwaltet alle Anfragen bezüglich der aktuellen Umstände, wodurch unnötige Ladeoperationen verhindert werden. Die bedeutende Eigenschaft des Ladequeues ist, dass er eine benutzerdefinierte Menge an parallelen Ladeoperationen ausführen kann, wodurch interaktive Frameraten erhalten bleiben. Da die Renderpipeline bereits auf Softwareebene startet, lässt sich diese durch parallele Programmierung erheblich beschleunigen. Während der Hauptprozess, also die Anwendung, weiter läuft, können im Hintergrund benötigte Daten geladen werden. Ist ein Ladevorgang beendet, wird das enthaltende Modell in der Szene instanziiert, mit seinen vorgeladenen Materialien und Texturen verknüpft und für den Renderprozess freigegeben. Alle anderen bereits instanziierten LOD Modelle des selben Objekts setzt man auf unsichtbar. Verlässt die letzte Sphäre eine Grundform, so werden alle instanziierten Repräsentationen des Objekts zerstört und die Dateien entladen, was Speicherplatz für neue Modelle freilässt. Mithilfe des Queues werden die dringlichsten Modelle zuerst geladen und das Stadtmodell abhängig vom Kamerastandpunkt schrittweise an die aktuelle Situation angepasst. Außerdem wird das Szenen-

managementsystem mit dem LOD System verbunden, indem man nur jeweils das aktuell ausreichende LOD Modell eines Objekts lädt und anzeigt. Durch dieses Verfahren reduziert man die Rechenzeit zur Echtzeitdarstellung einer Szene mit zahlreichen Geometriemodellen sowie den dafür benötigten Speicherplatz.

Für das Verknüpfen von Informationen mit einzelnen Objekten wird eine zentrale Datenbank auf einem Webserver angelegt, die in der Anwendung als Wissenvermittlung dient. Dabei vernetzt eine Tabelle die ID eines Objekts mit verschiedenen Hintergrundinformationen. Den Zugriff auf die Datenbank, hinsichtlich des Abrufs, der Erstellung und Veränderung von Datensätzen, steuern Programme auf dem Webserver. Die Webserverprogramme können dann von mehreren Instanzen der Anwendung gleichzeitig aufgerufen werden. Somit ist die Informationsdatenbank für jeden Benutzer aktuell und zugleich aus der Anwendung heraus editierbar.

Umgesetzt wird das Konzept mit der Game-Engine Unity3D. Die Software bietet für das Auslagern von Szeneninhalten und das dynamische Laden zur Laufzeit sogenannte Asset-Bundles an. Die Kollisionsberechnungen finden im Physiksystem der Middleware statt und die Anwendungslogiken bezüglich des LOD und Szenenmanagementsystems implementiert man durch eigens geskriptete Komponenten. Um das ganze Verfahren zu testen, werden mithilfe der Game-Engine ausführbare Programme erstellt, in denen man die Eigenschaften des Systems verändert und die Auswirkungen auf die Anwendung hinsichtlich der Performance, Speicherauslastung und Bildqualität beobachtet. Ohne jegliche Optimierungen ist bei ca. 200 hochaufgelösten Modellen keine Interaktivität mehr möglich. Das heißt die Bildrate pro Sekunde sinkt unter 25 Hz. Mithilfe des Szenenmanagmentsystems ist die Darstellung der kompletten Szene mit einem Radius von 60 Einheiten bei einer Framerate von 23 Hz möglich. Die errechnete Bildqualität dafür liegt bei 10800 Einheiten und der benötigte Arbeitsspeicher ist 655 MB. Durch die Verbindung mit dem LOD System kann bei einer Framerate von 25 Hz mehr als die doppelte Bildqualität erlangt werden. Diese Verbesserung geht jedoch leicht auf die Kosten des Speicherplatzes, da für viele Objekte mehrere LOD Modelle geladen sind.

Kapitel 2

Level of Detail und Szenenmanagement

Dieses Kapitel beschäftigt sich mit den Grundlagen und dem Stand der Technik von LOD- sowie Szenenmanagementsystemen. Beide Methoden sind in der Computergrafik ein altbekanntes und zugleich ein aktuelles Thema. Schon 1976 beschrieb James Clark [Cla76] die Vorteile, Objekte in einer Szene in verschiedenen Auflösungen anzuzeigen und wie man sie leistungsoptimiert in einer Szenenstruktur verwaltet. Heute haben sich diese Methoden längst durchgesetzt und wurden für viele Teilgebiete verfeinert. Die Basisidee eines Level of Detail ist jedoch immer ein Objekt in einer umso simpleren Variante anzuzeigen, desto weniger Beitrag es an einem Bild leistet. Im Allgemeinen besteht ein LOD Algorithmus aus drei Schritten: Erstellung, Auswahl und Austausch [TAMH08]. Die LOD Erstellung ist der Teil, in dem die verschiedenen Darstellungen eines Modells in einem unterschiedlichen Detailgrad erzeugt werden. Hierbei können die Modelle beispielsweise alle per Hand oder durch bestimmte Regeln automatisch mit einem Programm erstellt werden. Der Auswahlmechanismus bestimmt dann zur Laufzeit die Detailstufe eines Objekts mithilfe von festgelegten Kriterien, wie dem Abstand von der Kamera oder der Größe auf dem Bild. Zuletzt muss die aktuelle Detailstufe mit der neuen ausgewählten Detailstufe ausgetauscht werden. Der LOD Austausch bestimmt auf welche Weise die Modelle umgeschaltet werden. In den letzten Jahren wurden vor allem für die Auswahl und den Austausch von LODs einige unterschiedliche Techniken entwickelt, die auch auf verschiedene Problemstellungen angewandt werden. Diese Techniken werden in dem Kapitel genauer betrachtet, um später zu entscheiden, welche für die Datenbasis von Stadtmodellen eine Grundlage bilden könnte und welche zusätzlichen Optimierungen darauf aufsetzen.

Nimmt eine interaktive, virtuelle Welt zu viel Speicher ein, um im Ganzen geladen zu werden, wird oft auf die Unterteilung in Teilszenen zurückgegriffen. Beim Wechsel von einem Abschnitt zum nächsten muss dieser dann erst vorgeladen werden, wodurch ein unschöner Bruch entsteht. Vor allem bei Computerspielen kennt man den Ladebalken, der nach Beendigung den nächsten Level freigibt. Um eine virtuelle Welt möglichst ohne Abbruch zu realisieren, müssen also zukünftige, relevante Daten geladen werden, ohne dass der Benut-

zer den Prozess bemerkt. Die Methode nennt sich Sichtbarkeitsvorbearbeitung [SJT91] und wird ebenfalls in dem Kapitel erleutert.
Außerdem wird genauer auf hierarchische Datenstrukturen und Nebel eingegangen, da diese beiden Techniken fast immer in Verbindung mit LODs und Szenenmanagement stehen.

2.1 Hierarchische Datenstrukturen

Hierarchische Datenstrukturen werden überall da benutzt, wo eine große Anzahl an Objekten räumlich aufgeteilt sind und nach bestimmten Objekten gesucht wird. Sie organisieren Geometrie in einem n-dimensionalen Raum, wobei hier nur zwei- und dreidimensionale Strukturen beschrieben werden. Dieser Raum wird in Unterräume zerlegt, auf die sich die Objekte aufteilen. Der Vorgang zieht sich dann rekursiv bis zu einer Abbruchbedingung fort. Eine Abbruchsbedingung könnte eine festgelegte Anzahl an Objekten in einem Unterraum sein oder ein Limit, wie oft eine Unterteilung stattfinden darf. In virtuellen Realitäten finden hierarchische Datenstrukturen bei der Sichtbarkeitsbestimmung [SJT91], Kollisionsbestimmung [RGL05] und auch bei vielen LODs ihren Einsatz [MD97, Lev02, CZ02]. Der Hauptgrund für die Verwendung von Datenstrukturen ist, dass die benötigten Schnittabfragen zur Kollisions- oder Sichtbarkeitsbestimmung ein schnelleres Ergebnis zurückliefern. Bei der Kollisionserkennung wird z. B. die meiste Zeit in die Schnittpunktberechnung von Objekten investiert, die sich weit auseinander befinden und gar nicht kollidieren. Für n bewegte Objekte in einer Szene hat die Kollisionsberechnung für ein Objekt eine Komplexität von $O(n-1)$. Mithilfe der Hierarchie werden aber nur die Objekte geprüft, welche sich in der Nähe befinden[1]. Die meisten Datenstrukturen leisten eine signifikante Verbesserung auf $O(log(n))$ [TAMH08]. Zwei der bekanntesten hierarchischen Strukturen, die Bounding Volume Hierarchies (BVH) und Binary Space Partitioning Bäume (BSP Bäume), sollen hier vorgestellt werden.

Eine Bounding Volume Hierarchie ist eine Baumstruktur für geometrische Objekte. Jedes Objekt wird in einem Blatt innerhalb des Baumes gehalten. Dabei besteht der Baum aus Knoten, die wiederum aus Knoten bestehen können bis man zu einem Blatt mit dem entsprechenden Objekt gelangt [Hav04]. Ein Bounding Volumen (BV) ist ein Volumen, welches eine Anzahl von Objekten komplett umschließt und einen Knoten des Baums repräsentiert. Die Idee dahinter ist, dass ein BV eine sehr viel simplere geometrische Form als die umfassten Objekte hat und so verschiedene Tests im Gegensatz zu den beinhalteten Objekten schneller berechnet werden können. Beispiele von BV sind Kugeln und achsenparallele Boxen. Ein BV wird später nicht visuell im Bild zu sehen sein, sondern beschleunigt nur Rechenprozesse.

Bei der Erstellung der Bounding Volume Hierarchie wird zuerst das BV der ganzen Szene berechnet. Dieses Volumen bildet den Wurzelknoten der Hierarchie. Darauf werden naheliegende Objekte zusammengefasst, deren BV berechnet und als Kindknoten gespeichert. Das Prinzip wird solange fortgesetzt bis keine optimierte Zusammenfassung mehr möglich ist (Abbildung 2.1). Der Aufbau des Baums passiert meistens in einem Prozess vor dem

[1] Anschauliche Demonstration zur Kollisionserkennung unter Benutzung einer Datenstruktur: http://lab.polygonal.de/2007/09/09/quadtree-demonstration/

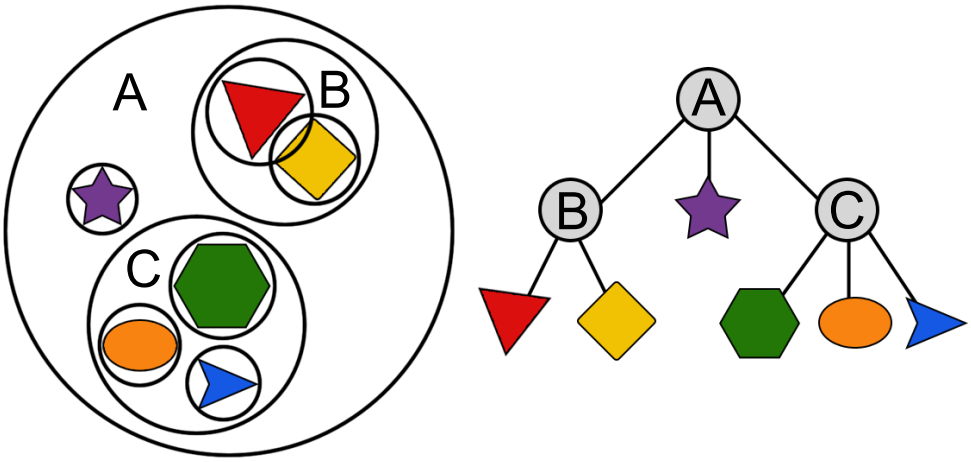

Abbildung 2.1: Diese Szene wurde mit Bounding Volumen unterteilt. Auf der rechten Seite ist der Baum abgebildet, der für die linke Szene generiert wurde. Abbildung nach [TAMH08].

Start der Anwendung, weil er viel Rechenzeit beanspruchen kann. Ändert sich die Position einer Geometrie zur Laufzeit, ist der Baum fehlerhaft und müsste neu berechnet werden. Aus diesem Grund eignen sich hierarchische Datenstrukturen nur für statische Szenen. Es gibt allerdings Verfahren, die auf Veränderungen reagieren und den Baum entsprechend anpassen [TAMH08].

Will man nun die Kollisionen für ein Objekt bestimmen, wird der aufgebaute Baum vom Wurzelknoten ab nach unten auf Schnittpunkte überprüft. Schneiden sich das Objekt und Wurzelknoten nicht, sind keine weiteren Berechnungen mehr nötig, da alle übrigen Objekte innerhalb des BV des Wurzelknotens liegen. Schneiden sie sich doch, werden alle Kindknoten auf Schnittpunkte mit dem Objekt überprüft. Bei einer Kollision testet man wieder die Kindknoten des getroffenen BV. Im Gegensatz zu anderen Raumunterteilungsverfahren, besitzt die BVH die Eigenschaft jedes Objekt nur einmal im Baum zu speichern. Dadurch werden mehrfache Schnittberechnungen für ein Objekt mit demselben Test verhindert.

Eine weitere Methode zur Beschleunigung von Schnittberechnungen ist der BSP Baum. Bei einem BSP Baum werden die Objekte einer Szene nicht hierarchisch durch ein Volumen zusammengefasst, sondern die Szene bzw. der Raum wird hierarchisch durch Schnittebenen unterteilt. Da es ein binärer Baum ist, hat jeder Knoten genau zwei Kindknoten oder gar keinen. Je nachdem wie die Schnittebenen liegen, redet man entweder von einem achsenparallelen BSP Baum oder einem polygonausgerichteten BSP Baum[TAMH08]. Polygonausgerichtete BSP Bäume teilen den Raum ausgehend von den Polygonen in der Szene. Die Position der Schnittebene wird so gewählt, dass sie den Raum an einer Polygonfläche eines geometrischen Objektes schneidet. Bei einem achsenparallenen BSP Baum, auch kd-Baum genannt [Ben75], liegen die Unterteilungsschnitte immer achsenparallel zu den Koordinatenachsen.

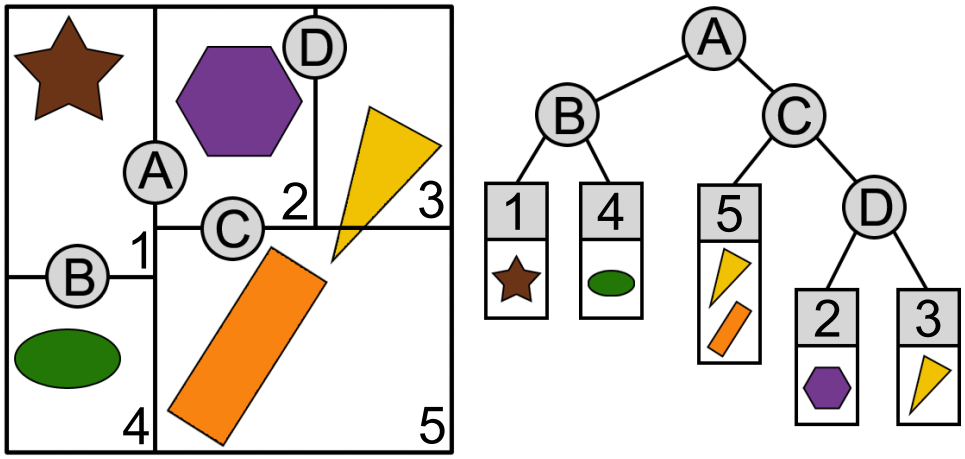

Abbildung 2.2: Ein kd-Baum ist ein achsparalleler BSP-Baum, der die Szene in jedem Schritt durch eine Schnittebene (A-D) in zwei Boxen (1-5) aufteilt. Zu bemerken ist, dass das gelbe Dreieck in zwei Knoten gespeichert wurde, da es auf einer Schnittebenen liegt. Im schlechtesten Fall müssen also zwei Schnittberechnungen mit dieser Geometrie durchgeführt werden. Abbildung nach [TAMH08].

Zur Konstruktion eines kd-Baums wird wieder ein rekursives top-down Verfahren angewandt. Als erstes wird die achsenparallele Bounding Box(AABB) der ganzen Szene sowie jedes einzelnen Objekts berechnet. Die Bounding Box der Szene ist der Wurzelknoten in die als nächstes die erste Schnittebene gelegt wird und den Raum damit in zwei Kindknoten mit den jeweiligen AABB aufteilt. Mithilfe der vorher berechneten AABB für jedes Objekt und den Bounding Boxen der zwei Kindknoten werden die Objekte auf diese verteilt. Überlappt die AABB eines Primitiven die Schnittebene und liegt somit in beiden Knoten, wird das Objekt auch in beiden referenziert. Der Prozess wird wieder für jeden Knoten bis zu einer bestimmten Abbruchsbedingung fortgeführt (Abbildung 2.2). Die Achse und Position der Schnittebene sind wichtig für die maximale Effizienz eines BSP-Baumes. Eine Methode für das Unterteilen einer AABB ist die Schnittachse bei jedem Durchlauf zu wechseln. Der Wurzelknoten wird immer entlang der x-Achse geteilt, dessen Kindern entlang der y-Achse, deren Kinder entlang der z-Achse und wieder von vorne. Eine andere Strategie ist jeden Knoten entlang seiner längsten Seite zu spalten. Auf jeden Fall gilt es Referenzen verschiedener Knoten auf das gleiche Objekt zu vermeiden.

2.2 Sichtbarkeitsvorbearbeitung

Teller und Séquin [SJT91, TAFT92] waren die Ersten, die diese Technik der Sichtbarkeitsvorbearbeitung zum Verwalten von großen Datenmengen in interaktiven Gebäuderundgängen nutzten. Sie basiert auf räumlichen Unterteilungen der Szene (sieht Abschnitt 2.1), Sichtbarkeitsanalysen und einer Datenbank, die alle Szenenobjekte hält. In jedem Frame des virtuellen

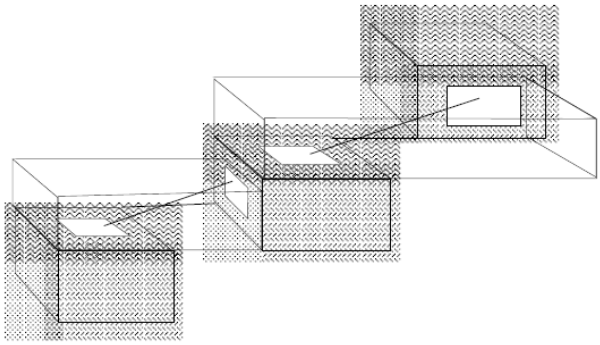

Abbildung 2.3: Drei Zellen und vier Portale mit einer Sichtlinie, die alle Zellen miteinander verbindet, da sie alle Portale durchläuft. Entnommen aus [TAFT92].

Rundgangs wird eine Anzahl an Objekten gerendert, die potenziell vom Betrachter sichtbar sind und ein Set von Objekten in den Arbeitsspeicher geladen, die in naher Zukunft sichtbar werden könnten. Mit dieser Technik werden die unsichtbaren Modelle, welche meistens den größten Teil ausmachen, ausgesondert, da sie für den aktuellen Betrachterstandpunkt irrelevant sind. Der Arbeitsspeicher als auch der Renderprozess werden auf diese Weise entlastet. Die Namensgebung deutet schon darauf hin, dass die Vorhersage von zukünftig sichtbaren Objekten nicht zur Laufzeit berechnet wird, sondern in einem aufwendigen Vorprozess. Aufgrund dessen ist das Verfahren auf statische Szenen beschränkt und kann nicht für Umgebungen eingesetzt werden, deren Objekte ihre Position verändern. Die Sichtbarkeitsvorbearbeitung wird besonders gern innerhalb von Gebäuden realisiert, da in den häufigsten Fällen nur der aktuelle Raum plus die Nachbarräume geladen werden müssen. Viele ähnliche Methoden wurden danach entwickelt [LC99, BC96], die beispielsweise eine andere räumliche Datenstruktur verwenden, das Prinzip der Sichtbarkeitsvorbearbeitung bleibt dabei immer das gleiche.

Generell wird das Verfahren in drei Phasen aufgeteilt. Erstens die Modellierungsphase, in der alle Szenenobjekte konstruiert, positioniert und in der Datenbank abgelegt werden. Mit dieser Basis unterteilt die Vorbearbeitungsphase die Szene in kleinere Räume und führt die Sichtbarkeitsanalyse der Zellen plus eingeschlossener Objekte aus. Die resultierenden Informationen werden in der Datenbank gespeichert, um in der letzten Rundgangsphase von den Anzeige- und Speicherverwaltungsalgorithmen genutzt zu werden. Eine Variante der kd-Baum Datenstruktur (Abschnitt 2.1) teilt das Modell in kleinere Zellen. Dabei werden die Schnittebenen entlang der wichtigsten lichtundurchlässigen Elemente, also Wände, Türen, Böden und Decken, gelegt. Der Prozess endet, wenn genügend achsenparalle, opake Elemente in der Szene koplanar mit einer Schnittebene von mindestens einer Zelle sind. Nach dem Aufbau der Zellen werden alle Portale identifiziert, dass heißt transparente Bereiche von Grenzlinien, wie Fenster oder Türöffnungen durch die ein Betrachter in eine andere Zellen schauen kann (Abbildung 2.3). Die entsprechenden Kinderzellen werden zusammen mit einer Kennung zur Zelle, in die das Portal führt, gespeichert. Eine Kennzeichnung der

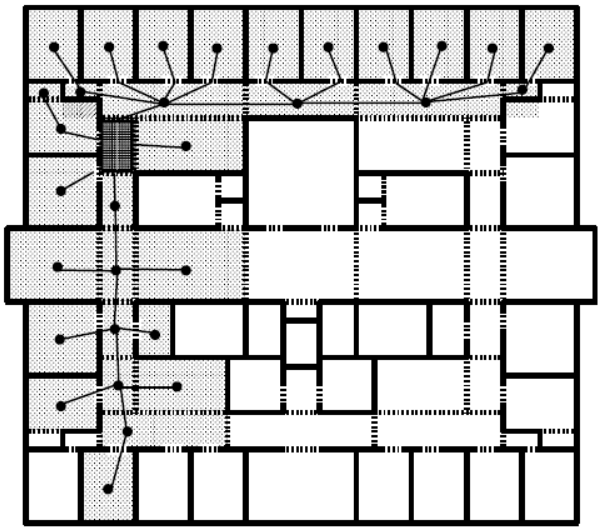

Abbildung 2.4: Zelle-zu-Zelle Sichtbarkeit und der entsprechende Sichtbarkeitsbaum. Die aktuelle Zelle mit dem zu berechneten Sichtbarkeitsbaum ist dunkelgrau, alle sichtbaren Zellen sind hellgrau und unsichtbare Zellen sind weiß unterlegt. Entnommen aus [TAFT92].

Portale auf diesem Weg führt zum Aufbau eines Sichtbarkeits- oder Angrenzungsgraphen aller Kinderzellen der Datenstruktur. Zwei Knoten grenzen genau dann an, wenn ein Portal sie miteinander verbindet.

Ist die räumliche Unterteilung der Szene und ihre Verbindungen einmal konstruiert, wird eine Zelle-zu-Zelle Sichtbarkeit für jede Kindzelle berechnet und in der Datenbank gespeichert. Dabei werden sämtliche Zellen berücksichtigt, die aus allen Richtungen jeder Position eines bestimmten Knotens sichtbar sind. Der Datenbankeintrag der Zelle-zu-Zelle Sichtbarkeit für eine bestimmte Zelle C enthält exakt die Zellen, zu der eine ungehinderte Sichtlinie von C führt (Abbildung 2.3). Solch eine Sichtlinie darf kein opakes Element schneiden und muss ein Portal durchdringen, um von einer Zelle in die nächste zu gelangen. Bei Zellen, die keine direkten Nachbarn sind, aber trotzdem durch eine Sichtlinie in Verbindung stehen, hat diese eine Sequenz an Portalen durchstoßen. Bei der Berechnung der Zelle-zu-Zelle Sichtbarkeit entsteht so automatisch ein Sichtbarkeitsbaum für jede Zelle C der Raumunterteilung, wie Abbildung 2.4 zeigt. Jeder Knoten des Sichtbarkeitsbaumes zeigt auf eine Zelle, die sichtbar von C ist, und jede Kante entspricht einem durchdrungenen Portal.

Nachdem die Sichtbarkeit von Zellen festgelegt ist, wird diese zusätzlich für einzelne Objekte, beispielsweise Tische und Stühle, innerhalb einer Zelle berechnet (Zelle-zu-Objekt Sichtbarkeit) und ebenfalls in der Datenbank gespeichert. Nicht alle Modelle einer Raumunterteilung sind zwingend vom Betrachter sichtbar. Für Nachbarzellen, bei denen der Sichtpunkt direkt in die Portalebene platziert werden kann, ist dies allerdings der Fall. Weiter entfernte Zellen vom Betrachter sind generell nur teilweise sichtbar und umso mehr Portale

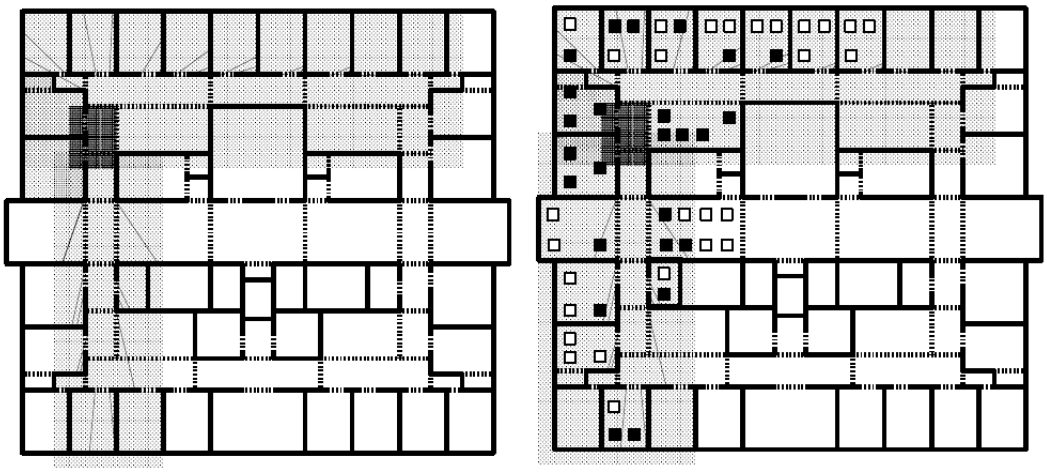

Abbildung 2.5: Zelle-zu-Objekt Sichtbarkeit: Links sind die Frustums in den sichtbaren Zellen zu erkennen und rechts alle Objekte, die sich in den Zellen befinden. Schwarze Quadrate sind vom Betrachter sichtbar und weiße nicht. Entnommen aus [TAFT92].

eine Sichtlinie durchkreuzen muss, desto kleiner ist der Ausblick. Dieser Ausblick in eine Zelle bildet die Geometrie eines Frustums und alle Objekte, die das Frustum schneiden oder komplett darin liegen, sind sichtbar. Die Zelle-zu-Objekt Sichtbarkeit gibt also bei einer gegebenen Zelle C für jede sichtbare Zelle R ein Set an Objekten in R und sichtbar von C zurück. Abbildung 2.5 veranschaulicht diesen Prozess in einem zweidimensionalen Gebäudeplan.

Die Ergebnisse der Vorbearbeitungsphase sind in der Datenbank gelagert. Dazu gehören alle Zellen, Portale und Objekte sowie ihre gegenseitigen Abhängigkeiten bezüglich der Sichtbarkeit. Befindet sich der Betrachter zur Laufzeit des Programms in einer bestimmten Zelle können hierdurch schnell ihre Verbindungen gefunden und relevante Objekte geladen werden. Da das komplette Modell nicht in den sekundären Speicher passt, kann nur eine Untermenge an Objekten geladen werden. Als Minimum werden die Objekte gewählt, die im nächsten Frame zu rendern sind. Um diese zu finden folgt man einfach dem Sichtbarkeitsbaum der aktuellen Zelle. Durch die Referenzen in den Datenbankeinträgen hangelt man sich automatisch an den Kanten und Knoten des Baums entlang. Allerdings dauert es eine gewisse Zeit, Daten von der Festplatte in den Arbeitsspeicher zu transportieren, weshalb auch zukünftig sichtbare Modelle geladen werden sollten. Ansonsten käme es aufgrund der Ladezeiten zu Verzögerungen. Man findet die nächsten Objekte durch das erneute Durchlaufen des Baums. Dabei wird für jede traversierte Zelle die minimale Zeit berechnet, die der Betrachter braucht, um in diesen Raum zu gelangen. Legt man vorher ein Zeit fest, in der die Untermenge an Objekten in einer Zelle geladen werden soll, werden bei der Speicherverwaltung auch zukünftig sichtbare Modelle berücksichtigt. Dieser Prozess muss nicht in jedem Frame laufen, sondern wird nur dann gestartet, wenn der Betrachter über eine Zellenbegrenzung läuft.

2.3 Auswahlkriterien von LODs

Während sich das Szenenmanagement um die optimale Speicherauslastung kümmert, konzentrieren sich „Level of Detail“ Systeme auf die Beschleunigung des Rendervorgangs, indem sie komplexere Modelle durch simplere austauschen, sobald sie weniger relevant sind. Bevor ein Austausch passiert, muss jedoch erst entschieden werden, ob ein Objekt wichtig ist oder nicht. Diese Relevanz wird anhand von Auswahlkriterien bestimmt, die sich in verschiedenen Systemen unterscheiden [TAMH08, DLH03].

Eine distanzbasierte Selektion ist wahrscheinlich der gebräuchlichste Weg ein LOD zu organisieren. Jeder LOD Stufe des Objekts wird eine bestimmte Distanz zugewiesen, die sich relativ auf die Kameraposition bezieht. Die detaillierteste Version hat einen Bereich von null bis zu einem benutzerdefinierten Wert von r_1, was bedeutet, dass dieses LOD angezeigt wird, wenn die Distanz von der Kamera bis zum Objekt kleiner als r_1 ist. Dem nächsten LOD wird ein Bereich von r_1 bis r_2 zugewiesen, wobei $r_2 > r_1$. Ist die Distanz zum Objekt größer oder gleich r_1 und kleiner als r_2, wird dieses LOD genutzt und so weiter. Die Idee dahinter ist einfach: Umso weiter ein Objekt vom Betrachter entfernt ist, desto ungenauer braucht es aufgelöst zu werden. Ein subtileres Thema ist die Frage, von welchem Punkt des Modells aus die Distanz zur Kamera gemessen wird. Der Mittelpunkt ist nicht immer die beste Wahl, denn bei ungleichmäßigen Modellen, zum Beispiel ein Lastwagen, weicht dieser sehr vom naheliegendsten Punkt ab. Diese Eigenschaft hat einen entscheidenden Nachteil. Bewegt man sich in die Nähe des Lastwagens, wird die falsche Entfernung gemessen und der LOD Austausch zu spät oder zu früh ausgeführt, was der Benutzer natürlich visuell mitbekommt und der Effekt als „Popping“ bezeichnet wird [DLH03]. Will man sicher gehen, muss also zuerst der Punkt im Modell berechnet werden, der am nähsten zum Betrachter liegt. Trotzdem ist die distanzbasierte Selektion eine einfach und effiziente Methode um ein LOD zu implementieren, da pro Objekt nur eine aufwändige Rechnung zur Ermittlung der dreidimensionalen euklidischen Distanz zweier Punkte gebraucht wird.

Alternativ zur Distanz kann die LOD Stufe anhand der Größe gewählt werden, bzw. der Größe des Abbildes eines Objekts auf dem Bildschirm. Denn umso weiter weg sich ein Modell befindet, desto weniger Platz nimmt dessen Projektion im Bild ein. Anstatt den verschiedenen Distanzen bestimmte Werte zuzuweisen, legt man nun die Größe oder die Anzahl an Pixel im Bild für jede LOD Stufe fest. Die größenbasierte Auswahl vermeidet einige Probleme der distanzbasierten Selektion, da die Projektionsfläche unabhängig von der Auflösung des Bildschirms oder den Proportionen eines Modells ist. Mit dieser Methode wird immer das ganze Objekt betrachtet und nicht nur ein beliebiger Punkt in der Geometrie. Dieser Vorteil beansprucht allerdings einen aufwändigeren Prozess, da das Objekt zuerst in Bildschirmkoordinaten umgerechnet werden muss. Hierbei wird nicht das originale Modell genommen, sondern ein vereinfachtes Bounding Volumen, meist eine Box oder Kugel, um Rechenzeit zu sparen. Bei rotierenden, dünnen Objekten, wie ein Messer, kann sich die Größe sehr schnell verändern, was einen unerwünschten und schnellen Austausch mehrerer LOD Stufen zur Folge hat, weshalb gerne eine Bounding Kugel genommen wird, deren Projektionsfläche aus einer Distanz immer gleich aussieht.

In vielen Umgebungen sind manche Objekte besonders wichtig, um die Immersion der Szene und die Akzeptanz des Benutzers aufrecht zu erhalten. Zum Beispiel tragen in architektonischen Rundgängen die Wände des Gebäudes mehr zur Wahrnehmung des Benutzer bei, als ein kleiner Stift auf dem Schreibtisch. Somit stellt die prioritätbasierte Auswahl eine dritte Möglichkeit dar, LOD Stufen zu bestimmen. Die Prioritätswerte vergibt man vorzeitig, sodass zur Laufzeit wichtige Elemente gar nicht und nebensächliche je nach Priorität vereinfacht werden. Damit lässt sich diese Methode auch mit anderen LOD Systemen verbinden.

Ein andauernder, unnötiger Wechsel zwischen zwei LOD Stufen kann auftreten, wenn der Betrachter sich stets am zugewiesenen Grenzwert aufhält. Findet beispielsweise ein Austausch der detaillierteren Stufe 0 zur groberen Stufe 1 bei einer Distanz von 100 Metern statt und die Bewegung des Benutzers verläuft eine Zeit lang auf dieser Grenze, werden die zwei LODs oft vertauscht, was eventuell einen unschönen, sichtbaren Effekt zur Folge hat. Um diese Erscheinung zu vermeiden, gibt es die Histeresismethode, die den vergangenen Pfad des Betrachters mit einbezieht. Der im Beispiel verwendete Wert 100 für das Über- und Unterschreiten der Grenze wird dann in einen Bereich von etwa 90 bis 110 erweitert. Überschreitet der Benutzer nun die 110 Meter, so wird die grobere LOD Stufe 1 angezeigt. Der Wechsel zum feineren Modell passiert jedoch erst wieder, wenn die Distanz von 90 Metern unterschritten sind.

Anhand der erläuterten Auswahlkriterien von LOD Stufen kann der Austausch zweier Modelle ablaufen. Dabei ist es generell möglich, dass jedes Kriterium für jede Austauschmethode einsetzbar ist, weshalb die beiden Themen auch getrennt voneinander behandelt werden.

2.4 Diskretes Level of Detail

Der originale Entwurf, von James Clark [Cla76] 1976 veröffentlicht, wird bis in die heutige Zeit ohne Modifikation in den meisten 3D-Grafikapplikationen verwendet [DLH03]. Die Idee dahinter ist mehrere Versionen eines Objekts mit jeweils unterschiedlichem Detailgrad in einem Vorprozess zu modellieren. Zur Laufzeit wird dann ein geeignetes LOD gewählt, welches das Objekt repräsentiert. Der Austausch erfolgt sofort und das neue Modell wird an den Renderprozess geschickt. So kann in jedem Frame eine Selektion durchgeführt sowie das nächste LOD angezeigt werden. Durch die gröbere Auflösung und entsprechend weniger Primitiven in niedrigeren LOD Stufen wird der Rendervorgang beschleunigt ohne die Qualität des Objekts zu verschlechtern, da grobe Modelle erst ab einer weiteren Entfernung angezeigt werden. Abbildung 2.6 zeigt ein simples Beispiel einer Kugel in unterschiedlichen LOD Stufen und den eigentlichen Abstand vom Betrachter.

Alle Modelle erstellt man während einer Vorarbeitsphase, weshalb man nicht vorhersagen kann aus welcher Richtung das Objekt später zu sehen ist. Die Vereinfachung der Modelle reduziert deswegen die Details gleichmäßig über das ganze Objekt, was auch als sichtunabhängiges LOD gilt. Ein diskretes LOD hat einige Vorteile. Die Entkopplung von

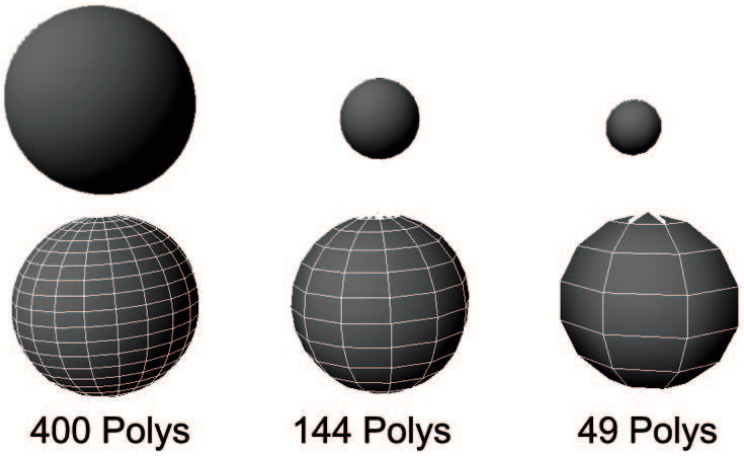

Abbildung 2.6: Ein diskretes LOD einer Kugel in drei Detailstufen mit entsprechenden Polygonzahlen und den eigentlichen Entfernungen (oben).

Modellvereinfachung und Rendering macht es zum einfachsten implementierbaren System. Der Simplifizerungsalgorithmus kann so viel Zeit verbrauchen wie nötig und der Laufzeitprozess braucht nur noch zu entscheiden, welcher Detailgrad in einem Frame angezeigt wird. Um eine möglichst perfekte Annäherung an das Original zu haben und gleichzeitig unnötige Primitive wegzulassen, werden die Modelle meistens per Hand angefertigt. Außerdem ist diese Methode gut für neue Grafikkarten geeignet, denn die separaten, statischen Meshes (dreidimensionale Geometriebeschreibungen durch Primitive) können im Grafikkartenspeicher abgelegt und wiederbenutzt werden. Die vorgefertigten Geometriebeschreibungen können ebenfalls durch Optimierungen, wie Dreiecksbänder und Dreiecksfächer [FE96], das Rendering beschleunigen. Einen entscheidenden Nachteil zieht das diskrete LOD System dennoch mit sich: das Popping, also der harte Übergang zwischen zwei Detailstufen, der sich hier innerhalb eines Frames vollzieht. Die Grenzen für den Übergang sollte man sehr vorsichtig wählen, sonst bekommt der Betrachter diesen Effekt mit.

Das visuelle Vermischen zweier Modelle über einen kurzen Zeitraum, auch Blending genannt [TAMH08], sorgt für einen gleichmäßigeren Übergang. Andererseits sind die Kosten zwei Geometrien zu berechnen höher als für eins, was den Absichten von LOD Systemen widerspricht. Wird die Übergangszeit sehr kurz gehalten, verlaufen nicht viele Blendings zeitgleich und die Kosten sind die Verbesserung wert[TAMH08]. Das Hauptproblem ist hierbei, wie die beiden LODs miteinander vermischt werden, denn beide Stufen semitransparent anzuzeigen resultiert auch in einem semitransparenten Objekt. Giegl und Wimmer [MG07] entwickelten eine Blendingmethode, die gut funktioniert und einfach zu implementieren ist. Hierbei wird das aktuelle Modell solange opak gerendert, bis die nächste LOD Stufe vom anfänglichen Alphawert 0 den Endwert 1 erreicht hat. Ab dann verhält es sich umgekehrt und die neue LOD Stufe wird opak gerendert, bis das alte Modell durch den Alphawert 0 vollkommen ausgeblendet ist.

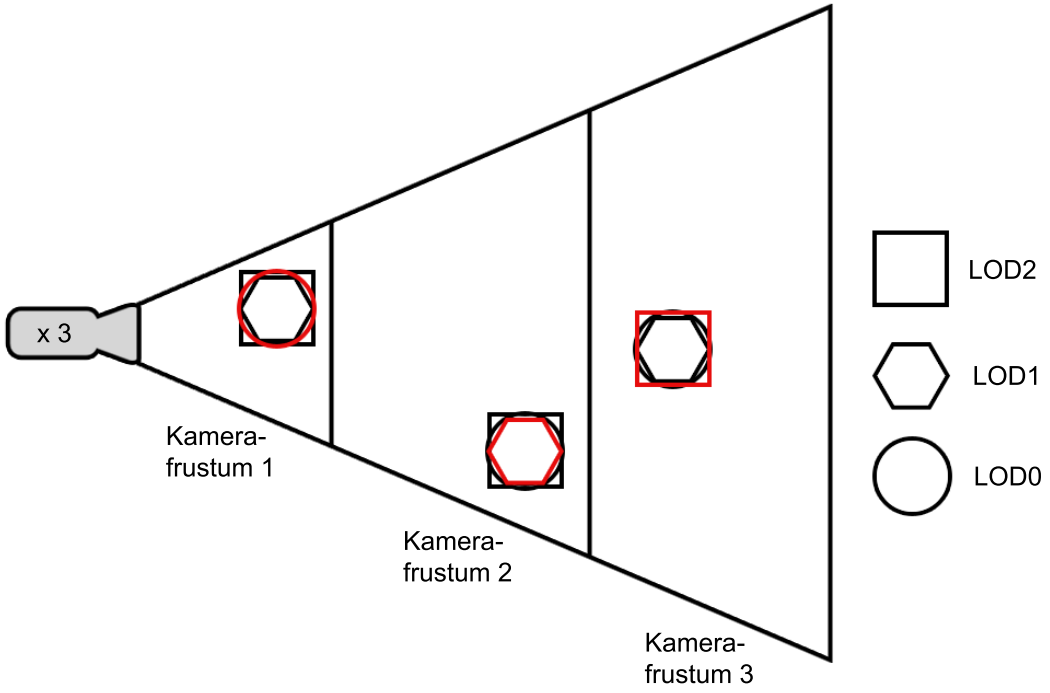

Abbildung 2.7: Implementation eines diskreten LOD Systems: Szene mit 3 Kugeln in unterschiedlichen Entfernungen. Alle gerenderten Modelle sind in rot markiert.

Eine einfache plus schnelle Implementationsmöglichkeit für ein diskretes LOD, welches die distanzbasierte Selektion benutzt, ist das Arbeiten mit mehreren Kameras und entsprechenden Rendermasken. Bei beispielsweise drei LOD Stufen werden drei Kameras und drei Masken angelegt. Die Kameras müssen dem Benutzer vorgeben nur eine Kamera zu sein. Dies soll heißen, dass jede Kamera nur einen Teilbereich des Bildes berechnet, der abhängig von der Objektentfernung ist. Ergo muss die Far Clip Plane der ersten Kamera auf der Near Clip Plane der zweiten liegen und deren Far Clip Plane auf der Near Clip Plane der dritten. Near und Far Clip Plane begrenzen das Kamerafrustum, in dem Objekte gerendert werden, wobei ersteres die Startebene und letzteres die Schlussebene darstellt. Somit wird ein Bild erst durch alle drei Kameras vollständig berechnet. Außerdem wird ihnen jeweils eine Maske zugewiesen, damit die Kamera für den nahen Bereich nur Modelle mit hoher Auflösung rendert, die Kamera für mittelweite Distanzen die mittlere LOD Stufe berechnet und so weiter. Zuletzt werden alle Objekte mit allen LOD Stufen in der Szene platziert, sodass immer drei Modelle eines Objekts auf dessen Position stehen. Den hochdetaillierten Modellen wird folglich die gleiche Maske zugeordnet, wie der Kamera für den nahen Bereich und den zwei anderen Detailgraden die jeweiligen anderen Masken (siehe Abbildung 2.7). Der Nachteil dieser Implementation ist deutlich erkennbar, denn alle Detailstufen müssen schon im Voraus in der Szene sein, was einen enormen Speicherumfang ausmacht.

2.5 Kontinuierliches Level of Detail

Das kontinuierliche Level of Detail System (CLOD) [FRG04, TAMH08, DLH03] weicht vom herkömmlichen Ansatz des diskreten ab, um die Nachteile des Poppings und den höheren Speicherverbrauch zu vermeiden. Anstatt verschiedene LODs während eines Vorprozesses zu generieren, stellt das System eine Datenstruktur zur Verfügung, die ein kontinuierliches Spektrum an Detailstufen enthält. Aus dieser Struktur wird dann zur Laufzeit der gewünschte Detailgrad entnommen. Der Hauptvorteil hierbei ist die bessere Granularität, also der Verlauf an Übergängen zwischen den Schritten. Die Detailstufen müssen nicht wie beim diskreten Verfahren im Voraus feststehen und sind deswegen auch nicht beschränkt. So werden aus drei etwas größeren Modellsprüngen zehn kleinere, die auch das Auge weniger wahrnimmt. Außerdem wird die Geometrie eines Objekts zu jedem Zeitpunkt genauer beschrieben, als aus drei vorgefertigten Modellen zu wählen. Diese sind nur an genau drei Entfernungen zum Objekt optimal getroffen. Bewegt sich der Betrachter noch ein kleines Stück nach vorne oder hinten, kann schon wieder das nächste minimale Detail weggelassen oder hinzugefügt werden. So werden durch die bessere Granularität der Meshbeschreibung nie mehr Polygone als nötig benutzt, was wiederum eine größere Polygonanzahl für andere Objekte freigibt.

Doch das kontinuierliche LOD zieht auch diverse Nachteile mit sich, denn nicht jedes Modell sieht mit einem CLOD gut aus. Man kann nicht spezifizieren an welchen Stellen Details weggenommen oder hinzugefügt werden. Im Durchschnitt werden die Verfeinerungen gleichmäßig über das ganze Modell vorgenommen. Viele Geometrien besitzen eine Grundstruktur mit zusätzlichen Details. Heutige Algorithmen können jedoch nicht zwischen den beiden Bestandteilen unterscheiden, woran die Grundstruktur eines Objekts leidet. Ein weiterer Nachteil kommt beim CLOD zum Vorschein, wenn mehrere Instanzen des gleichen Objekts in einer Szene sind. Die Instanzen sind an unterschiedlichen Positionen platziert und haben verschiedene Distanzen zum Betrachter, was zur Folge hat, dass sie auch verschiedene Detailgrade haben und damit eine eigene Datenstruktur benötigen. Für jede Instanz eines Objekts muss also eine eigene CLOD Datenstruktur angelegt werden, was zusätzlich Speicher wegnimmt. Zuletzt verbraucht ein Algorithmus zum dynamischen Anpassen einer Geometrie mehr Rechnzeit, als ein simples Austauschen zweier Meshes.

Hoppe [Hop96] war 1996 einer der ersten in diesem Gebiet und nannte seine Datenstrukturen „Progressive Meshes“. Heutzutage basieren alle CLODs auf seinen Ideen der selektiven Anpassung von polygonalen Modellen. Hierbei wird ein initiales Mesh $\hat{M}$ durch eine simplere Repräsentation angenähert. Im Verlauf der Anwendung greift die Meshoptimierung immer wieder auf drei grundlegende Transformationen zurück: Kantenzusammenfall (Edge Collapse), Vertexspaltung (Vertex Split) und Kantentausch (Edge Swap). Beim Edge Collapse wird das Mesh vereinfacht, indem die beiden Vertexpunkte v_s und v_t einer Kante $\{v_s, v_t\}$ in der Transformation $ecol(\{v_s, v_t\})$ zu einem einzelnen Vertexpunkt v_s zusammenfallen (Abbildung 2.8). Dadurch verschwinden die benachbarten Dreiecke $\{v_s, v_t, v_l\}$ sowie $\{v_t, v_s, v_r\}$,

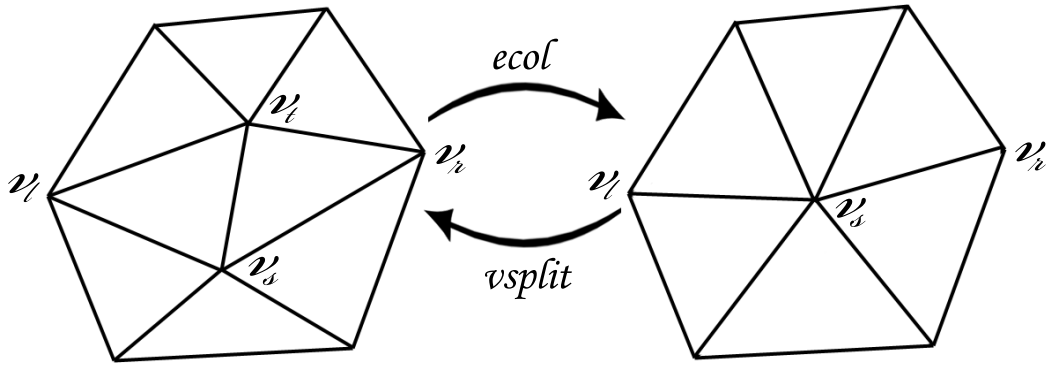

Abbildung 2.8: Edge Collapse und Vertex Split.

auch „Faces“ genannt, und die Geometrie verliert an Detail. So kann ein initiales Mesh $\hat{M} = M^n$ durch n erfolgreiche Edge Collapse Transformationen in ein gröberes Mesh M^0 vereinfacht werden:

$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} ... \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0.$$

Die Sequenz an Edge Collapse Transformationen muss vorsichtig gewählt werden, da sie in direkter Verbindung zur Qualität des Meshes steht. Die Simplifikation soll zu einem anderen Zeitpunkt, zum Beispiel bei der Annäherung des Objekts zum Betrachter, wieder rückgängig gemacht werden. Das heißt die vorigen Schritte arbeitet man in umgekehrter Reihenfolge mit der Vertex Split Transformation ab. Hierbei wird ein Vertexpunkt v_s in zwei gespalten, wodurch eine neue Kante $\{v_s, v_t\}$ entsteht, und alle Kanten mit benachbarten Vertexpunkten von v_s aktualisiert werden (Abbildung 2.8). Ausgehend von der letzten Detailstufe M^0 kann so in n Vertex Split Transformationen wieder das initiale Mesh $\hat{M}$ erreicht werden:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} ... \xrightarrow{vsplit_{n-1}} (M^n = \hat{M}).$$

Diese Transformationen haben eine weitere positive Eigenschaft, die zur noch besseren Kontinuität beiträgt. Nicht nur der gesamte Verlauf an Optimierungen ist feiner unterteilt, sondern es lässt sich auch ein flüssiger Übergang von einem Mesh M^i zum nächsten M^{i+1} realisieren, was Geomorphing genannt wird [Hop96, TAMH08]. Hierzu wird ein Geomorph $M^G(\alpha)$ mit dem Blendparamter $0 \leq \alpha \leq 1$ angelegt, sodass $M^G(0)$ wie M^i und $M^G(1)$ wie M^{i+1} aussieht. Der Geomorph $M^G(\alpha)$ interpoliert dann die Position jedes neuen Vertexpunkts v in M^i linear zur Endposition in M^{i+1}:

$$v^G(\alpha) = (\alpha)v^{i+1} + (1 - \alpha)v^i$$

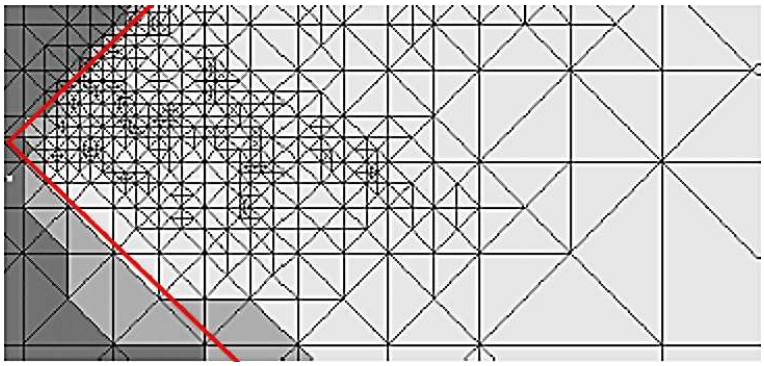

Abbildung 2.9: Sichtabhängiges Level of Detail bei einem Gelände. Das Kamerafrustum ist in rot dargestellt. Man kann deutlich erkennen, dass die Triangulation am Anfang des Frustums feiner ist, als in der Ferne.

2.6 Sichtabhängiges Level of Detail

Das sichtabhängige LOD erweitert das kontinuierliche, indem es sichtabhängige Vereinfachungskriterien benutzt, um dynamisch den am besten geeigneten Detailgrad vom aktuellen Betrachterstandpunkt zu wählen. Die Vor- und Nachteile bleiben fast die gleichen, wie sie in Abschnitt 2.5 beschrieben werden und nur die Art und Weise, wo man Optimierungen vornimmt, ändert sich. Diese sind nämlich abhängig von der Distanz und Richtung des Objekts zur Kamera, weshalb man auch von einem anisotropen LOD spricht [DLH03]. Dadurch werden etwa nahe Bereiche und die Umrissen eines Objekts detaillierter dargestellt als entfernte und innere, beziehungsweise verdeckte Bereiche (Abbildung 2.9). Dies führt zu einer noch besseren Granularität. Polygone werden dort platziert, wo man sie am meisten innerhalb eines Objekts braucht. Man erreicht also eine bessere Genauigkeit für eine vorgegebene Anzahl an Polygonen sowie eine optimierte Ausnutzung der Resourcen. Sehr komplexe Modelle, die ebenfalls räumlich große Ausmaße haben, wie ein Gelände, können ohne eine sichtabhängige Technik nicht optimal vereinfacht werden. Auch ein diskretes oder kontinuierliches LOD hilft hier nichts, denn der Betrachterstandpunkt ist einerseits nahe am Boden und andererseits weit entfernt von Bergen im Hintergrund. Andere LODs optimieren ein Modell jedoch nur gleichmäßig, was entweder zu einem hohen Detailgrad mit nicht akzeptablen Frameraten oder zu guten Frameraten mit einer schrecklichen Genauigkeit führt.

Haupteinsatzgebiet des sichtabhängigen LODs ist die Echtzeitdarstellung von Gelände in dem sich der Benutzer frei bewegen kann. Eine sehr beliebte Implementation in interaktiven, virtuellen Welten ist das ROAM-Verfahren (Real-time Optimally Adapting Meshes) [MD97]. Es basiert ähnlich wie bei Hoppe [Hop96] auf Operationen, die ein Dreieck spalten oder zwei zusammenfügen (Split- und Mergeoperationen), und dadurch das Gelände in jedem Frame an die derzeitigen Verhältnisse anpassen. Dafür benutzt der Algorithmus zwei Queues zur Ausführung der Operationen. Der erste Queue enthält eine nach Prioritäten geordnete Liste an Dreiecksplits, sodass die Verfeinerung des Geländes nur das wiederholte Teilen des

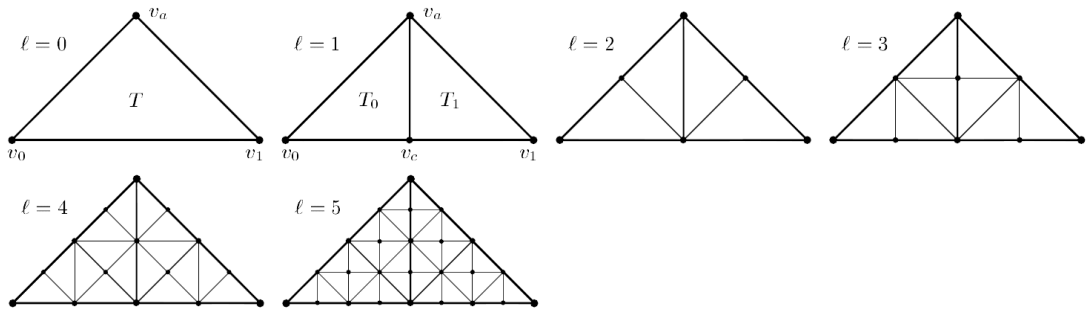

Abbildung 2.10: Rekursiver Aufbau des binären Baums: Level 0 - 5. Entnommen aus [MD97].

Dreiecks mit der höchsten Priorität bedeutet. Der zweite Queue enthält eine weitere Liste, welche alle Mergeoperationen nach ihren Prioritäten ordnet und das Gelände vereinfacht. Damit ist sogar der Zusammenhang zwischen Frames gewährleistet, indem man die Triangulation, also der Aufbau der Dreiecke, vom letzten Frame übernimmt und nur noch für den aktuellen Frame anpasst.

Für das Mesh der Landschaft wird ein reguläres Netz benutzt, bei dem der Abstand von zwei Höhenwerten immer der gleiche ist [Kra05]. Wie hoch ein Punkt auf dem Gebiet ist, definiert eine Höhenkarte in Graustufen. Dabei legt man für den Weißwert eine bestimmte maximale Höhe und den Schwarzwert eine minimale Höhe fest und für jede Graustufe dazwischen wird die Höhe linear interpoliert. Bei einer Höhenkarte von 512 auf 512 Pixeln erhält das Mesh ebenfalls quadratische Ausmaße von jeweils 512 Vertexpunkten pro Seite, was eine Gesamtanzahl von 262.144 Vertexpunkten ergibt. Dieses dient dann als Grundlage zur Erstellung eines binären Baums (2.1), womit es bei n Unterteilungen das feinste Level $l = n$ repräsentiert. Durch den binären Baum können die Operationen später schneller ausgeführt werden. Er wird wie folgt aufgebaut: Das Wurzeldreieck $T = (v_a, v_0, v_1)$ wird als ein rechtwinkliges, gleichschenkliges Dreieck im gröbsten Level $l = 0$ der Unterteilungen definiert. Im nächst feineren Level $l = 1$ werden die beiden Kindknoten festgelegt, indem man die Wurzel entlang der Mittelkante teilt. Die Mittelkante wird durch den Vertexpunkt im Scheitel v_a und den Mittelpunkt v_c der Hypothenuse (v_0, v_1) beschrieben. Der linke Kindknoten von T ist dann $T_0 = (v_c, v_a, v_0)$ und der rechte $T_1 = (v_c, v_1, v_a)$. Der Rest des Baums wird durch rekursives Wiederholen der Splitoperation aufgebaut. Abbildung 2.10 veranschaulicht den Prozess.

Auf diese Weise kann ein Dreieck unabhängig von anderen durch Split- und Mergeoperationen auf den gewünschten Detailgrad gebracht werden. Bei einer Splitoperation wird das aktuelle Dreieck durch seine beiden Kindknoten in der Triangulation ersetzt und erhält dadurch eine feinere Geometrie. Im Gegensatz dazu steht die Mergeoperation. Sie ersetzt zwei Kindknoten durch ihren Elternknoten, wodurch ein gröberer Aufbau entsteht. Auf welche Dreiecke nun die Verfahren angewandt werden, entscheiden die Prioritäten in dem jeweili-

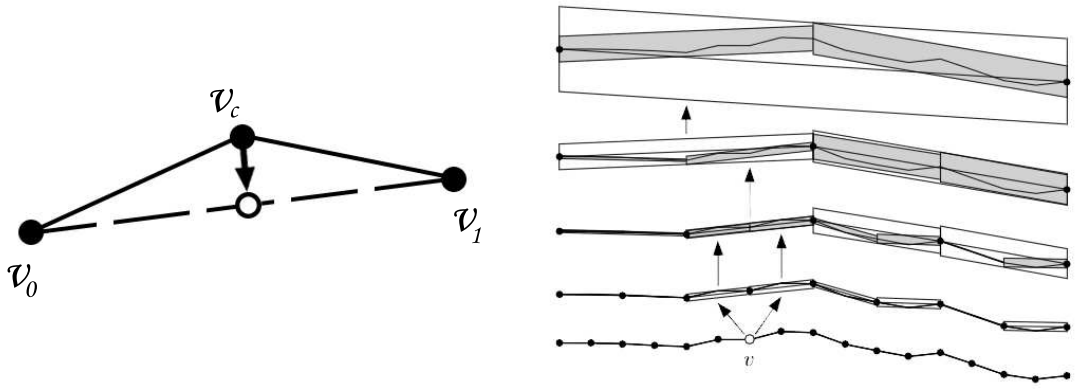

Abbildung 2.11: Auf der linken Seite verschwindet der Vertexpunkt v_c, wodurch ein Sprung in der Geometrie entsteht. Auf der rechten Seite werden die Bounding Volumen so gebildet, dass sie alle Kindknoten umfassen. Entnommen aus [MD97].

gen Queue. Die Prioritäten legt ein Verfahren fest, das den Wert eines metrischen Fehlers berechnet. Dieser Fehler gibt die Distanz des Sprungs an, der beim Vereinfachen des Meshes entsteht. Dabei werden nämlich Vertexpunkte weggelassen, was einen Sprung in der Geometrie zur Folge hat (Abbildung 2.11). Die Distanz zwischen dem originalen Vertexpunkt und der angenäherten Position entspricht dann dem Fehler. Um die Sichtabhängigkeit zu gewährleisten, misst man diesen in der Bildebene.

Dazu wird für jedes sichtbare Dreieck ein Bounding Volumen in die Bildebene projeziert. Alle BV erzeugt man in einem Vorprozess für die einzelnen Dreiecke in sämtlichen Leveln. Hierbei wird das Dreieck durch eine festgelegte Dicke d zu einer Keilform extrudiert. Auf dem feinsten Level l_n ist $d = 0$ und für jedes niedrigere Level wird d so festgelegt, dass das BV alle Kindknoten umfasst (Abbildung 2.11). Dadurch hat das BV automatisch die Dicke des entstehenden Sprungfehlers. Die Bounding Volumen werden in die Bildebene projeziert und dort die maximale Länge der Abbildung ermittelt. Somit steht der Wert des Fehler und die Priorität für die aktuelle Situtation fest. Bei einer großen Abbildung ist die maximale Länge und damit die Priorität höher. Folglich steht die dazugehörige Splitoperation entsprechend weit am Anfang des Queues und das Dreieck wird im nächsten Durchlauf verfeinert. Dessen Kindknoten haben ein kleineres BV und werden demnach eine niedrigere Priorität bekommen. In jedem Frame werden die Prioritäten neu berechnet und das Mesh durch so wenig wie möglich Split- und Mergeoperationen an den aktuellen Betrachterstandpunkt angepasst.

2.7 Nebel

In vielen LOD Systemen hilft das Hinzufügen von Nebel im Renderprozess die verschiedenen Detailgrade noch weniger auffallen zu lassen [TAMH08]. Üblicherweise benutzt man ihn vor

allem in Außenszenen, um der Umgebung einen simplen atmosphärischen Effekt zu verleihen [FK03] und ein realistischeres Ergebnis zu erreichen. Der Nebel dient dem Benutzer aber auch zur besseren Tiefeneinschätzung von Objekten. Ohne diesen Effekt sehen Computergrafikszenen meistens abnormal scharf und klar aus, denn auch in der Wirklichkeit ist man ständig von Nebelarten umgeben, die dem Bild einen besonderen Charakterzug geben (Abbildung 2.12). Dabei steht er für alle möglichen Partikel, die in der Luft herumschweben, beispielsweise Wasserdampf, Staub oder Rauch, und dem Betrachter mit steigender Entfernung die Sicht versperren. Aus diesem Grund lässt sich diese Technik optimal mit LODs verbinden, denn der Nebel verschleiert nicht zu sehende Details im Hindergrund. In Kombination mit einem Kamerafrustum, bei dem die Far Clip Plane auf die Weite gesetzt wird, in der Objekte vollständig im Nebel verschwinden, lässt sich dieser sogar zur Beschleunigung des Renderings verwenden (Abbildung 2.12). Alle Objekte, die sich hinter der Far Clip Plane befinden, werden nicht gerendert, weil sie sowieso nur in der Farbe des Nebels erscheinen würden.

In der Realität wird Nebel sichtbar, weil die Lichtstrahlen eines Objekts auf dem Weg zum Auge jede Menge Partikel durchqueren müssen. Hierbei werden manche Strahlen komplett von den Partikel absorbiert oder in sonstige Richtungen gestreut. Andere wiederum durchqueren den Nebel ungehindert oder Strahlen aus verschiedenen Richtungen werden durch den Zusammenprall mit Partikeln zum Auge weitergeleitet. Durch diese Einflüsse vermischt sich allmählich die Farbe des Nebels mit der Farbe des Objekts. In der Computergrafik wird der Nebeleffekt in einem Shader berechnet [FK03]. Einige Attribute spielen dabei eine Rolle:

- Die Farbe des Nebels C_{nebel} wird als konstant angenommen und kann nach eigenen Belieben gewählt werden.
- Die Farbe des Objekts C_{objekt} variiert je nach Objekt und wird bei der Berechnung distanzabhängig mit der Nebelfarbe interpoliert.
- Das Maß an verbliebener Lichtintensität über eine bestimmte Einheitsdistanz ist die Nebeldichte g. Laufen die Lichtstrahlen eines Objekts durch ein Einheitssegment mit festgelegter Distanz, so ist am Ende des Einheitssegments nur noch ein bestimmter Prozentsatz g der ursprünglichen Lichtstrahlen vorhanden. Die restlichen Strahlen wurden vom Nebel absorbiert oder gestreut, wodurch die Lichtintensität für das Objekt abnimmt. Da nicht mehr Licht abhanden kommen kann als vorher in das Segment eingetreten ist, gilt die Bedingungen $0 \leq g \leq 1$. Umso kleiner die Nebeldichte, desto früher verschwinden Objekte im Nebel. Sie bleibt innerhalb einer Szene konstant und kann beliebig gewählt werden.
- Die Anzahl z an traversierten Einheitssegmenten von einem Objekt hin zur Kamera ist variabel und je größer die Distanz ist, desto mehr verhüllt der Nebel ein Objekt.

Mit diesen Attributen lässt sich die Farbe des Lichts am Ende des ersten Segments $C_{segment}$ bestimmen. Dabei wird die Intensität des Objekts und des Nebels wie folgt gemischt:

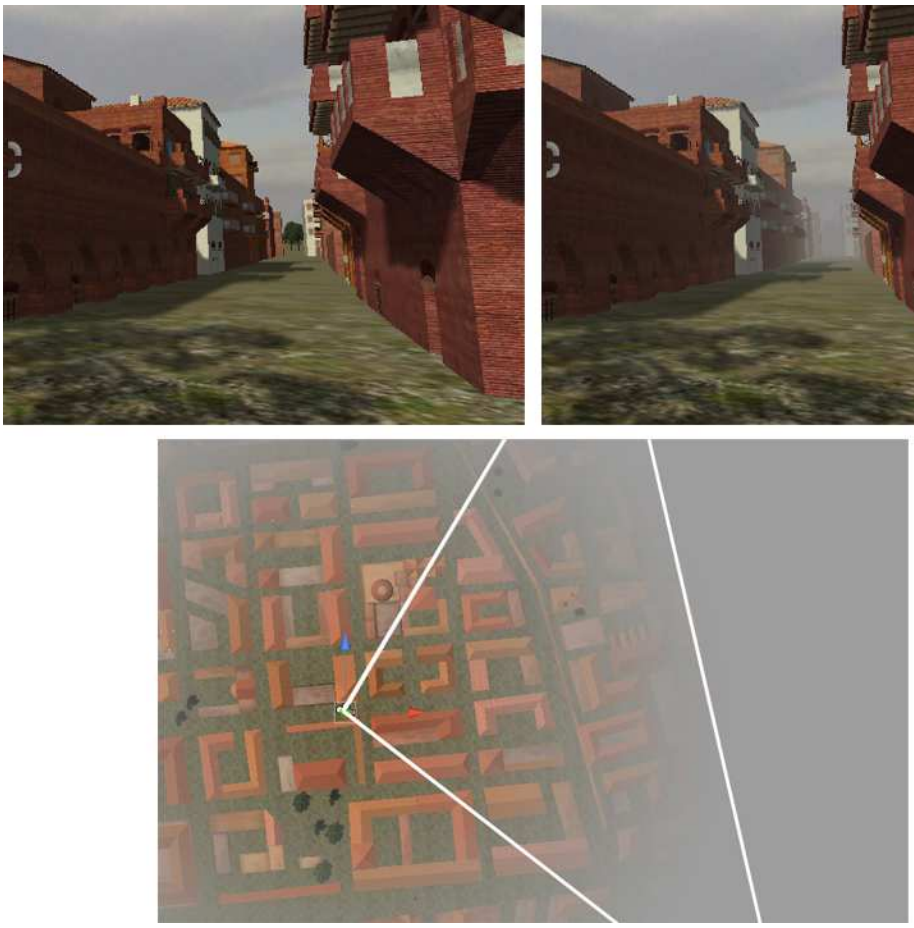

Abbildung 2.12: Links: Szene ohne Nebel. Rechts: Szene mit Nebel. Unten: Szene von Oben mit Nebel und Kamerafrustum.

$$C_{segment} = g \cdot C_{objekt} + (1 - g) \cdot C_{nebel}$$

Anstatt nun das Ergebnis $C_{segment}$ bei längeren Distanzen wieder in die Formeln als C_{objekt} einzusetzen, exponiert man einfach die Dichte g mit der Segmentanzahl z, um die Lichtfarbe C_{auge} für das Auge zu bestimmen:

$$C_{auge} = g^z \cdot C_{objekt} + (1 - g^z) \cdot C_{nebel}$$

2.8 Zusammenfassung

In diesem Kapitel wurde auf die Grundlagen und den Stand der Technik von LOD- und Szenenmanagementverfahren eingegangen. Dabei kommen bei beiden hierarchische Datenstruk-

turen zum Einsatz, die Szenen in kleinere Unterräume teilen, um räumliche Abfragen zu beschleunigen. Eine besonders effektive Methode zur Objektverwaltung in Gebäuderundgängen wurde beschrieben. Sie basiert auf der Tatsache, dass die Sicht in einem Raum durch die angrenzenden Wände sehr beschränkt wird und deshalb in den häufigsten Fällen nur die Objekte des aktuellen Raums und der Nachbarräume geladen werden müssen. Wendet man das Verfahren in der gleichen Weise auf Außenszenen mit Bergen, Tälern und etlichen Gebäuden an, so würde das Resourcenmanagement aufgrund der enorm divergenten Sichtverhältnisse das System überlasten. Steht der Betrachter auf einem Hügel mit einer weiten Rundumsicht, ist es zum Beispiel nicht möglich alle Häuser in Reichtweite zu laden und rendern.

Danach wurden die Auswahlkriterien und Austauschverfahren von LOD-Systemen genauer erläutert. Hierbei wird die simpelste Lösung am häufigsten in interaktiven, virtuellen Welten, wie Computerspielen, implementiert [DLH03]. Als Grundlage dienen dafür verschiedene vorgefertigte statische Modelle eines Objekts, die man je nach Abstand zur Kamera einblendet. Die Vorteile dieser Methode sind die Wiederverwendbarkeit von Objekten, die optimale Anpassung der Details per Hand und die kurze Rechenzeit des Algorithmus. Bei einem zu harten Wechsel der Modelle, kann der Benutzer jedoch ein Poppingeffekt wahrnehmen, was ein Nachteil ist. Diesen beseitigt das kontinuierliche LOD durch die automatische Anpassung der Modellbeschreibung zur Laufzeit. Der Nachteil dabei ist die höhere Rechenlast und das sich ein Objekt im Speicher nicht wiederverwenden lässt. Das sichtabhängige LOD erweitert das kontinuierliche, indem es die fortlaufenden Anpassung am Modell abhängig von der Position und Richtung des Betrachterstandpunkts vornimmt. Das System wird aufgrund der Komplexität in der Praxis nur zur Vereinfachen von Landschaften oder in der Medizin benutzt [DLH03]. Zuletzt geht das Kapitel auf den Nutzeffekt bei der Kombination von LODs mit Nebel ein. Durch den Nebel ensteht nicht bloß eine realistischere Atmosphäre, er verschleiert auch die fehlenden Feinheiten.

Bevor nun die Datenbasis des Stadtmodells optimiert in der Objektdatenbank gespeichert werden kann und mithilfe der erleuterten Techniken ein passendes LOD mit einer Datenverwaltung verknüpft wird, stellt das nächste Kapitel die dafür benötigten Grundlagen dar. Alle Optimierungen sollen entweder den Renderprozess beschleunigen oder den Speicherverbrauch reduzieren. Deswegen geht das folgende Kapitel auf die Renderpipeline ein, die ein Objekt durchlaufen muss, bevor es am Bildschirm angezeigt wird. Zusätzlich wird die Software und ihre speziellen Eigenschaften vorgestellt, mit der das Vorhaben dieser Arbeit umgesetzt werden soll.

Kapitel 3

Echtzeitrendering

Nachdem die Grundlagen sowie der Stand der Technik für LOD und Szenenmanagementsysteme geklärt sind, geht dieses Kapitel auf Echtzeitrendering und die Systeme ein, mit denen man interaktive, virtuelle Welten entwickelt. Die Objekte des zugrundeliegenden Stadtmodells von Rom sollen zuerst speicher- und performanceoptimiert in einer Datenbank gehalten werden. Zur Laufzeit der Anwendung werden dann Objekte geladen und zum Rendern an die Grafikkarte weitergegeben. Dabei durchlaufen sie die sogenannte Renderpipeline [MB05]. Wie sich diese aufbaut und welchen Einfluss das auf die Optimierung der Basisdaten hat, erläutert dieses Kapitel.

Heutzutage wird der Großteil an Echtzeitwelten mithilfe von Game-Engines entwickelt. Sie besitzen einen fundamentalen Funktionsumfang, der für die Erstellung aller interaktiven, virtuellen Welten benötigt wird und auf den Entwickler zurückgreifen können. Die Umsetzung des Rome Reborn Projekts in eine Echtzeitwelt passiert mittels der Game-Engine Unity3D[1], auf die das Kapitel genauer eingeht.

3.1 Renderpipeline und Engpässe

Wie der Name schon sagt besteht der Vorgang zur Berechnung eines Bild aus einer Pipeline. Man kann sie sich wie einen Kanal vorstellen durch den die Daten laufen müssen, damit am Ende das fertige Ergebnis, in diesem Fall ein computergeneriertes Bild, erscheint. Die zu verarbeitenden Daten sind geometrische Informationen über die Objekte, also Vertexpunkte, Kanten und Polygone, sowie Texturen, die auf den Modellen liegen [MB05]. Die Renderpipeline besteht aus verschiedenen Abschnitten, welche die Daten durchlaufen [TAMH08]. Dabei können sie nicht in den nächsten Abschnitt gelangen, bevor die Verarbeitung im aktuellen Arbeitsschritt beendet ist. Zur gleichen Zeit können sich aber weitere Daten in anderen Abschnitten befinden, was einen Geschwindigkeitszuwachs bedeutet und der Hauptgrund zur Verwendung von Pipelines ist. Idealerweise kann ein System mithilfe einer Pipeline mit n Unterteilungen um das n-fache beschleunigt werden. Oft ist das jedoch nicht der Fall, denn das System ist immer nur so schnell, wie der langsamste Arbeitsabschnitt. Nimmt dieser

[1] http://unity3d.com/

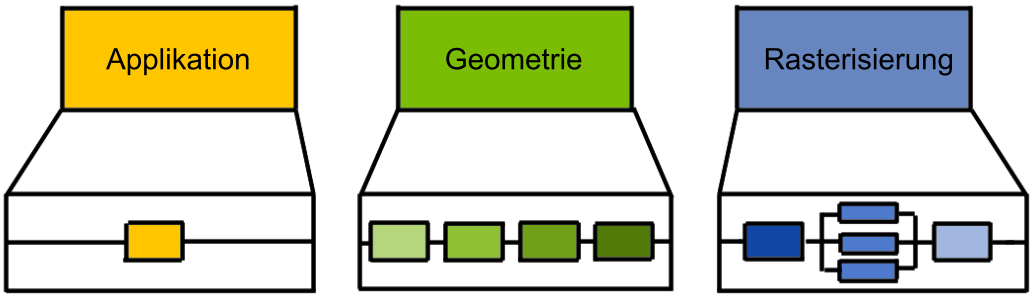

Abbildung 3.1: Die drei Basisstufen der Renderpipeline: Applikation, Geometrie und Rasterisierung. Dabei stellen diese wieder eine Pipeline dar, deren Arbeitsschritte sogar teilweise parallelisiert sind. Entnommen aus [TAMH08].

im Verhältnis zu den anderen Abschnitten bedeutend mehr Zeit in Anspruch, so entsteht schnell ein Flaschenhals im Datenstrom.

In der Renderpipline können an bestimmten Stellen ebenfalls Engpässe vorkommen, die es zu optimieren gilt. Eine grobe Aufteilung der Echtzeitrenderpipeline in die drei Abschnitte Applikation, Geometrie und Rasterisierung zeigt Abbildung 3.1 [TAMH08]. Diese Struktur ist der Kern für Computergrafikanwendungen in Echtzeit, worauf auch die im nächsten Abschnitt beschriebenen Game-Engines aufbauen.

Wie man sieht, befindet sich die erste Stufe des Kanals noch in der Anwendung selbst, das heißt sie ist in der Software implementiert, die auf einem üblichen Computer läuft. Demnach hat der Entwickler vollen Zugriff auf die Programmierung des Systems und kann die Stufe beliebig optimieren, um gleich den ersten Engpass zu vermeiden. Die Nutzung der parallelen Programmierung mithilfe von Threads erlaubt dem Computer eine große Anzahl an Aufgaben effizient zu bewältigen und folglich das System wesentlich zu beschleunigen [TR07, TAMH08]. Derzeitige Computer besitzen häufig mehrere Rechenkerne, die zur parallelen Ausführung unterschiedlicher Aufgaben fähig sind. Auf diese Weise können unabhängige Aufträge zur gleichen Zeit berechnet werden und die Durchlaufzeit für den Pipelineabschnitt verkürzt sich. Bei voneinander abhängigen Aufträgen lässt sich diese Methode nicht anwenden, weil ein Prozess erst auf seinen Vorgänger warten müsste und damit wieder eine Pipeline gebildet wird. Typische Implementierungen im Applikationsabschnitt ist das Aussortieren von nicht sichtbaren Modellen, also die im Kapitel 1 dargestellen Methoden des Back Face-, View Frustum und Occlusion-Cullings. Auch das datenbankbasierte LOD-System für das Stadtmodell wird in dieser Ebene aufgesetzt. Veränderungen in dem Abschnitt können sich auf die Performance in späteren Abschnitten auswirken. Zum Beispiel beschleunigt das Wegfallen von unsichtbaren Geometrien das Zeichnen auf den Bildschirm. Die wichtigste Aufgabe des Applikationabschnitts ist am Ende die zu rendernden Geometrien und Texturen an den Geometrieabschnitt weiterzuleitet.

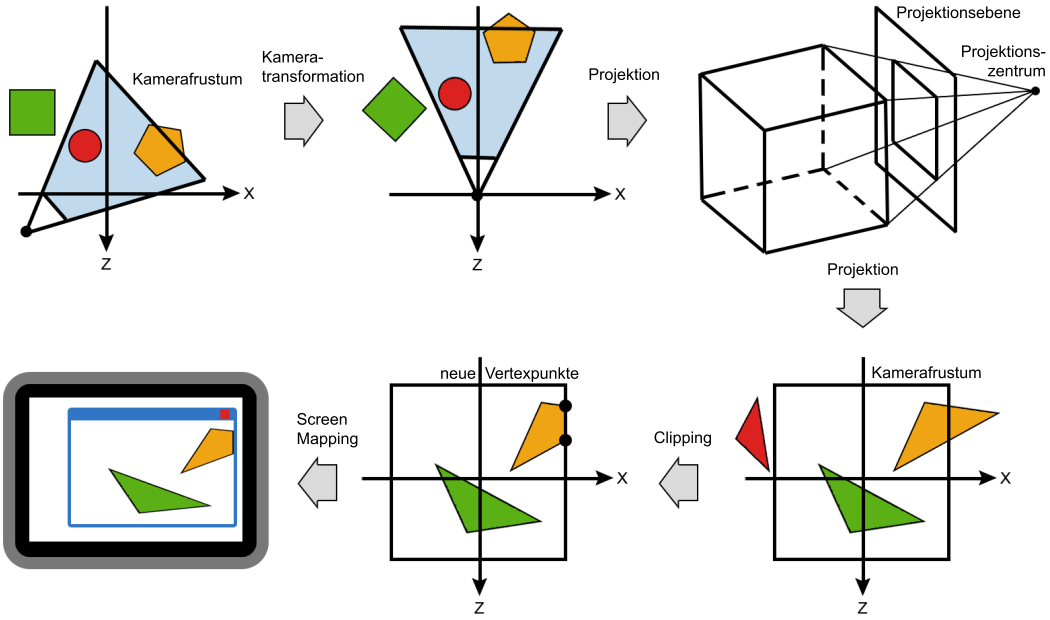

Abbildung 3.2: Kamera-Transformation, Projektion, Clipping, Screen Mapping, Scan Conversion. Angelehnt an [TAMH08].

Ab hier werden die Berechnungen nicht mehr auf dem Hauptprozessor (CPU = Central Processing Unit) des Computers, sondern auf der Grafikkarte (GPU = Graphics Processing Unit) durchgeführt. Die Architektur der Grafikkarte ist auf die Berechnung von Dreiecken konzipiert, weshalb das polygonale Netz eines Meshes nur aus Dreiecken bestehen darf [FK03]. Der Zugriff auf die Funktionen der GPU ist limitiert und auch die Anpassung, bzw. die Programmierbarkeit der verschiedenen Abschnitte ist nur begrenzt möglich [TAMH08]. Verschiedene Programmierschnittstellen (API = Application Programming Interface) geben dem Entwickler Zugang zu den Funktionen der Grafikkarte. Die zwei am häufigsten benutzten Schnittstellen in der Echtzeitcomputergrafik sind OpenGL[2] und Direct3D[3].

Soll nun ein Objekt durch den entsprechenden Funktionsaufruf in der API gerendert werden, absolvieren die Geometriedaten nicht nur einen Abschnittswechsel, sondern werden auch vom Arbeitsspeicher des Computers in den Speicher der Grafikkarte transferiert. Die Datenübertragung dauert eine bestimmte Zeit, weshalb man so wenig Informationen wie nötig freigeben sollte. Bei der schnellsten Übertragungsmethode wird Rücksicht auf die mehrfache Instanziierung eines Objekts in der Szene genommen [TAMH08]. Dafür werden Vertexpufferobjekte (VBO = Vertex Buffer Object) im Speicher der Grafikkarte angelegt,

[2]OpenGL (Open Graphics Library) ist eine plattform- und programmiersprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafik. http://www.opengl.org/

[3]Direct3D ist eine Programmierschnittstelle von Microsoft für 3D-Computergrafik und läuft nur auf eigenen Systemen, wie Windows oder der Xbox. Direct3D ist ein Bestandteil von DirectX. http://msdn.microsoft.com/de-de/directx/

auf die man von der Anwendung aus zugreifen kann [Cor03]. Dadurch muss ein Objekt nur einmal an die GPU geschickt werden, obwohl es mehrfach in der Szene vorkommen kann. Im ersten Abschnitt der Geometriepipeline werden dann die unterschiedlichen Koordinatenräume der Instanzen in das Koordinatensystem der virtuellen Kamera transformiert (Abbildung 3.2). Damit befinden sich alle Objekte im selben Koordinatenraum und verfügen über eine eindeutige Position und Rotation. Darauf folgt das Vertex Shading. In der Computergrafik bestimmt das Shading die äußerliche Erscheinung eines Objekts [FK03]. Dazu gehört das Material sowie die Lichtberechnung. Von einem simplen Erscheinungsbild bis zu aufwändigeren Berechnungen, wie Reflektionen, ist heute vieles möglich. Das Vertex Shading ist dabei für Berechnungen auf Vertexebene zuständig, während sich zu einem späteren Zeitpunkt die Operationen des Pixel Shadings auf einzelne Pixel beziehen. Das Programm, welches all diese Berechnungen auf der Grafikkarte ausführt, nennt man Shader.

Im nächsten Schritt werden die Primitive aller Objekte vom dreidimensionalen in den zweidimensionalen Bildraum der virtuellen Kamera projeziert (Abbildung 3.2). In der Regel benutzt man hier die perspektivische Projektion, bei der Objekte umso kleiner erscheinen, desto weiter sie entfernt sind. Dabei können die Polygone eines Objekts außerhalb und innerhalb des Kamerafrustums liegen. Weil außenliegende Polygone nicht im Bild erscheinen und somit auch für weitere Berechnung irrelevant sind, werden sie durch einen Algorithmus von der restlichen Geometrie abgeschnitten (Abbildung 3.2). Diese Stufe nennt man Clipping [TAMH08]. Im letzten Schritt der Geometriepipeline werden die projezierten Koordinaten der Primitiven an die konkreten Bildschirmkoordinaten, meistens ein Fenster im Betriebssystem, angepasst (Abbildung 3.2), bevor sie an die Rasterisierung weitergegeben werden. Im Verlauf der Verarbeitungen im Geometrieabschnitt hängt die Durchlaufzeit primär von der Anzahl an Vertexpunkten ab, da alle Berechnungen pro Vertexpunkt ausgeführt werden.

Die transformierten und projezierten Vertexpunkte dienen zusammen mit den Daten aus dem Vertex Shading als Input für den Rasterisierungsabschnitt. Dessen Ziel ist es die Farben für die Pixel zu berechnen, die von Objekten abgedeckt werden. Während die Geometriepipeline ihre Berechnungen also pro Dreieck, bzw. pro Vertex ausführt, arbeitet die Rasterisierungspipeline pro Pixel. Im ersten Schritt werden die Differentiale für die Oberfläche der Polygone gebildet, sodass für jeden Pixel ein Wert feststeht. Dazu gehört beispielsweise die Interpolation der Normalen. Darauf werden die Pixel auf Überlappung mit den Dreiecken geprüft (Abbildung 3.2). Nach diesem Schritt steht fest, welches Objekt welche Pixel einnimmt und das Pixel Shading kann angewendet werden, was den endgültigen Farbwert für jedes Pixel berechnet. Im letzten Abschnitt der Renderpipeline werden noch verschiedene Tests pro Pixel durchgeführt. Dazu gehört der Z-buffer-Test. Im Z-buffer der GPU steht der Wert für die Distanz des zu rendernden Objekts zur Kamera für jedes Pixel. Ist die Distanz des neuen Objekts größer, befindet es sich hinter dem bereits gezeichneten Objekt und der Farb- sowie Z-buffer-Wert werden verworfen. Ist die Distanz kleiner, werden Farbe und Z-buffer mit den neuen Werten überschrieben. Auf diese Weise können die Objekte der Szene in beliebiger Reihenfolge gezeichnet werden und sind nicht voneinander abhängig, weshalb viele Berechnungen parallel auf der GPU laufen können. Zuletzt wird das gerenderte Bild mit dem aktuellen ausgetauscht.

Ein allgemeiner Engpass kommt noch mit den Funktionsaufrufen in der API daher. Der Befehl für die Grafikkarte zum Rendern eines Meshes, also ein zusammengefasstes Netz an Dreiecken, heißt in der Echtzeitcomputergrafik auch Draw Call [TAMH08]. Nun ist der Aufruf weniger Draw Calls mit großen Meshes effizienter, als der mehrerer Draw Calls mit kleinen Meshes[4]. Die Ursache ist ein Zuschlag an fixen Kosten, die mit jedem Draw Call unabhängig von der Größe des Meshes verbunden sind. Die Basisidee zur Vermeidung vieler Draw Calls ist das Zusammenfassen von Objekten, sodass innerhalb eines Aufrufs gleich mehrere Meshes die Pipeline durchlaufen. Als Voraussetzung gilt jedoch die Benutzung des selben Shader Programms. Haben die Objekte also unterschiedliche Materialien, lassen sie sich nicht für eine bessere Performance zusammenfassen. Außerdem können kombinierte Objekte später nur noch gemeinsam verändert werden. Sie lassen sich etwa nicht einzeln verschieben oder rotieren. Entwickler oder Grafiker müssen demnach selbst entscheiden, welche Objekte sich performanceoptimiert verknüpfen lassen, ohne dabei ungewollte Abhängigkeiten zu schaffen.

3.2 Unity3D

Bei der Entwicklung von 3D-Echzeitanwendungen, insbesondere der Spieleentwicklung, sind Game-Engines heute kaum mehr wegzudenken [Kra07]. Die Entwicklungszeit und Kosten für eine leistungsstarke und optisch ansprechende 3D-Echtzeitanwendung sind hoch, wenn man sie von Grund auf entwerfen will [SZ04]. Dies ist in den meisten Fällen nicht nötig, weil sich die Basis dieser Anwendungen immer wiederverwenden lässt. Die Implementation eines Systems, welches Objekte performant am Bildschirm anzeigt, funktioniert allgemein bei jeder Anwendung. Aus dem Gedanken der Abstraktion und Wiederverwendung hat sich im Laufe der Softwareentwicklung die Benutzung von Middleware durchgesetzt, zu deren Gruppe auch Game-Engines gehören [Kra07].

Middleware lässt sich als anwendungsunabhängige Technologie bezeichnen, die Funktionen für darauf aufbauende Programme bereitstellt, sodass die Komplexität und Infrastruktur der zugrundeliegenden Technologie verborgen bleibt [WAR01]. Game-Engines nehmen Entwicklern einen großen Teil an Arbeit ab, da man auf bereits programmierten Quellcode zurückgreift. Durch eine modularisierte Softwarearchitektur bieten Game-Engines den Designern die Möglichkeit sich auf den wesentlichen, kreativen Anteil bei Spielen zu konzentrieren, ohne technisches Fachwissen besitzen zu müssen. Für Programmierer gilt das gleiche. Die Engines stellen ihnen Quellcode zur Verfügung, der sich nicht auf die anwendungsspezifische Logik bezieht [LJ02]. Nur das individuelle Verhalten jeder Applikation muss dann noch vom Programmierer spezifiziert werden. Die im vorigen Abschnitt beschriebene Renderpipeline sowie Kollisionsbestimmung, Netzwerkunterstützung, Tonausgabe, Eingabesteuerung und Animationssystem sind beispielsweise Module, die man oft benötigt und deren Funktionsweise konstant bleibt, weshalb sie von der Game-Engine bereitgestellt werden. Die persönlichen Eigenschaften und das Verhalten eines Charakters unterscheiden sich in ver-

[4] Batchtests von NVIDIA: http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.pdf

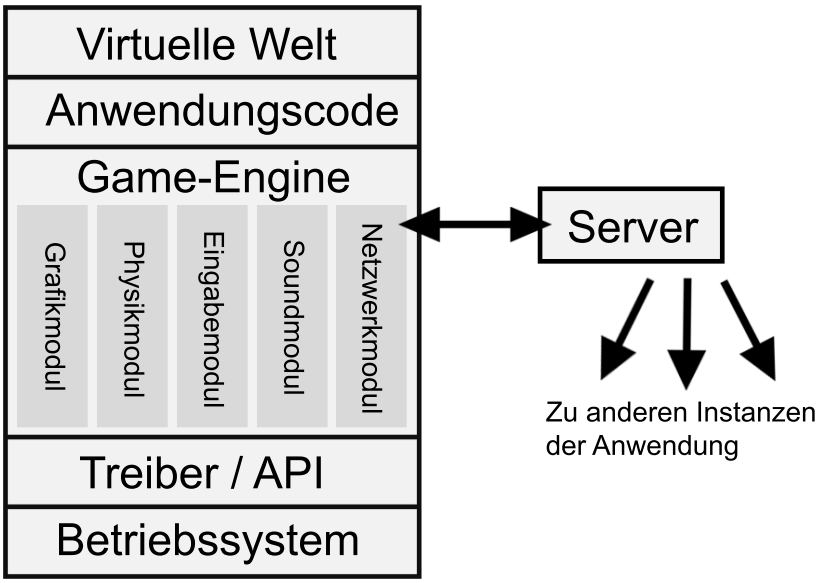

Abbildung 3.3: Schichtenmodell von 3D-Echtzeitanwendungen.

schiedenen Anwendungen und werden demnach vom Programmierer neu festgelegt.

Durch die Verwendung von Middleware und der verschiedenen Module, die wiederum durch Schnittstellen auf andere Systeme zugreifen, lässt sich die Infrastruktur einer 3D-Echtzeitanwendung in einem Schichtenmodell darstellen (Abbildung 3.3) [LJ02, Kra07]. Auf der obersten Ebene liegt die virtuelle Welt, also die aktuelle Szene plus die umfassten Objekte mit denen der Benutzer interagiert. Die nächst tiefere Schicht gibt der Welt ihre spezielle Anwendungslogik, die auf Funktionen in der darunterliegenden Ebene der Game-Engine zurückgreift. Die Engine ist in Module aufgeteilt, welche sich um unterschiedliche Aufgaben kümmern. Um Zugriff auf die Renderpipeline der Grafikkarte zu haben, benutzt das Grafikmodul eine tieferliegende API. Das Netzwerkmodul bildet selbst eine Schnittstelle zu außenliegenden Systemen. Die letzte Ebene dieses Modells ist das Betriebssystem auf dem die Anwendung läuft und die Verbindung zur Hardware darstellt.

Inzwischen sind zahlreiche Game-Engines auf dem Markt erschienen, von komplett freierhältlichen und offenen Systemen bis zur Software mit kostenpflichtigen Lizenzmodellen und verschlossenem Quellcode. Dazu gehören Ogre 3D[5], JMonkey Engine[6], Unreal Development Kit[7], Unity3D[8] und die CryEngine[9], um nur ein paar beliebte Systeme zu nennen. Unity3D ist eine der am weitest verbreiteten Engines und verfügt über eine ausführliche

[5] http://www.ogre3d.org/
[6] http://www.jmonkeyengine.com/home/
[7] http://www.udk.com/
[8] http://unity3d.com/
[9] http://mycryengine.com/

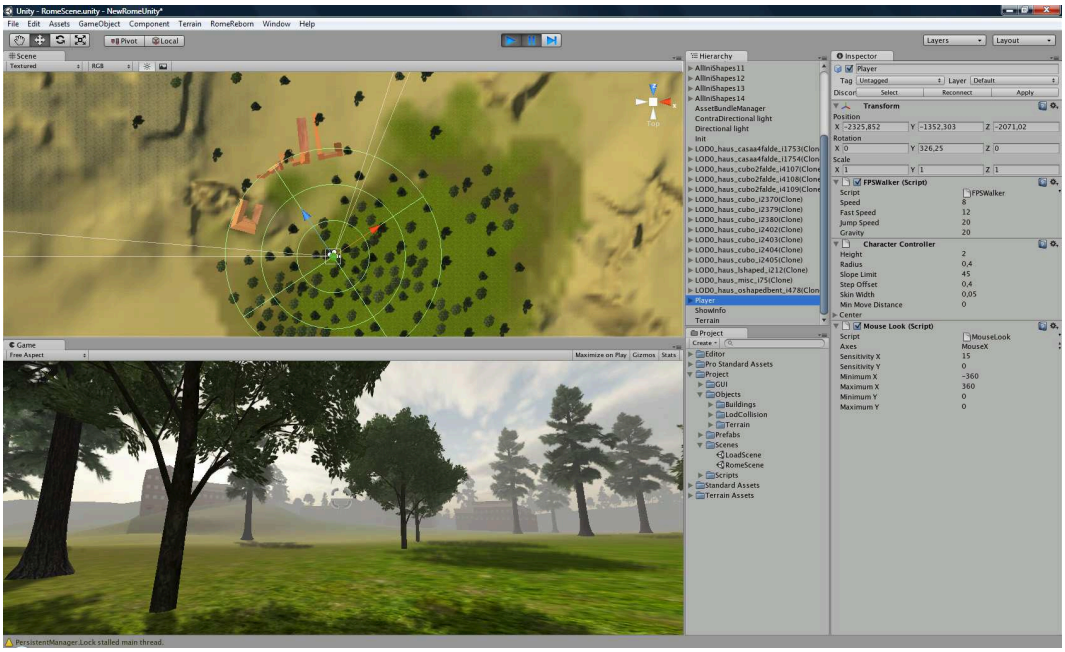

Abbildung 3.4: Arbeitsumgebung der Game-Engine Unity3D.

Dokumentation, aus welchem Grund sie als Fundament für die Implementation des datenbankbasierten LOD Systems dient. Abbildung 3.4 zeigt die typische Arbeitsumgebung der Middleware. Prinzipiell ist keine Game-Engine ausschließlich auf die Entwicklung von Spielen beschränkt, sondern lässt sich für jede Art von 3D-Echtzeitanwendungen benutzen.

Für die Implementation und den Arbeitsablauf bietet Unity3D einige Vorteile, die für das datenbankbasierte LOD System genutzt werden können und man nicht neu programmieren muss. Zum ersten lässt sich der Import und Export jeglicher Assets per Skript anpassen, wodurch zahlreiche Arbeitsschritte automatisiert werden und man viel Zeit spart. Unter einem Asset versteht man speziell in Unity3D eine Ressource, meistens eine Datei, auf die in der Anwendung zugegriffen werden kann, zum Beispiel Musik, Texturen oder Meshes. Ein weiteres wichtiges Attribut ist das Programmieren eigener Editorfunktionen. Speziell für die Umsetzung des Rome Reborn Projekts in eine Echtzeitanwendung müssen einige Vorarbeiten geleistet werden, die alle der ca. 20.000 Objekte in der Szene betreffen. Aus diesem Grund spielt die Automatisierung generell eine bedeutende Rolle. Außerdem stellt die Renderpipeline bereits den Effekt von Nebel zur Verfügung, dessen Eigenschaftswerte konfigurierbar sind. Unity 3D benutzt das Physiksystem PhysX von Nvidia[10], welches einige physikalische Verhalten in Echtzeit simuliert. Dazu zählt auch die Kollisionsbestimmung von Szenenobjekten. Zur Beschleunigung der räumlichen Abfragen benutzt PhysX eine hierarchische Datenstruktur, die Kollisionstest bedeutend performant gestaltet. Welche Datenstruktur konkret genutzt wird, konnte an dieser Stelle nicht herausgefunden werden.

[10] http://developer.nvidia.com/object/physx.html

Durch eine Programmierschnittstelle in Unity 3D kann dann auf die Kollision eines Objekts mit einem anderen reagiert werden, wobei die Schnittberechnungen selbst vom Modul des Physiksystems durchgeführt wird.

Das wichtigste Prinzip in Unity3D ist der logische Aufbau eines Objekts, welches sich später in der fertigen Anwendung befindet[11]. Ein Objekt, welches am Programm teilnimmt und sich dafür in der Szene befinden muss, wird als Spielobjekt bezeichnet. Ein Spielobjekt ist vorerst ein leerer Behälter ohne Logik oder Funktionen, welches dann durch Komponenten gefüllt werden kann. Die Komponenten behandeln unterschiedliche Aufgaben und fügen dem Objekt Logik und Funktionen hinzu, deren Eigenschaften editierbar sind. Zum Beispiel gibt die Transformationskomponente dem Spielobjekt eine Position, Rotation und Skalierung. Die Attributswerte lassen sich per Skript oder im Editor verändern. Selbstgeschriebene Skripte sind ebenfalls Komponenten und lassen sich an Spielobjekte anfügen, um sie mit eigenen Logiken auszustatten. Alle Skripte werden im Rahmen dieser Arbeit mit der Programmiersprache C# geschrieben, welche von Unity3D als Skriptsprache unterstützt wird. Aufgrund der Plattformunabhängigkeit nutzt die Game-Engine das offene Framework von Mono[12].

3.3 Zusammenfassung

In diesem Kapitel wurde die Renderpipeline und ihre möglichen Engpässe sowie die Benutzung von Game-Engines zur Entwicklung von interaktiven, virtuellen Welten vorgestellt. Die Renderpipeline lässt sich grob in drei Abschnitte, Anwendung, Geometrie und Rasterisierung, unterteilen. Während man in der Anwendung vollen Einfluss auf die Pipeline hat und sie durch Parallelisierung beschleunigen kann, bieten die zwei nächsten Stufen nur einen beschränkten Zugriff, weil sie teilweise fest in der Grafikkarte implementiert sind. Desweiteren erhöht die Wiederbenutzung von Meshes und Reduzierung der Draw Calls die Rendergeschwindigkeit. Game-Engines sind anwendungsunabhängige Systeme und verfügen über wesentliche Funktionen zur Konstruktion von 3D-Echtzeitapplikationen. Im Rahmen dieser Arbeit wird die Game-Engine Unity3D verwendet.

Das nächste Kapitel geht auf die Datenbasis des Rome Reborn Projekts ein, die für das Durchlaufen der Renderpipeline optimiert werden soll, sodass man möglichst viele Engpässe vermeidet. Dabei besitzen der Umfang und die besonderen Eigenschaften der 3D-Modelle eine wichtige Rolle für die Datenoptimierung. Diesbezüglich wird die Generierung der Modelle erläutert.

[11] Unity3D Basiswissen: http://unity3d.com/support/documentation/Manual/
[12] http://www.mono-project.com/

Kapitel 4

Datenbasis

Die primäre Absicht des Rome Reborn Projekts ist es die Stadt zur Zeit 320 n. Chr. nachzubilden und für wissenschaftliche sowie schuliche Zwecke zu präsentieren. Ungefähr zu dieser Zeit hatte das römische Reich die Spitze der Entwicklung errungen. Um ein realistisches Ergebnis zu erhalten, wurden historische, topografische und archeologische Informationen sowie begründete Theorien zusammengetragen und in ein virtuelles Modell der Hauptstadt übersetzt. Bislang realisierte man aus dem Stadtmodell Bilder, Filme[1] und die Integration in Google Earth[2]. Für diese Medien kamen hochaufgelöste Geometriemodelle und aufwändige Lichtberechnungen zum Einsatz, weil die Renderprozesse nicht in Echtzeit ausgeführt werden mussten. Bei der Umsetzung in eine interaktive, virtuelle Welt gelten aufgrund des zusätzlichen Zeitlimits der Renderpipeline jedoch strengere Kriterien, welche das vorige Kapitel behandelt. Hier kann nicht jedes Gebäude ohne Probleme im höchsten Detailgrad geladen und angezeigt werden. Damit gleich beim ersten Schritt, der Datenübernahme in die Echtzeitumgebung, alle vorteilhaften Eigenschaften des Stadtmodells von Rom bekannt sind, erläutert das Kapitel, wie dieses zustande kam.

4.1 Umfang des Rome Reborn Projekts

Das ganze Projekt umfasst das Rahmengelände Roms mit dem Fluss Tiber, Wälder und andere Vegetation, historisch wichtige Bauten, konventionelle Gebäude und Straßen. Alle Daten zusammengefasst machen einen Speicherumfang von ca. 14 GB aus. Abbildung 4.1 zeigt eine Übersicht von Rome Reborn mit den meisten genannten Elementen. Die Landschaft und historische Bauten entstammen aus geografischen Daten, welche ein naturgetreues Abbild von Rom gewährleisten. Der wesentliche Bestandteil Roms beinhaltet die Gebäude. Diese lassen sich in zwei Gruppen unterteilen. Historisch bedeutende Bauten bilden den qualitativ hochwertigen Teil und gehören den Modellen erster Klasse an, während andere Gebäude als Füllobjekte betrachtet werden, die den quantitativen Teil tragen und zu den Modellen zweiter Klasse gehören. 40 von insgesamt 200 historischen Modellen sind bereits vorhanden,

[1] http://www.romereborn.virginia.edu/gallery-current.php
[2] http://earth.google.com/rome/

Abbildung 4.1: Rome Reborn im Überblick. Zu sehen sind das Gelände und zahlreiche konventionelle Gebäude. Da auf diese Weise nicht alle ausmodellierten Gebäude darstellbar sind, werden hier nur die Grundformen angezeigt.

darunter zählen etwa das Forum von Julius Caesar, der Tempel der Venus und das Kolosseum. In der Gruppe zweiter Klasse befinden sich beispielsweise Wohnhäuser, Tempel und Regierungsgebäude. Ungefähr 7.000 bis 10.000 von ihnen sind in der Stadt platziert, wobei ein Gebäude aus mehreren Objekten bestehen kann und somit mehr als 22.000 Objekte in der Szene sind. Modelle erster Klasse wurden hinsichtlich der Qualität handmodelliert. Die Modellierung aller Gebäude Roms per Hand würde zu lange dauern, weshalb man die Modelle der zweiten Klasse prozedural generierte.

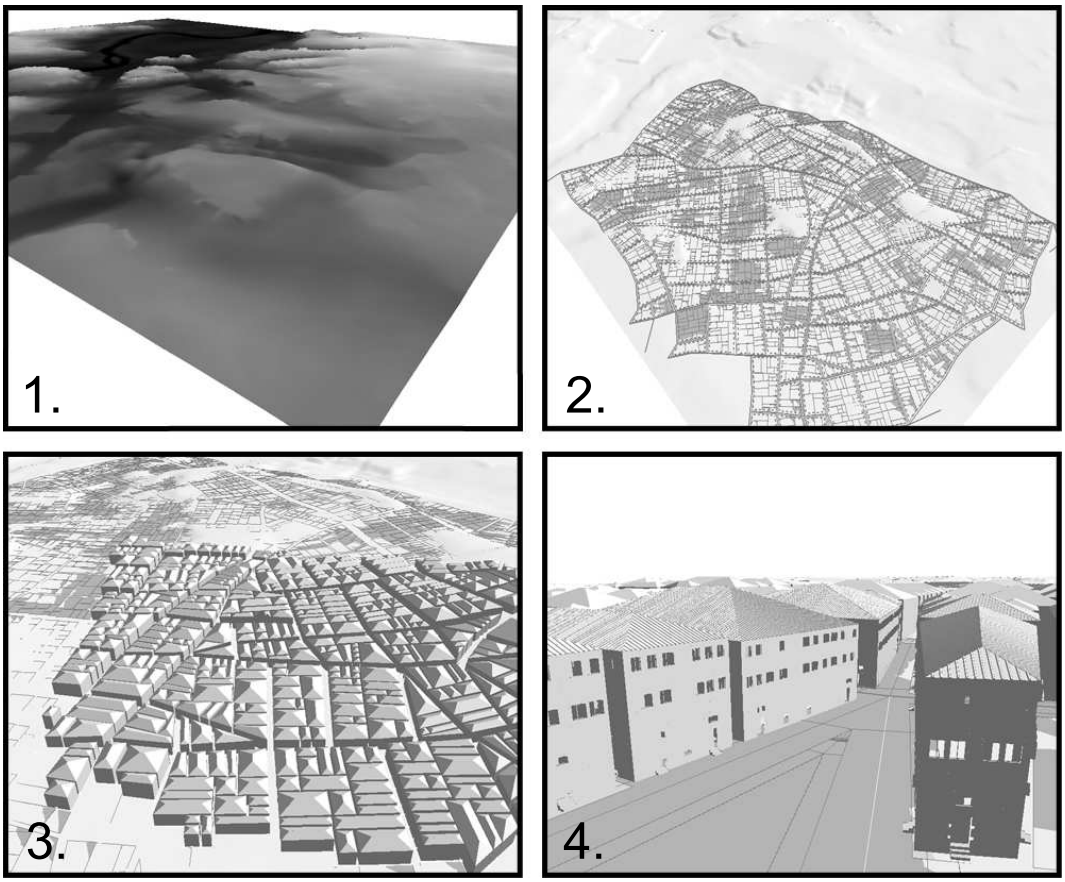

Abbildung 4.2: Prozedurale Erstellung einer Stadt mit der CityEngine: 1. Erstellung des Geländes per Höhenkarte. 2. Bau eines Straßennetzwerkes und die Einteilung der Zwischenräume in Grundstücksflächen. 3. Erstellung der Grundformen. 4. Generierung der finalen Modelle.

4.2 Prozedurale Modellierung von Gebäuden

Zur prozeduralen Erzeugung von dreidimensionalen Gebäudemodellen wurde die CityEngine von Procedural[3] genutzt. Hierfür lädt man zuerst die geografischen Daten in die Software, sodass alle Objekte an der richtigen Stelle im Gelände platziert werden. Eine Höhenkarte beschreibt die Landschaft und eine weitere schwarz-weiße Hinderniskarte definiert an schwarzen Stellen unpässliches Gelände, in dem keine Häuser vorkommen. Innerhalb eines ausgewählten Bereichs wird dann automatisch ein Straßennetzwerk errichtet und die Zwischenräume in Grundstücksflächen unterteilt. Alternativ kann man direkt originale Weltkarten mittels dem Open-Street-Map-Format importieren[4]. Auf den Grundstücksflächen werden dann Grundformen für die darin befindlichen Häuser erstellt. Eine Grundform besteht aus wenigen, ungefähr

[3] http://www.procedural.com/
[4] http://www.openstreetmap.de/

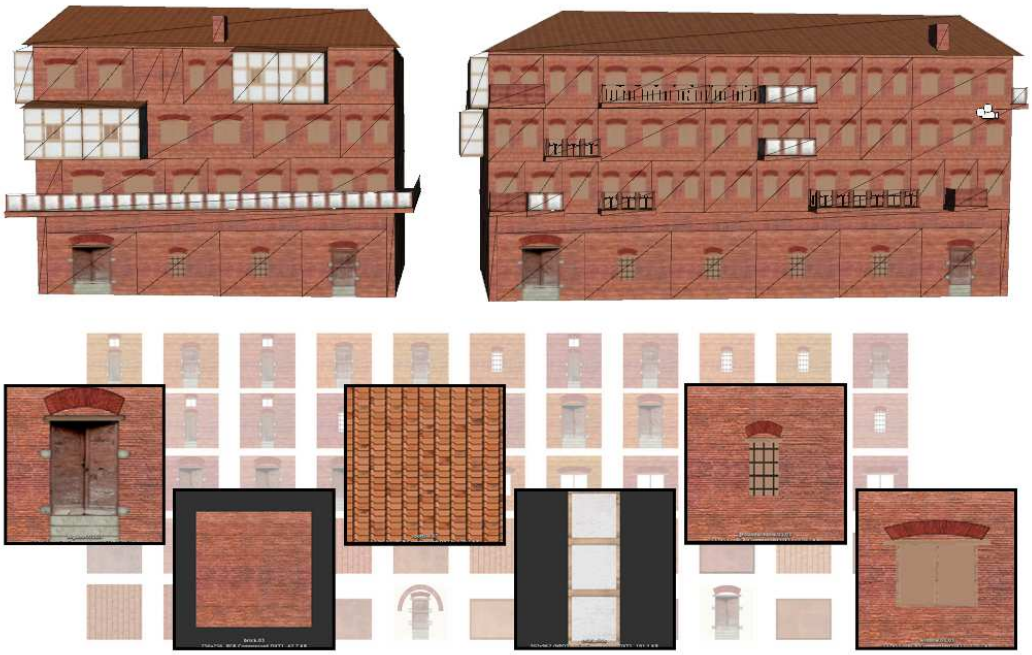

Abbildung 4.3: Gebäude werden prozedural aus kachelbaren Texturen zusammengesetzt. Zu sehen sind zwei gekachelte Häuser, die auf den selben Texturvorrat zugreifen.

4 bis 20 Vertexpunkten und spannt einen genauen Rahmen um das später ausmodellierte Haus auf. Sie dient der letztendlichen Modellgenerierung als Ausgangspunkt, den es zu verfeinern gilt. Aus den Grundformen werden dann im letzten Schritt die fertigen Gebäude modelliert. Abbildung 4.2 veranschaulicht die genannten Schritte nochmals.

Alle diese Arbeitsschritte werden anhand von benutzerdefinierten Regeln berechnet. Die Regeln sind teilweise von der CityEngine vorgegeben oder können per Skriptsprache programmiert werden. Zur Umsetzung des Rome Reborn Projekts definierte man einige Regeln für die verschiedenen Bautypen und den speziellen Charakter Roms. Diese Regeln werden folglich den Grundformen der Häuser zugewiesen. Dabei erhält jedes Objekt eine zufällige Nummer und eine eindeutige Identifikationsnummer (ID), bestehend aus Regelname und Objektnummer. Die Zufallszahl geht in die Modellberechnung ein und wird von der angehängten Regel ausgewertet. Bestimmte Verarbeitungen und Selektionen werden innerhalb der Regelkette mittels der Zufallzahl entschieden, weshalb kein Gebäude mehr als ein Mal im Projekt vorkommt.

Eine Regel zur prozeduralen Modellierung von Gebäuden besteht hauptsächlich aus der Ermittlung geometrisch wichtiger Punkte, der Unterteilung in feinere Geometrien und dem Extrudieren von Polygonen [PM06]. Dabei wird die Grundform beispielsweise durch Stockwerke, Türen, Fenster und Dächer spezialisiert. Die Kachelung von Meshes spielt bei der

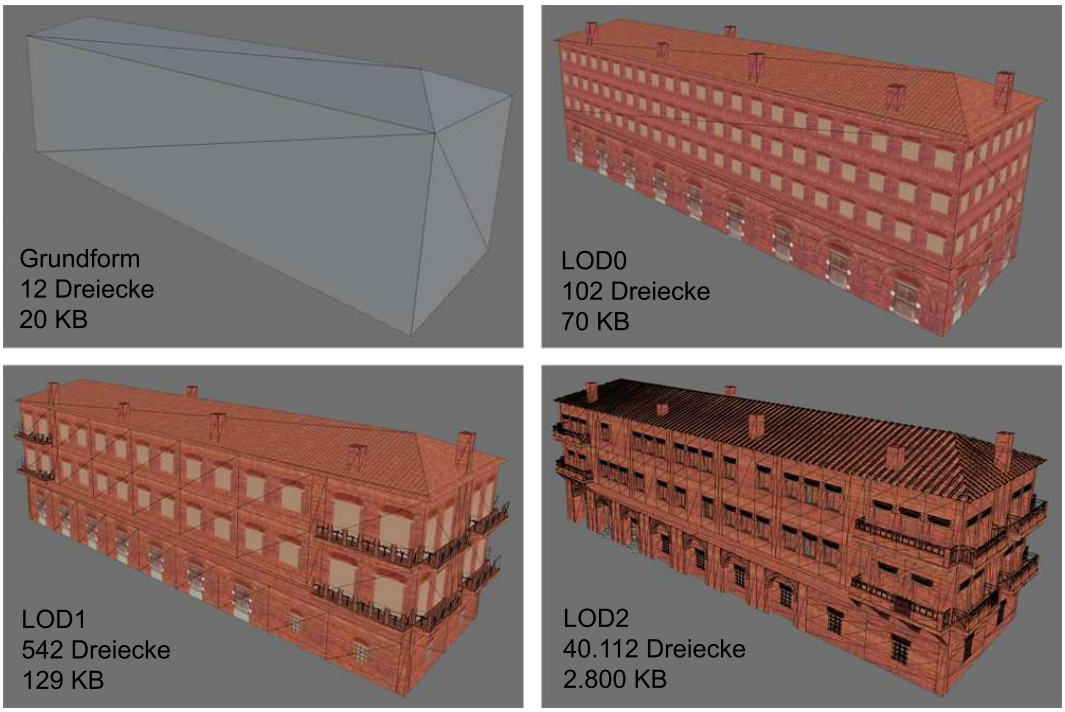

Abbildung 4.4: Die Grundform und LOD Stufen eines Gebäudes aus Rome Reborn mit entsprechender Dateigröße und Polygonanzahl.

prozeduralen Modellierung eine wesentliche Rolle [PM07]. Hier wird das Modell in angrenzende Rechtecke unterteilt, die in einem bestimmten Muster angeordnet sind (Abbildung 4.3). Insbesondere die Hauswände mit Fenstern und Türen werden gekachelt. Auf diese Weise lässt sich der Inhalt einer Kachel auch auf andere anwenden. Die meisten Details im Rome Reborn Projekt liegen in den Texturen, das heißt Fenster, Mauern und Türen sind in Bildern abgespeichert. Die Erstellungsregel bezieht dann Größe und Verhältnis der Texturen mit in die Modellierung ein und legt sie wiederholt auf die einzelnen Kacheln (Abbildung 4.3). Der Texturvorrat für Rom misst ungefähr 38 MB und umfasst 114 Bilder mit denen alle Gebäude erstellt werden. Durch die vielen Variationen der Grundformen und Kacheln sowie der Zufallszahl bildet sich dennoch immer ein individuelles Muster.

Am Anfang einer Regel lassen sich bestimmte Attribute festlegen, die dem Anwender offengelegt werden und er somit die Werte jederzeit beliebig verändern kann. Innerhalb der Regel dienen die Attribute zur prozeduralen Modellierung, sodass ein angepasster Attributswert auch zu einem alternativen Ergebnis des Objekts führt. Die Regeln für die Gebäude Roms lassen sich ebenfalls auf diese Weise einstellen. Darunter fällt auch ein Level of Detail Attribut, welches den Detailgrad des fertigen Modells angibt. Der Wert kann auf 0,1 oder 2 gesetzt werden, wobei 0 das gröbste Resultat und 2 das feinste Modell liefert. Zusammen mit der Grundform hat ein Objekt also vier unterschiedliche visuelle Repräsentationen, wie

Abbildung 4.4 zeigt. Während das Modell der niedrigsten LOD Stufe (LOD0) nur Texturen für die Darstellung von Fenstern und Türen benutzt sowie keine Balkone besitzt, sind diese im LOD2 Modell enthalten und Türen plus Fenster durch Geometrien ausmodelliert. LOD1 stellt einen Zwischenschritt dar, bei dem die Balkone durch transparente Texturen gebildet werden. Generell lassen sich auch Innenräume von Gebäuden prozedural erstellen. Die Regeln des Rome Reborn Projekts sehen dies aber nicht vor. Die Möglichkeit von begehbaren Gebäuden in der Echtzeitanwendung wird jedoch nicht ausgeschlossen.

4.3 Ergebnisse der Basisdatenuntersuchung

Anhand der untersuchten Basisdaten lassen sich schon konkrete Schlussfolgerungen ziehen, welche in die Entwicklung des datenbankbasierten LOD- und Szenenmanagementsystems integriert werden. Der erste Vorteil liegt in der Kachelung der Gebäude, bei dem die enthaltene Textur auf das gleiche und auf andere Objekte wiederverwendet wird. Da nicht jedes einzelne Haus eine eigene vorgefertigte Textur verwendet, sondern alle Modelle auf einen gemeinsamen Vorrat zugreifen, verbraucht das Verfahren weniger Speicherplatz. Redundante Informationen werden beseitigt. Diese Eigenschaft kann man sich in der Echtzeitanwendung zu Nutzen machen, indem die Modelldatenbank und der Texturvorrat getrennt voneinander gelagert werden. Somit liegt eine Textur weder mehrfach im Speicher, noch muss sie mehrfach geladen werden, wie es bei einer direkten Verknüpfung von Modellen mit benutzten Texturen der Fall wäre. Zur Laufzeit muss nur sichergestellt werden, dass für ein geladenes Modell die verwendeten Bilder ebenfalls geladen und zugewiesen werden. Auf diese Weise spart man Speicherplatz und optimiert den Ladevorgang.

Eine weitere vorteilhafte Eigenschaft der Modellbasis lässt sich aus Abbildung 4.4 entnehmen. Der begrenzende Rahmen der Grundformen wird durch die ausmodellierten Gebäude nicht wesentlich erweitert und bildet daher ein exaktes Bounding Volumen für das Objekt. Zusätzlich wird dieser Rahmen durch wenige Vertexpunkte aufgespannt. Aufgrund dieser beiden Merkmale eignet sich die Grundform zur effektiven Kollisionsbestimmung. Die visuelle Darstellung und die Kollisionsbestimmung für ein Objekt sind nicht voneinander abhängig. Das heißt es kann ein LOD Modell zur Visualisierung und die Grundform zur Kollisionsberechnung für das selbe Objekt dienen. Im Gegensatz zur Kollisionsbestimmung durch ein LOD2 Modell beschleunigt das Verfahren den Rechenprozess bedeutend. Ein Zuschlag an Daten pro Objekt entsteht dabei, der jedoch wegen der geringen Größe von 4 bis 20 Vertexpunkten vernachlässigt werden kann.

Der letzte Rückschluss lässt sich durch die bereits in der Datenbasis vorhandenen statischen LOD Stufen ziehen. Damit stehen die Phasen der LOD Generierung und des Austauschs fest. Das datenbankbasierte LOD System wird also auf einem diskreten LOD Verfahren mit drei Detailstufen basieren. Zu welchem Zeitpunkt man eines der Repräsentationen zur Darstellung nutzt, also die LOD Auswahl, steht hier noch nicht fest und wird im nächsten Kapitel erarbeitet. Zwei weitere Gründe sprechen für den Gebrauch von diskreten LODs.

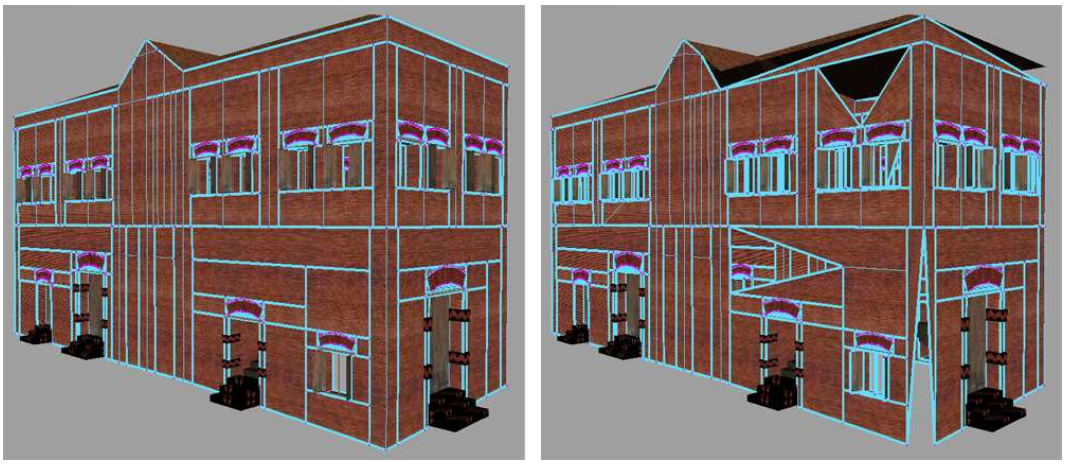

Abbildung 4.5: Diese deutlich sichtbaren Fehler entstehen bei der Wegnahme von Vertexpunkten im Modell.

Durch die Wiederverwendbarkeit von Objekten, welches ein diskretes LOD erlaubt, lässt sich ein Gebäude bei gleichem Speicherverbrauch mehrmals instanziieren. Obwohl im Moment kein Objekt doppelt in der Szene vorkommt, ist der Gedanke für zukünftige Optimierungen vorteilhaft. Außerdem wäre das visuelle Resultat der kontinuierlichen LOD Vereinfachung der Gebäude Roms nicht befriedigend. Häuser sind meistens rechteckige Körper, das heißt sie besitzen kaum Rundungen, die durch Geometriereduzierung vereinfacht werden könnten. Durch das prozedurale Verfahren bestehen die Modelle schon aus den minimalen Vertexpunkten. Die Wegnahme eines Vertexpunktes am Rand der Geometrie verändert die äußere Form des Objekts und innerhalb der Geometrie wird die Kachelung der Texturen zerstört. Beide Operationen führen zu einem deutlich sichtbaren Fehler, wie Abbildung 4.5 zeigt.

4.4 Historische Informationen

Aktuell existiert keine veröffentlichte Datenbank mit historischen oder sonstigen Informationen für das Rome Reborn Projekt. Allerdings sollen in der interaktiven, virtuellen Welt verschiedene Informationen zu einzelnen Objekten abruf- und einfügbar sein. Wer Daten editieren darf oder woher die Informationen genommen werden ist noch nicht definiert. Ungeachtet dessen steht die Verknüpfung von Objekten mit Informationen oder Verweisen fest.

4.5 Zusammenfassung

In diesem Kapitel wurde die Datenbasis des Rome Reborn Projekts auf vorteilhafte Eigenschaften untersucht, die bei der Speicher- und Performanceoptimierung für die 3D-Echtzeitanwendung hilfreich sind. Das Projekt umfasst ca. 14 GB Speicherplatz, wobei der ausschlaggebende Teil die Gebäude bilden. Davon wurden die 40 historisch bedeutenden Bauten per Hand modelliert. Die restlichen 7.000 - 10.000 Bauwerke generierte man auf-

grund des enormen Arbeits- und Zeitaufwands prozedural. Das prozedurale Verfahren erstellt zuerst Grundformen der Häuser, die in der 3D-Anwendung angesichts der Performanceoptimierung zur Kollisionsberechnung dienen. Danach werden anhand von benutzerdefinierten Regeln drei statische LOD Modelle erzeugt, die man zur visuellen Repräsentation des Objekts nutzt. Das datenbankbasierte LOD System wird demnach auf einem diskreten LOD basieren. Bei der prozeduralen Modellierung werden die Gebäude aus Kacheln zusammengesetzt, auf denen die Texturen von Fenstern, Türen und Wänden liegen. Dabei greifen alle Modelle auf den selben Vorrat an Bildern zurück. In der Echtzeitanwendung können so Texturen und Modelle seperat behandelt werden, was den Speicherplatz und die Ladezeit optimiert. Allein die Abhängigkeiten zwischen ihnen muss man beachten.

Nachdem die grundlegenden Daten für das Projekt untersucht wurden, behandelt das nächste Kapitel das im Rahmen dieser Arbeit erarbeitete Konzept und die Umsetzung des datenbankbasierten LOD- und Szenenmanagementsystems, angefangen vom Aufbau der Datenbank über die Vorbereitung der Szene bis zum Anzeigen des LOD Modells.

Kapitel 5

Entwicklung des datenbankbasierten LOD und Szenenmanagementsystems

Die Datenbasis, die es in Echtzeit darzustellen gilt, wurde im letzten Kapitel untersucht und daraus wichtige Erkenntnisse gewonnen, die nun bei der Entwicklung hilfreich sind. Dieses Kapitel behandelt das im Rahmen dieser Arbeit entwickelte LOD und Szenenmanagementsystem. Hierbei werden die Erkenntisse aus dem Stand der Technik berücksichtig, übernommen, kombiniert und mit neuen Verfahren ausgestattet. Es wird das allgemeine Konzept sowie dessen Umsetzung mit der Game-Engine Unity3D erläutert.

5.1 Ziele und Voraussetzungen

Zuerst werden nochmal die Probleme, die Ziele des Systems, beziehungsweise daraus resultierenden Voraussetzungen und der derzeitige Erkenntnisstand wiederholt:

- Die interaktive Echtzeitdarstellung einer Szene mit zahlreichen Geometriemodellen ist heutzutage ohne Optimierungen nur in einem begrenzten Maß möglich, weil die Szene zu viel Speicherplatz verbraucht, um in der Gesamtheit geladen zu werden. Deshalb muss ein Managementsystem schrittweise die benötigten Objekte laden, während es unnötige entlädt. Welche Objekte relevant oder irrelevant sind, erkennt es anhand von Regeln, die man in jedem Frame auf alle Objekte anwendet. Die Ladeoperationen selbst sollen performant und ohne Kenntnisnahme des Benutzers ausgeführt werden.
- Die Voraussetzung dafür ist, dass alle in der Szene befindlichen Modelle entfernt und in einer Objektdatenbank abgelegt werden. Bei diesem Export sollen keine redundanten Daten mehr in den Modellen vorkommen, was die Ladezeit beschleunigt und den Speicherverbrauch vermindert. Beim Start der Anwendung enthält die Szene also keinerlei Modelle. Auf diese Weise kann man die virtuelle Kamera beliebig in der Szene platzieren und zur Laufzeit werden dann exportierte Modelle geladen und entladen.

- Sind nun innerhalb eines Zeitabschnitts die benötigten Objekte geladen, kann man nicht alle von ihnen in der höchsten Detailstufe anzeigen, da der Renderprozess für eine Echtzeitdarstellung zu viel Zeit beanspruchen würde. Aus diesem Grund wird das diskrete LOD mit drei Detailstufen genutzt, wie es aus dem letzten Kapitel hervorgeht. Der Qualitätsverlust für das menschliche Auge soll dabei kaum wahrnehmbar sein. Außerdem gilt es das LOD System mit dem Szenenmanagementsystem zu verbinden.

- Generell sollen die Eigenschaften aller Regeln und Verfahren individuell einstellbar sein.

- Zu den einzelnen Szenenobjekten sollen sich verschiedene Informationen hinterlegen lassen, die man in der Anwendung abrufen und verändern kann. Eine zentrale Datenbank dient dazu die IDs der Objekte mit anderen Daten zu verknüpfen.

- Man soll einzelne Modelle aus der Objektdatenbank herunterladen und beliebig verändern können. Das Halten von Modellen in separaten Dateien wird also vorausgesetzt.

5.2 Konzept

Für das datenbankbasierte LOD und Szenenmanagementsystem wurde folgendes Konzept entwickelt, welches die genannten Probleme löst und die Ziele umsetzt. Das Konzept geht nur auf die wesentlichen Grundideen des Systems ein und nicht auf die konkrete Implementierung in der Game-Engine. Allgemein lässt sich die Arbeit in drei Abschnitte unterteilen: Erstellung der Objektdatenbank, Szenenvorbereitung und Laufzeitprozess. Die Erstellung der Objektdatenbank und die Szenenvorbereitung sind Prozesse, die vor der Laufzeit der Anwendung und nur einmalig durchgeführt werden müssen. Der Hauptteil des LOD und Szenenmanagementsystems befindet sich im Laufzeitprozess.

5.2.1 Erstellung der Objektdatenbank

Die Objektdatenbank soll alle Modelle des Projekts umfassen. Da sie sich aufgrund der enormen Speicherauslastung nicht standardmäßig in der Szene befinden können, werden alle Modelle als Dateien auf der Festplatte gelagert. Zur Laufzeit können diese dann je nach Bedarf dynamisch in die Anwendung geladen und entladen werden. Dabei wird die Objektdatenbank im ersten Abschnitt so erstellt, dass für jedes Objekt innerhalb der Szene eine Datei pro LOD Modell mit eindeutiger ID angelegt wird. Somit kann gezielt auf eine bestimmte Detailstufe eines Objekts zugegriffen werden. Beim Anlegen der Dateien werden redundante Informationen innerhalb aller LOD Modelle gesucht und aussortiert. In diesem Fall sind das die Materialien und Texturen, welche sich immer wieder in den Dateien wiederholen. Diese Daten trennt man vom Modell und speichert sie nur ein Mal separat mit einer eindeutigen ID ab. Je Textur und Material wird also eine Datei erstellt, wodurch man ebenfalls auf eine bestimmte Information zugreifen kann. Durch das Entfernen benötigter Daten entsteht automatisch eine Abhängigkeit zwischen den Modellen, Materialien und Texturen. Beispielsweise steht ein Modell, dem die Informationen zu seinem Material und der

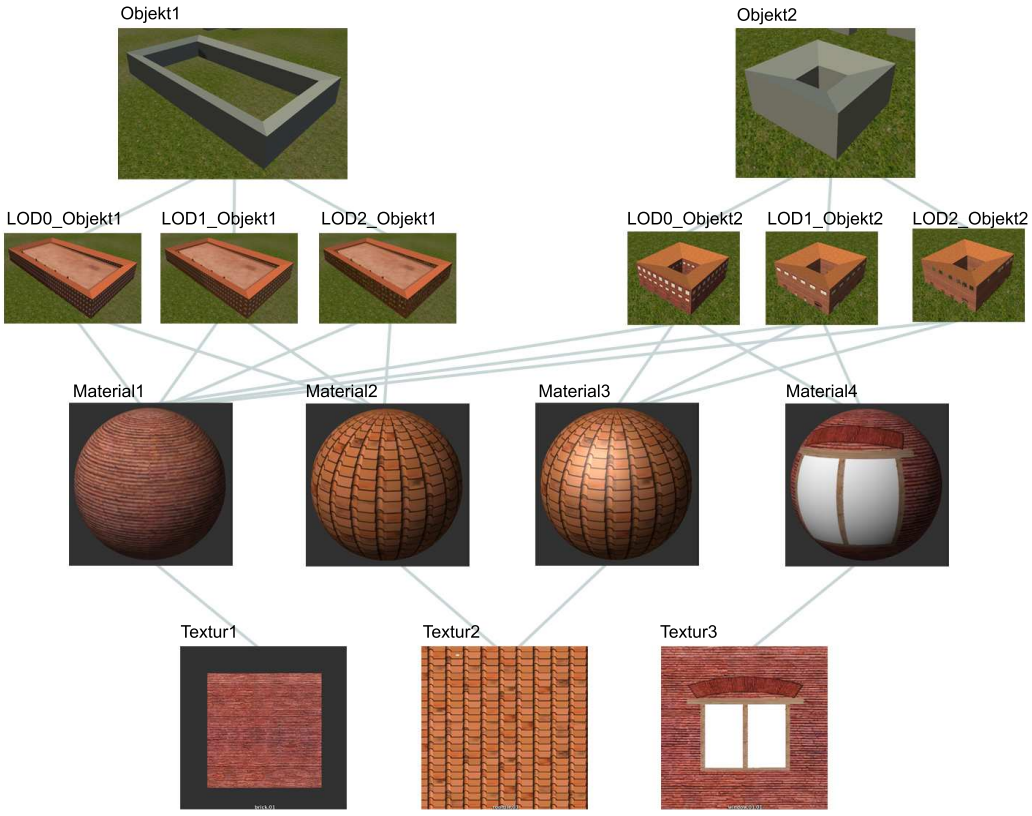

Abbildung 5.1: Erstellung der Objektdatenbank: Zu sehen sind verschiedene Modelle, die die selben Materialien referenzieren, die teilweise wieder die selbe Textur referenzieren.

daraufliegenden Textur entzogen wurden, in Abhängigkeit zu diesen. Damit diese Verbindungen nicht verloren gehen, werden die entsprechenden IDs der benötigten Informationen in der davon abhängigen Datei gespeichert (Abbildung 5.1). Rotations- und Positionsdaten des Modells in der Szene sind ebenso enthalten. Aufgrund der Separation kann man einzelne Modelle jederzeit editieren. Veränderungen in Materialien oder Texturen sind sofort für alle abhängigen Dateien gültig, ohne die Referenzen aktualisieren zu müssen.

5.2.2 Szenenvorbereitung

Im zweiten Abschnitt wird die Szene für das Verfahren zur Laufzeit vorbereitet. Da alle Gebäudemodelle aus der Szene entfernt wurden, befindet sich nur noch das Gelände darin, welches durch ein sichtabhängiges LOD System berechnet wird. Als nächstes wird die virtuelle Kamera angepasst. Um sie herum werden drei Sphären mit benutzerdefinierten Radien und eindeutigen IDs gehängt, sodass sich die Kamera im Mittelpunkt befindet und die

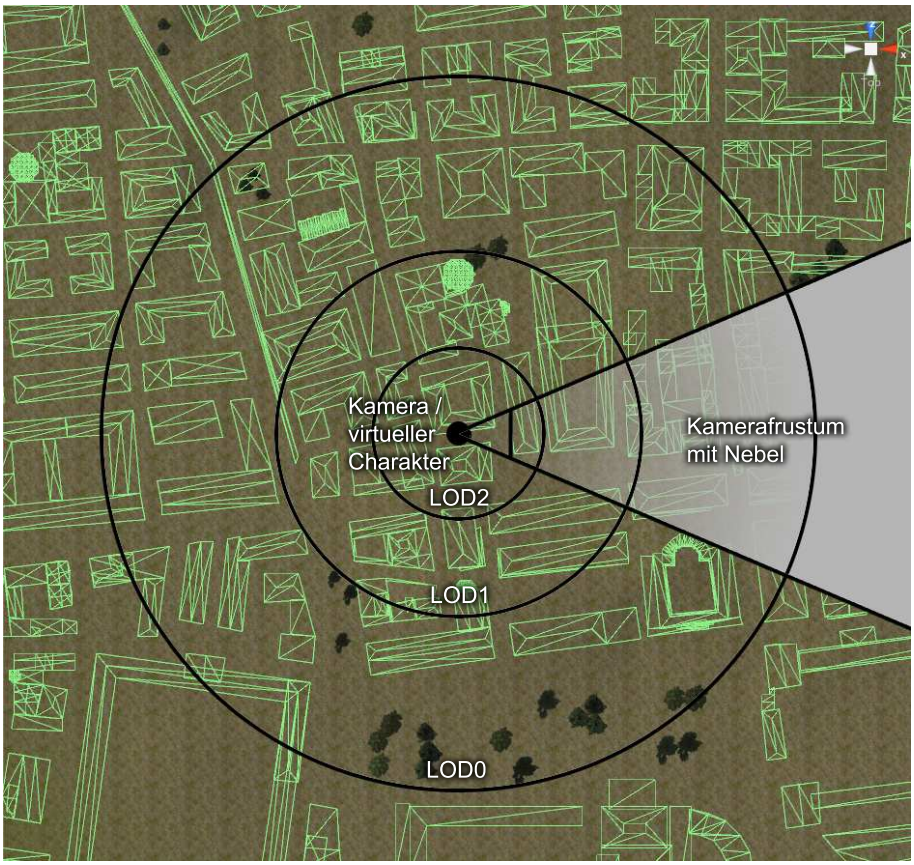

Abbildung 5.2: Das Ergebnis der Szenenvorbereitung: - Die virtuelle Kamera mit drei angehängten Kollisionssphären, welche die Bereiche für die LOD Stufen definieren. - Der Nebeleffekt, welcher Objekte hinter der größten Sphäre komplett verdeckt. - Die Grundformen als Kollisionsmodelle der Szenenobjekte.

Sphären sich immer synchron mit ihr bewegen. Die Sphären stehen für die drei LOD Stufen, weshalb man den Kugelradius für $LOD0$ (niedrigste Detailstufe) am größten und den für $LOD2$ (höchste Detailstufe) am kleinsten wählen sollte. Innerhalb dieser Bereiche werden später die entsprechenden LOD Modelle geladen und angezeigt. Hier ist wichtig, dass es sich bei den Sphären nicht um einfache Distanzangaben handelt, die in jedem Frame pro Objekt abgefragt werden, wie es Kapitel 2 vorstellt. Die Sphären sind geometrische Beschreibungen einer Kugel, die zu Kollisionszwecken genutzt werden und in der Anwendung nicht sichtbar sind. Danach wird für die Kamera ein Nebeleffekt angepasst, der Objekte hinter der größten Sphäre vollständig verdeckt. Im folgenden Schritt bereitet man alle Objekte, die sich ursprünglich in der Szene befunden haben, für zukünftige Ladeoperationen vor. Damit ein Modell in der Objektdatenbank weis, wann es geladen und angezeigt werden soll, braucht es einen entsprechenden Stellvertreter, der bereits in der Szene platziert ist. Für diesen Zweck verwendet man die Grundformen der Objekte, welche eine eindeutige ID des zu repräsentie-

renden Objekts besitzen. Alle Grundformen werden ebenso nur als Kollisionsmodelle an die Objektpositionen gestellt und sind in der Anwendung nicht sichtbar. Sie dienen erstens für das Kollisionsereignis mit einer Sphäre und zweitens zur Kollisionsbestimmung für den virtuellen Charakter, bzw. der Kamera. Damit man Schnittberechnungen für Kollisionsabfragen zur Laufzeit wesentlich schneller durchführen kann, wird im letzten Schritt eine hierarchische Datenstruktur für alle Kollisionsobjekte angelegt. Abbildung 5.2 veranschaulicht das Ergebnis der Szenenvorbereitung.

5.2.3 Laufzeitprozess

Die Voraussetzungen für den Laufzeitprozess sind geschaffen. Im Ausgangszustand befinden sich lediglich das Gelände, die Kamera mit drei LOD begrenzenden Sphären und die Grundformen in der Szene. Nun verbindet der Laufzeitprozess das LOD mit dem Szenenmanagement, indem das Verfahren nur das aktuell benötigte Detailmodell eines Objekts lädt und danach anzeigt. Das Szenenmanagementsystem berücksichtigt also das vorhandene LOD System, wodurch Speicherplatz gespart und Performance gewonnen wird. Das Verfahren läuft wie folgt ab:

Als erstes werden alle Texturen und Materialien vorgeladen, sodass man in der Anwendung direkt auf sie zugreifen kann. Der Grund dafür ist, dass schon bei wenig angezeigten Gebäuden fast alle Texturen und Materialien benutzt werden und sie nicht viel Speicherplatz wegnehmen. Der Aufwand für ein dynamisches Laden und Entladen ist demnach größer als alle Dateien initial zu laden. Danach startet die Anwendung und es wird in jedem Frame die Kollision für nicht statische Objekte berechnet, zu denen die virtuelle Kamera und die drei Sphären gehören. Es wird dabei zwischen zwei Kollisionsereignissen unterschieden. Zum einen gibt es das Eindringen in ein Objekt, also das Ereignis, bei dem sich zwei Objekte zum ersten Mal schneiden. Zum anderen gibt es das Verlassen eines Objekts, also das Ereignis, bei dem sich zwei kollidierte Objekte zum ersten Mal nicht mehr schneiden. Zunächst wird hier auf Ersteres eingegangen. Dringt eine Sphäre in eine Grundform ein, so wird dieses Ereignis abgefangen und darauf reagiert. Durch das Zusammensetzen ihrer beiden IDs entsteht eine weitere ID, welche auf eine bestimmte Datei in der Datenbasis verweist (Abbildung 5.3). Die Datei enthält das entsprechende LOD Modell für das Objekt. Dringt beispielsweise die Sphäre $LOD0$ in das Objekt $Objekt1$ ein, so bildet sich die eindeutige ID $LOD0\_Objekt1$. Ist diese Datei bereits geladen sowie das enthaltene LOD Modell in der Szene platziert, so muss man es nur noch auf sichtbar setzten und alle anderen in der Szene befindlichen LOD Modelle des selben Objekts vom Renderprozess ausschließen. Ist dies nicht der Fall, gilt es die Datei erst zu laden und dann das enthaltene Modell darzustellen. Durch die kollisionsbasierte LOD Auswahl umgeht man das Problem der Abstandsmessung bei der distanzbasierten Auswahl. Dies hat den Vorteil, dass frühzeitiges oder verspätetes Popping der LOD Modelle verhindert wird.

Innerhalb eines Frames kann jede Sphäre in mehrere Grundformen eindringen, was zu zahlreichen und unkontrollierten Ladeoperationen führen würde. Aus diesem Grund wer-

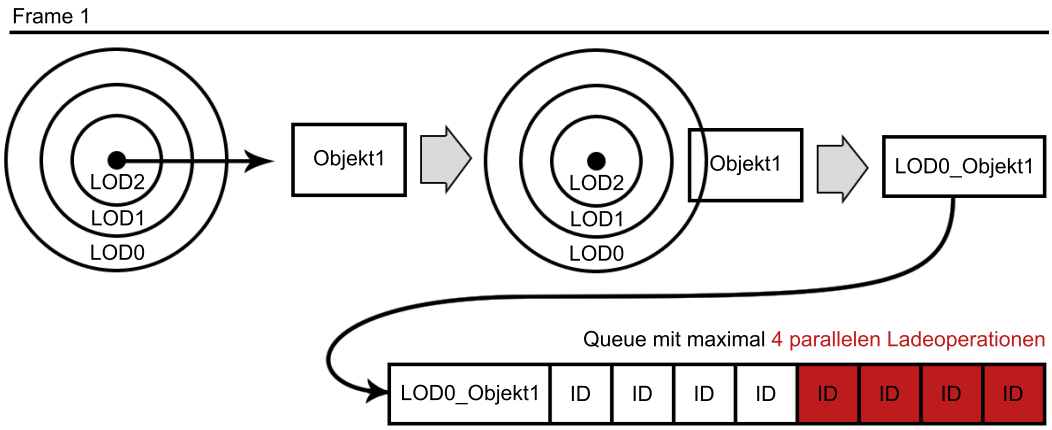

Abbildung 5.3: Die Reaktion auf ein Kollisionsereignis: Der virtuelle Charakter bewegt sich in Frame 1 in Richtung $Objekt1$, sodass die Sphäre $LOD0$ in das Kollisionsobjekt eindringt. Darauf wird die Ladeanfrage für die entsprechende Datei $LOD0\_Objekt1$ an das Ende des Queues gehängt.

den die benötigten Dateien nicht sofort geladen, sondern zuerst ihre IDs als Anfrage für eine Ladeoperation an einer zentralen Stelle gesammelt (Abbildung 5.3). Ein Queue verwaltet und verarbeitet die Anfragen, womit er zur besseren Kontrolle plus Übersicht dient. Wie üblich werden hierbei die Anfragen als erstes verarbeitet, die man auch als erstes in den Queue eingefügt hat. Auf diese Weise werden immer die dringlichsten Dateien zuerst geladen. Die wichtige Eigenschaft der Ladeoperationen ist, dass sie durch asynchrone Funktionen ausgeführt werden. Das bedeutet der Vorgang findet in einem Nebenprozess statt auf dessen Beendigung der Hauptprozess, also die Anwendung, nicht warten muss. Das Laden einer großen Datei kann durchaus einige Frames andauern, weshalb die Ausführung nicht im Hauptprozess stattfinden sollte, da sonst die Bedingung für eine Echtzeitanwendung nicht mehr erfüllt wird. Mithilfe der asynchronen Funktionen kann man außerdem mehrere Dateien parallel laden, was bezüglich der Renderpipeline einen erheblichen Geschwindigkeitsvorteil bedeutet. Die maximale Anzahl der parallelen Ladeoperationen kann vom Benutzer beliebig gewählt werden, um das Laufzeitverhalten besser zu kontrollieren.

In jedem Frame fügt man also alle IDs der benötigten Dateien an das Ende des Queues hinzu, wodurch sich automatisch eine Warteschlange bildet (Abbildung 5.3). Bevor eine Anfrage aber endgültig im Queue landet, sortiert man zunächst unnötige Anfragen aus. Dieser Vorgang findet auch statt, wenn sich das nachgefragte Modell bereits in der Szene befand. Als erstes werden alle anderen IDs, deren Dateien LOD Modelle für das selbe Objekt enthalten, aus dem Queue entfernt. Dabei spielt es keine Rolle, ob sich die Anfragen noch in der Warteschleife oder bereits im Ladezustand befinden. Wird eine Datei geladen, bricht man die Operation ab und verwirft die ID. Diese Anfragen gehen aus früheren Frames hervor und können gelöscht werden, weil sie der aktuellen Situation nicht mehr entsprechen. Solche unnötigen Anfragen kommen vor allem beim schnellen Durchlaufen der Szene und/oder bei

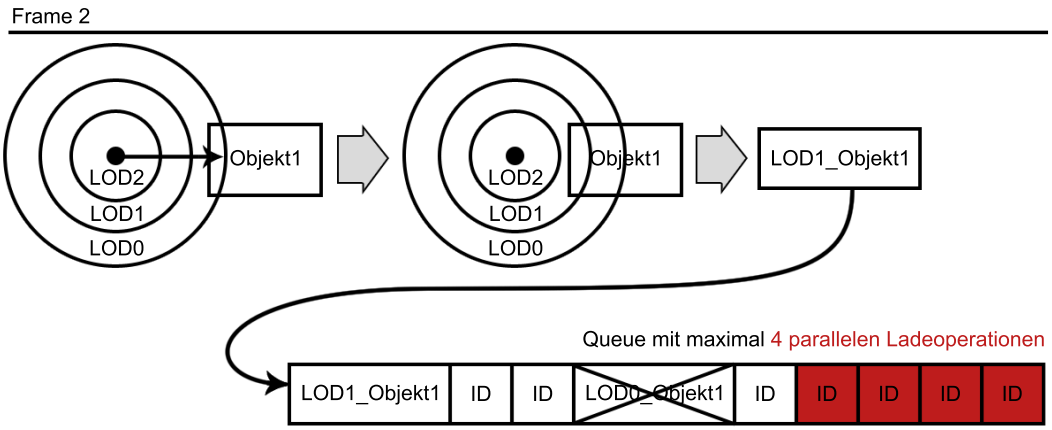

Abbildung 5.4: Die Reaktion auf ein Kollisionsereignis: Der virtuelle Charakter bewegt sich in Frame 2 weiter in Richtung $Objekt1$, sodass die Sphäre $LOD1$ in das Kollisionsobjekt eindringt. Darauf wird die Ladeanfrage für die entsprechende Datei $LOD1\_Objekt1$ an das Ende des Queues gehängt und alle anderen Anfragen entfernt.

einer geringen Anzahl an maximalen, parallelen Ladeoperationen vor. Zum Beispiel dringt im ersten Frame in Abbildung 5.3 die Sphäre $LOD0$ in $Objekt1$ ein. Die entsprechende ID wird an den Queue gehängt, kann jedoch nicht gleich geladen werden, da sich noch andere Anfragen vor ihr befinden. Bewegt sich die virtuelle Kamera weiter in Richtung des Objekts, so dringt in Frame 2 die Sphäre $LOD1$ ebenfalls in das Objekt ein (Abbildung 5.4). Somit fügt man eine weitere Anfrage für das selbe Objekt hinzu und die alte Anfrage wird ungültig. Das Prinzip gilt auch in umgekehrter Richtung, das heißt, wenn die Sphäre $LOD1$ das Objekt verlässt und $LOD0$ eindringt. Nachdem die unnötigen IDs entfernt wurden, überprüft man, ob die angefragte ID schon im Queue enthalten ist und folglich selbst ungültig wird. Ist dies der Fall, verwirft man die neue Anfrage und falls nicht, wird sie endgültig an das Ende des Queues gesetzt. Solche doppelten Anfragen kommen in Situationen vor, in denen der Benutzer schnell und oft seine Laufrichtung wechselt. Als Folge durchdringt und verlässt eine Sphäre mehrmals das gleiche Objekt.

Das zweite Kollisionsereignis, bei dem eine Sphäre eine Grundform verlässt, wird ebenfalls abgefangen und der selbe Prozess wie beim Eindringen gestartet. Dies ist möglich, weil das Verlassen eines Kollisionsobjekts der Sphäre S zugleich das Eindringen in das Objekt für die Sphäre $S-1$ bedeutet. Beim Verlassen einer Grundform der Sphäre S wird also das eben beschriebene Eindringverfahren für die Sphäre $S-1$ gestartet. Dieses Prinzip funktioniert solange $S > 0$ ist. Verlässt die letzte Sphäre 0 ein Kollisionsobjekt, wird eine andere Funktion aufgerufen. Hierbei liegt das Objekt außerhalb des Einflussbereichs aller Sphären und wird somit vollständig vom Nebel verdeckt. Aus diesem Grund werden alle Repräsentationen des Objekts überflüssig. Restliche Anfragen bezüglich dieses Objekts werden aus dem Ladequeue gelöscht sowie die instanziierten Modelle in der Szene zerstört. Analog dazu entlädt man die Dateien, um den Speicherplatz für zukünftige Ladeoperationen freizugeben. Wichtig ist,

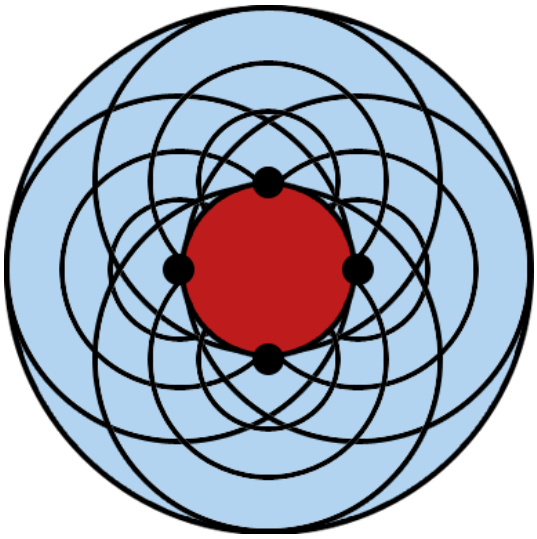

Abbildung 5.5: Durch das Halten von geladenen Modellen werden zeitaufwändige Ladeoperationen reduziert sowie die Performance der Anwendung optimiert. Zu sehen ist ein definierter Raum in rot, für den keine Ladeoperationen notwendig sind, solange sich der Benutzer eingeschlossen der Sphären innerhalb des blauen Bereichs aufhält.

dass alle LOD Modelle erst beim kompletten Austreten des Einflussbereichs entfernt werden und nicht pro verlassener Sphäre die jeweilige Instanz. Demzufolge befinden sich LOD Modelle in der Szene, die nicht angezeigt werden dürfen und deshalb im Eindringverfahren auf unsichtbar gesetzt werden. Dies hat jedoch den Vorteil, dass das Modell bei einem erneuten Eindringen nicht mehr geladen, sondern nur noch für den Renderprozess freigeschaltet werden muss. Meistens hält sich der Benutzer innerhalb eines gewissen Bezirks auf, wodurch dieser schrittweise geladen wird. Hierdurch entsteht für einen definierten Raum ein größerer Bereich, in dem sich die virtuelle Kamera eingeschlossen der Sphären bewegen kann, ohne das eine einzige Ladeoperation für den Raum benötigt wird (Abbildung 5.5). Auf Kosten des Speicherplatzes reduziert man also zeitaufwändige Ladeoperationen und optimiert die Performance der Anwendung.

Nachdem beide Kollisionsereignisse geklärt sind, wird nun genauer auf den Ladeqeue eingegangen. Dieser vergleicht in jedem Frame die maximale Anzahl an parallelen Ladeoperationen mit der aktuellen Anzahl. Die Differenz n der beiden Werte gibt an, wie viele neue Ladeoperationen man ausführen darf. Als nächstes ruft der Queue für die ersten n IDs eine asynchrone Funktion auf. Hier wird der Objektdatenbankpfad mit der ID verknüpft und die entsprechende Datei in einem Nebenprozess geladen. Dadurch können alle anderen Prozesse weiterlaufen, ohne das sie sich gegenseitig behindern. Ist das Laden einer Datei beendet, wird sie nach Abhängigkeiten durchsucht und das Modell in der Szene instanziiert. Die benötigten Texturen und Materialien können aufgrund des Vorladens direkt dem Modell zugewiesen werden. Als letztes entfernt die asynchrone Funktion die verarbeitete Anfrage aus dem La-

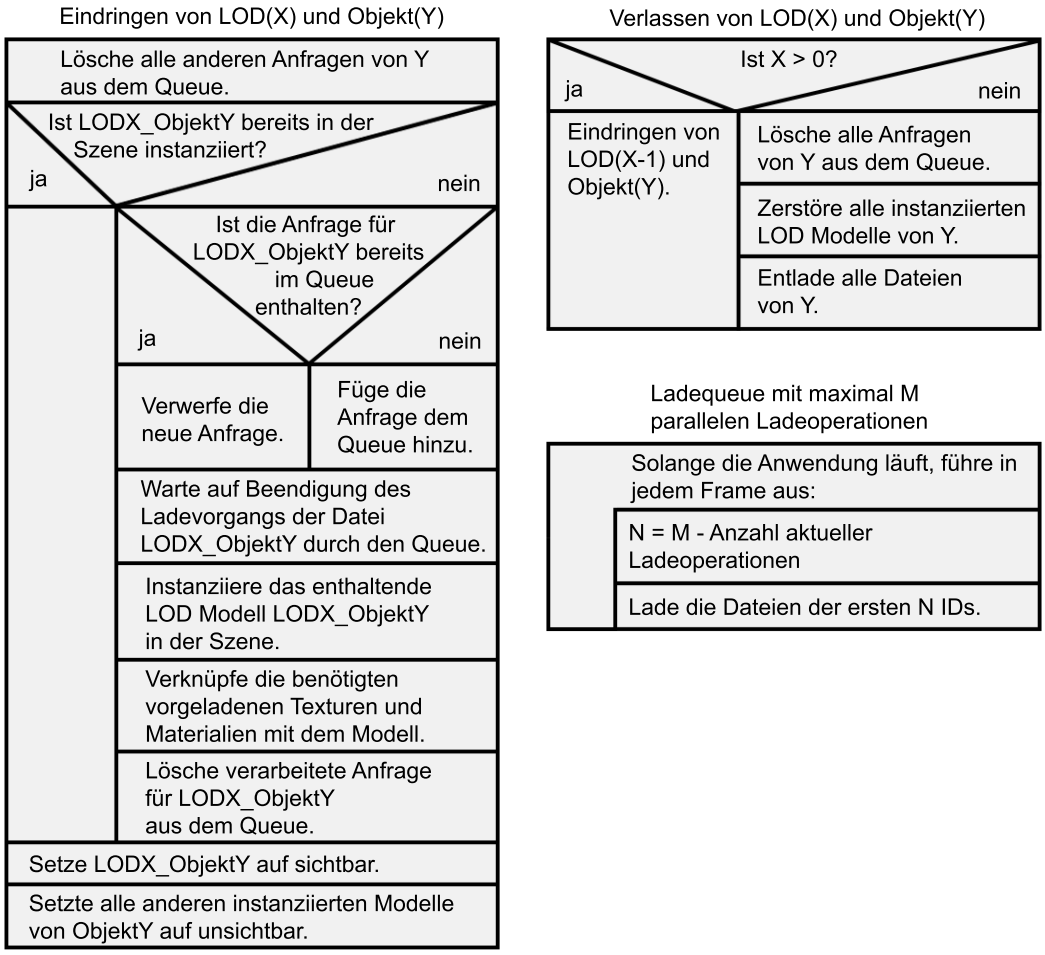

Abbildung 5.6: Zusammenfassung der wichtigsten Funktionen des Laufzeitprozesses.

dequeue, wodurch im nächsten Frame weitere Anfragen zum Laden aufrücken. Jetzt setzt man das LOD Modell auf sichtbar und alle anderen Repräsentationen des selben Objekts auf unsichtbar. Allgemein kann das Konzept des LOD und Szenenmanagementsystem mit dem ROAM-Verfahren verglichen werden [MD97]. Anstatt aber Split- und Mergeoperationen in Queues zu speichern, die ein Mesh abhängig vom Kamerastandpunkt schrittweise verfeinern, werden hier Lade- und Entladefunktionen durchgeführt, welche die gesamte Szene abhängig vom Kamerastandpunkt schrittweise anpassen. Abbildung 5.6 fasst noch einmal die wichtigsten Funktionen des Laufzeitprozesses zusammen.

5.3 Konzept und Implementierung der Informationsdatenbank

Der Abruf und die Verknüpfung von Informationen mit Objekten sind unabhängig vom Konzept des LOD und Szenenmanagementsystems. Weder die dafür erforderte Datenbank, noch

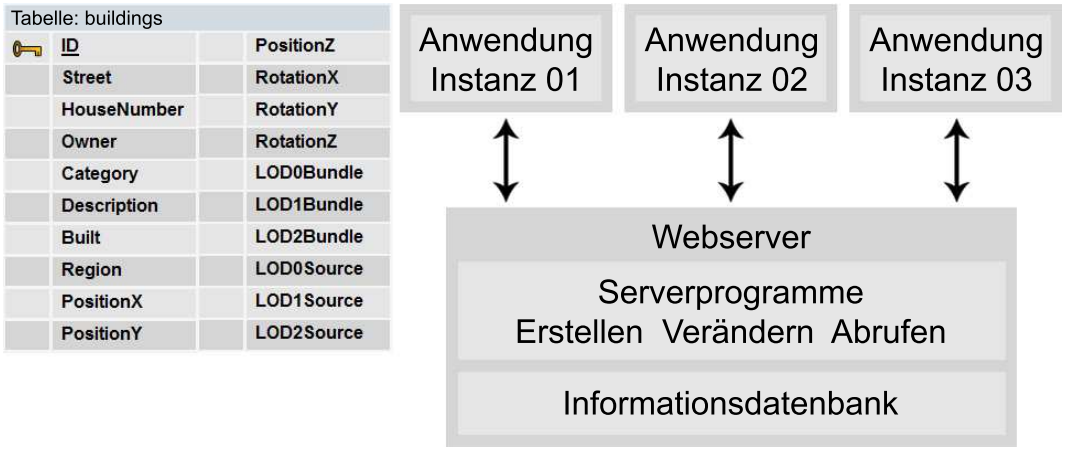

Abbildung 5.7: Aufbau der Informationsdatenbank. Links: Tabellenstruktur. Rechts: Schichtmodell des Datenbankzugriffs.

die darin enthaltenen Informationen nehmen am System teil, weshalb der Abschnitt dieses Ziel separat behandelt. Die Objektdatenbank, welche alle exportierten Modelle umfasst und die Grundlage für das LOD und Szenenmanagementsystem bildet, ist hierbei nicht mit der Informationsdatenbank zu verwechseln. Diese dient in der Anwendung als Wissensvermittlung und verbindet die einzelnen Objekte mit Hintergrundinformationen, wie zum Beispiel das Erbauungsdatum eines Gebäudes, der Gebäudetyp, Funktion, historische Informationen und eventuell Referenzen auf andere Artikel, Bilder oder Videos. Auch Verweise auf LOD Modelle in der Objektdatenbank werden hinterlegt, womit man beide Datenbanken verknüpft und für ein bestimmtes Objekt die entsprechenden Modelle finden kann. All diese Informationen werden jeweils einer ID eines Szenenobjekts zugewiesen. Nur eine Tabelle wird diesbezüglich vorausgesetzt. Abbildung 5.7 zeigt den Tabellenaufbau.

Die Datenbank soll für jeden Benutzer aktuell und zugleich aus der Anwendung heraus editierbar sein. Folglich wird eine zentrale SQL-Datenbank auf einem Webserver eingerichtet, auf die man mithilfe einer Schnittstelle zugreifen kann. Hierfür wird das Common Gateway Interface (CGI) verwendet, das die Kommunikation zwischen Programmen eines Webservers und dritter Software erlaubt [Vor03]. Die Programme des Webservers werden durch die Programmiersprache PHP realisiert und halten verschiedene Funktionen zum Erstellen, Verändern und Abrufen von Datensätzen bereit. Auf diese Weise können mehrere Instanzen der Anwendung gleichzeitig über die Serverprogramme auf die selbe Datenbank zugreifen. Dabei kann man sich das Verfahren wie ein Schichtmodell mit verschiedenen Zugriffsebenen vorstellen (Abbildung 5.7). Ist nun in der Anwendung die ID bekannt, für die eine bestimmte Aktion ausgeführt werden soll, so erstellt man eine HTTP-Anfrage, sendet sie an das entsprechende Programm und wartet auf Antwort. In der Anfrage ist mindestens die ID des Objekts plus zusätzliche Daten enthalten, welche für das Programm zum Erstellen oder Verändern von Inhalten gebraucht werden. Im Falle eines Informationsabrufs gibt das Programm die angeforderten Datensätze der ID an die Anwendung zurück, ansonsten erhält

man eine Auskunft über den Erfolg oder Misserfolg der Aktion.

5.4 Implementierung in Unity3D

Für die Umsetzung des Konzepts des LOD und Szenenmanagementsystems wurde die Game-Engine Unity3D genutzt. Der Einsatz dieser Game-Engine vereinfacht die Umsetzung erheblich, weil man auf viele nützliche Funktionen zurückgreifen und zugleich das System beliebig erweitern kann. Dabei wurde nicht nur die Anwendungslogik für das Konzept programmiert, sondern auch viele Funktionen, welche die Arbeitsabläufe automatisieren. Eines der Hauptprobleme waren die massigen Importe und Exporte, die zur Optimierung der Datenbasis und Generierung der Objektdatenbank führten. All diese Operationen automatisierte man mittels Skripte, wodurch der Arbeitsablauf des Projekts ebenfalls verbessert wurde.

5.4.1 Erstellung der Objektdatenbank

Der Arbeitsablauf startet mit dem Export der drei LOD Modelle und der Grundform jedes Objekts aus der CityEngine. Schon innerhalb dieser Software besitzen die Grundformen eine eindeutige ID. Beim Export der LOD Modelle wird die ID nach dem Prinzip $LODX\_haus\_ObjektY$ erweitert und als Dateiname benutzt. Durch den Gruppennamen $haus$ lassen sich beispielsweise beim Import in Unity3D andere Regeln auf zukünftige Objektgruppen anwenden. Als Exportregel wird in der CityEngine die automatische Zusammenfassung von Geometrien mit dem gleichen Material zu einem Mesh verwendet. Auf diese Weise werden so wenig Draw Calls wie nötig für ein Gebäudemodell aufgerufen. Das Verbinden von Meshes mit dem selben Material unter mehreren naheliegenden Gebäuden würde die Anzahl der Draw Calls und somit auch die Performance noch weiter optimieren. Allerdings ist dadurch der Austausch und die Bearbeitung einzelner LOD Modelle nicht mehr möglich. Die Grundformen stehen für das Objekt selbst und nicht für eine visuelle Repräsentation. Zudem werden sie in der Szene initial gesetzt und nicht einzeln nachgeladen. Folglich werden alle Grundformen zusammen in einer Datei exportiert, sodass jede Grundform ein separates Mesh mit dem Namen $ObjektY$ als ID bekommt. Die CityEngine exportiert außerdem alle benötigten Texturen mit. Nach dem kompletten Export sind für jedes Objekt drei FBX-Dateien mit den LOD Modellen, eine Datei für die Grundformen und alle Texturen vorhanden. In jeder einzelnen Modelldatei sind allerdings noch die Materialien mit einer Referenz auf die verwendete Textur gespeichert. Diese redundanten Daten gilt es spätestens beim Export aus der Game-Engine zu beseitigen.

Zuvor muss man jedoch alle Dateien nach Unity3D importieren. Dabei wird für jeden Dateiimport ein sogenannter FBX-Post/Preprocessor aufgerufen, der mithilfe eines Skripts überschrieben und angepasst werden kann. Hier setzt man den Skalierungsfaktor des Modells auf 1 und ändert das Verfahren für die Erstellung von Materialien. Sobald man ein Mesh nach Unity3D importiert, legt es automatisch ein Material für dieses an. Diese Generierung kann man so einstellen, dass immer nur ein Material pro Textur angelegt wird. Das heißt beim Import weiterer Meshes, deren Materialien die selbe Textur benutzen, wird

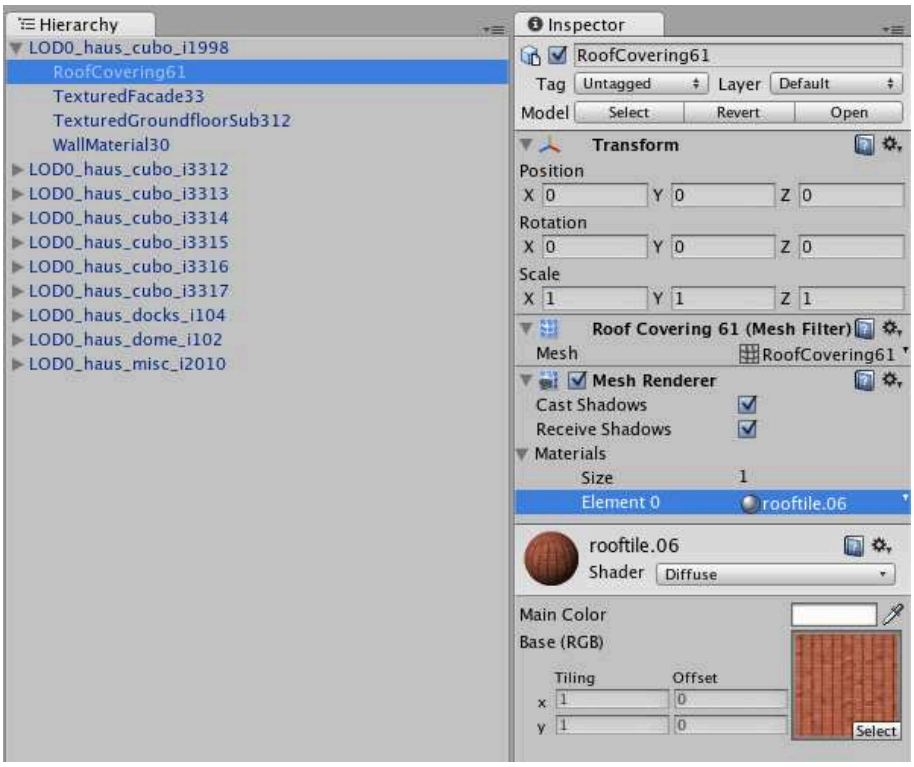

Abbildung 5.8: Zu sehen ist der Assetaufbau eines LOD Modells in Unity3D. Das Spielobjekt des gesamten Modells besitzt mehrere Kindobjekte, die jeweils ein Mesh repräsentieren. In der MeshRenderer-Komponente ist die Verbindung zum Material angegeben.

kein zusätzliches Material angelegt, sondern auf das bestehende zurückgegriffen. Dadurch entstehen genauso viele Materialien wie Texturen verwendet werden und alle Meshes greifen wieder auf den selben Material und Texturvorrat zu. Die Alternative dazu wäre für jedes Mesh ein eigenes Material zu generieren, ungeachtet dessen ob bereits eine Material mit den gleichen Eigenschaften existiert. Dies führt jedoch zu unerwünschter Redundanz. Desweiteren wird direkt beim Import eines LOD Modells eine Verbindung zur Informationsdatenbank aufgebaut und die ID des Objekts, zusammen mit der Position und Rotation sowie dem Pfad zur Quelldatei, vermerkt. Nach dem Import eines LOD Modells besteht das Asset aufgrund der Exportregeln der CityEngine aus mehreren Meshes, auf denen unterschiedliche Materialien liegen. Die Game-Engine erkennt automatisch, dass es sich um ein Mesh handelt und fügt jeweils die MeshRenderer-Komponente hinzu. Die Komponente übergibt das Mesh an den Renderprozess und verknüpft es mit einem Material (Abbildung 5.8). Jede Komponente eines Spielobjekts lässt sich per Skript abschalten, was am Beispiel des MeshRenderers das Abschalten der Sichtbarkeit bedeutet. Außerdem weist Unity3D generell jedem importierten Asset eine eigene, interne ID zu, die beim Export der Szeneninhalte eine wichtige Rolle spielt.

Für das Nachladen von Inhalten zur Laufzeit stellt die Software eine proprietäre Lösung

zur Verfügung. Sogenannte AssetBundles sind ein eigenes Dateiformat von Unity3D, in welche man beliebig viele Assets packen kann. Da vor dem Export der AssetBundles schon alle Assets in die Game-Engine importiert wurden, hat diese alle zeitintensiven Anpassungen an den internen Gebrauch vollzogen. Das bedeutet, beim Laden in die Anwendung müssen keine Umwandlungen mehr passieren und die Inhalte der AssetBundles können direkt verwendet werden. Zusätzlich komprimiert Unity3D die Dateien. Das Exportverfahren für die AssetBundles legt man durch ein Skript fest. Hierbei wurde besonders auf die Benutzerfreundlichkeit und höchste Automatisierung geachtet, denn der Export von AssetBundles ist auch im späteren Projektverlauf wichtig. Da einzelne Modelle nachträglich von anderen Projektmitglieder bearbeitet werden können, muss man die veränderten FBX-Dateien erneut als AssetBundles exportieren. Nicht die Quelldateien bilden die Objektdatenbank, auf welche die Anwendung zugreift, sondern die Sammlung an AssetBundles.

Eine weitere positive Eigenschaft von Unity3D kann man sich bei der Erstellung der Objektdatenbank zu Nutze machen. Alle Texturen, Materialien und Modelle besitzen eine von Unity3D intern vergebene ID. Zusätzlich erlaubt die Funktion zum Exportieren von AssetBundles das Speichern von Referenzen auf abhängige Dateien. Dabei bestehen die Referenzen aus den Unity-IDs der Assets. Lediglich um die korrekte Reihenfolge, in der die Assets exportiert werden, muss man sich kümmern, wobei unabhängige Assets immer zuerst exportiert werden müssen. Abbildung 5.1 zeigt, dass Modelle, bzw. Meshes, von Materialien und diese von Texturen abhängig sind. Man exportiert also zuerst alle Texturen, dann die Materialien und zum Schluss die LOD Modelle, welche mehrere Meshes umfassen. Zur Laufzeit muss man wiederum darauf achten, dass die Dateien in der selben Reihenfolge geladen werden, sonst können keine abhängigen Assets gefunden und zugewiesen werden. Der Dateiname für Texturen und Materialien ist aufgrund der Verwendung von Unity-IDs unbedeutend. Modell-Bundles werden jedoch mit dem bekannten Schema $LODX\_ObjektY$ exportiert, da kein anderes Asset auf ihre Unity-IDs verweist und dieses Namensmuster verständlicher ist als eine willkürliche Zahl. Möchte man nun eine beliebige Anzahl an Modellen in die Objektdatenbank exportieren, so wählt man die Assets in der Game-Engine aus und ruft das Exportskript auf. Dieses findet automatisch alle Abhängigkeiten und exportiert die Assets in der richtigen Reihenfolge. Hierfür speichert man alle ausgewählten LOD Modelle in eine Liste und untersucht dann die MeshRenderer-Komponenten der einzelnen Meshes nach den verwendeten Materialien. Diese legt man ebenfalls in einer Liste ab. Aus den Materialien holt man sich wiederum die darauf liegenden Texturen und speichert sie in einer dritten Liste ab. Jetzt müssen die einzelnen Assets in den Listen nur noch in der korrekten Reihenfolge als AssetBundles exportiert werden. Im letzten Schritt werden die Pfade des AssetBundles in die Informationsdatenbank bei der entsprechenden ID vermerkt. Am Ende aller Exports und Imports befinden sich die Szenenobjekte als separate AssetBundles in einem Dateipfad und die Objektdatenbank ist aufgebaut.

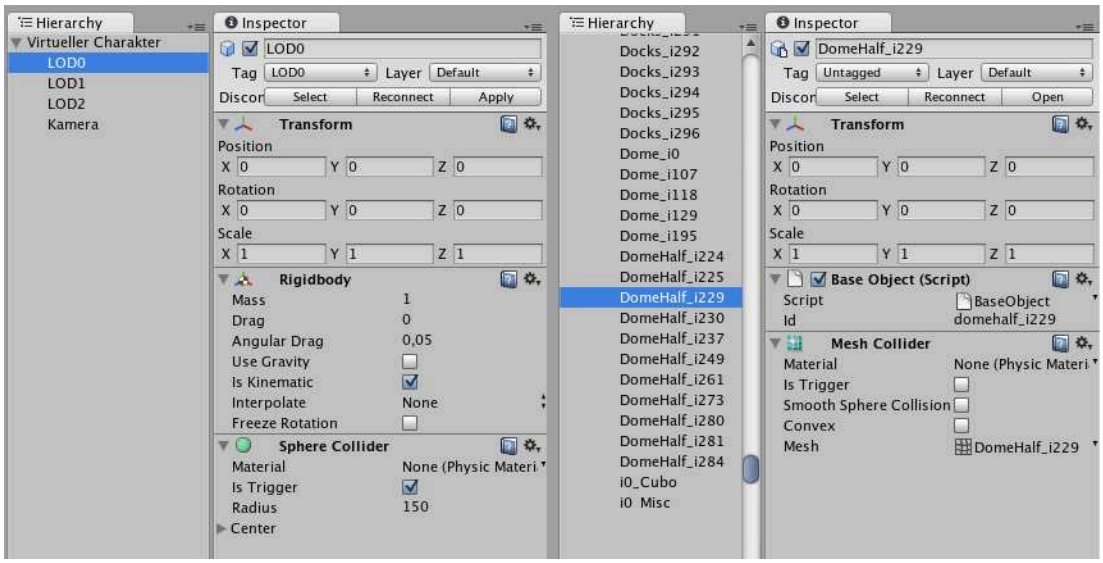

Abbildung 5.9: Links: Das Spielobjekt des virtuellen Charakters in Unity3D. Er umfasst weitere Spielobjekte als Kindknoten, darunter die Kamera und die drei Sphären. Rechts: Das Spielobjekt einer Grundform.

5.4.2 Szenenvorbereitung

Bei der Szenenvorbereitung setzt man zuerst das Gelände in die Szene. Hierfür wird das Unity-Terrainsystem verwendet, welches zur Generierung eine Höhenkarte und zur performanten Darstellung ein sichtabhängiges LOD benutzt. Die Höhenkarte kann aus der City-Engine übernommen und angepasst werden. Danach wird der virtuelle Charakter, welcher die Kamera beinhaltet, eingestellt. Unity3D bietet dafür ein vorgefertigtes Spielobjekt, das die Darstellung und Steuerung aus der typischen Erste-Person-Ansicht übernimmt. An dieses Spielobjekt werden nun drei Spielobjekte als Kindknoten mit jeweils einer SphereCollider-Komponente gehängt. Die Komponente repräsentiert eine unsichtbare Kugel mit benutzerdefinierten Radius als Kollisionsobjekt, übernimmt jedoch nicht automatisch die Berechnung der Kollision. Zu diesem Zweck ist die Rigidbody-Komponente gedacht, welche man ebenso an die drei Kindobjeke anfügt. Damit man in der Anwendung nicht automatisch stehen bleibt, wenn eine Sphäre mit einem anderen Objekt kollidiert, stellt man die SphereCollider-Komponenten als Trigger ein. Dadurch passiert zwar ein Kollisionsereignis, jedoch kann man in einem Skript selbst bestimmen, wie darauf reagiert wird. Die Rigidbody-Komponente sorgt auch dafür, dass sich das Objekt nach physikalischen Regeln verhält. Da sich die Sphären aber analog zur virtuellen Kamera bewegen sollen, werden die Rigidbody-Komponenten durch einen Schalter dementsprechend angepasst. Zusätzlich weist man den drei Kindobjekten, je nach Radius ihrer Sphären, die Tags $LOD0$, $LOD1$ und $LOD2$ zu, die hier im Sinne einer ID dienen. Somit kann bei einer Kollision überprüft werden, welche Sphäre das Ereignis verursacht hat. Alternativ könnte man auch den Namen der Spielobjekte dafür verwenden. Der Zugriff auf Tags per Skript gestaltet sich aber einfacher. Abbildung 5.9 zeigt noch einmal

alle vollständig eingestellten Spielobjekte. Der Nebeleffekt für die Kamera ist in Unity3D bereits enthalten und muss nur angeschaltet werden. Die Nebeldichte lässt sich so anpassen, dass Objekte hinter der größten Sphäre vollständig verschwinden.

Als nächstes können die Modelle der Grundformen in der Szene auf dem Terrain platziert werden. Da die Grundformen Meshes sind, besitzen sie die MeshRenderer-Komponente. Diese ist unnötig und wird gelöscht, weil die Grundformen nur als unsichtbare Kollisionsobjekte dienen sollen. Aus diesem Grund fügt man die MeshCollider-Komponente hinzu, welche die komplette Geometrie in die Kollisionsbestimmung einbezieht. Dringt nun zur Laufzeit der Anwendung eine Sphäre in eine Grundform ein, wird für das Spielobjekt der Grundform die Funktion $OnTriggerEnter(Collider\ other)$ ausgeführt, die es mit der eigenen Anwendungslogik zu überschreiben gilt. An dieser Stelle kommen geskriptete Komponenten ins Spiel, welche das Konzept des LOD und Szenenmanagementsystems umsetzen. Grundformen sind Basisobjekte, die durch drei LOD Modelle visuell repräsentiert werden. Die Modelle sind außerhalb der Anwendung in Dateien der Objektdatenbank gespeichert. Die Grundformen dürfen jedoch nicht sofort die Ladeoperationen ausführen, sondern nur Ladeanfragen an einen Queue stellen, der wiederum die Ladeoperationen von Dateien übernimmt. Das Umschalten der Sichtbarkeit der verschiedenen LOD Stufen und das Halten der Instanzen plus AssetBundles sowie das Entladen und Zerstören kann und wird jedoch vom Spielobjekt der Grundform selbst vorgenommen. Auf diese Weise verwaltet jedes Objekt sich selbst, während der Ladequeue alle Ladeoperationen organisiert. Dieses Verfahren setzt voraus, dass nachdem der Queue eine Datei geladen hat und das enthaltende Objekt instanziiert wurde, die Datei und die Instanz an das Basisobjekt zurückgibt. Für die Anwendungslogik des Basisobjekts wurde die BaseObject-Komponente implementiert, welche man ebenfalls an jedes Spielobjekt einer Grundform anfügt. Damit die BaseObject-Komponente direkt auf ihre Objekt-ID zugreifen kann, wird der Meshname der Grundform als ID-Attribut verwendet. Das fertige Spielobjekt einer Grundform ist in Abbildung 5.9 dargestellt.

Die Anwendungslogik des Queues muss sich ebenfalls in der Szene der Game-Engine befinden. Zu diesem Zweck wurde die AssetBundleManager-Komponente implementiert und an ein leeres Spielobjekt gehängt. Für die Komponente wurde das Singleton-Entwurfsmuster eingesetzt, da sie eine zentrale Anlaufstelle aller Basisobjekte ist, auf welche schnell und oft zugegriffen werden muss. Der AssetBundleManager enthält den Ladequeue sowie Funktionen, die den Zugriff auf diesen regeln, den Queue ständig aktualisieren und die Ladeoperation selbst. Die maximale Anzahl der parallelen Ladeoperationen und der Pfad zur Objektdatenbank lässt sich beliebig in den Eigenschaften der Komponente editieren. Hinsichtlich der Verbindung zur Informationsdatenbank setzte man die InfoManager-Komponente um, welche ebenso an ein leeres Spielobjekt in der Szene angefügt wird. Sie stellt Funkionen bereit mit denen sich ein Objekt anwählen lässt und man mittels HTTP-Anfragen an ein Webserverprogramm verschiedene Informationen über das Objekt erhält oder verändern kann. Um die Generierung einer Datenstruktur für alle Kollisionsobjekte braucht man sich nicht zu kümmern. Dieser Prozess wird automatisch vom Physiksystem der Game-Engine ausgeführt.

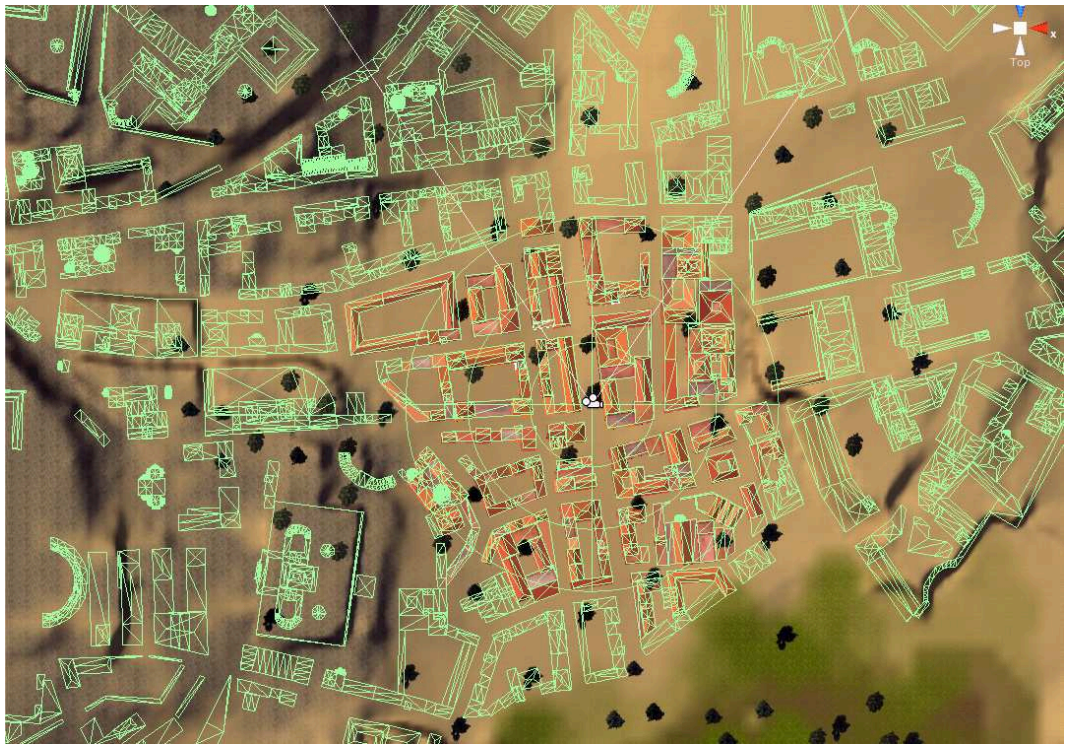

Abbildung 5.10: Die Vogelperspektive auf die Szene zur Laufzeit der Anwendung. Im Bereich der Sphären werden die passenden LOD Modelle geladen und angezeigt. Modelle außerhalb der Sphären werden weder angezeigt noch sind sie geladen.

5.4.3 Laufzeitprozess

Startet man die Anwendung, wird für alle Komponenten die Startmethode aufgerufen. Hier lädt der AssetBundleManager zuerst alle AssetBundles der Texturen und darauf der Materialien, welche sich im angegebenen Pfad der Objektdatenbank befinden. Dringt nun eine Sphäre in eine Grundform ein, so wird in der BaseObject-Komponente des Spielobjekts die Funktion $OnTriggerEnter(Collider\ other)$ aufgerufen, wobei der Parameter $Collider\ other$ für das andere Kollisionsobjekt steht. In diesem Fall ist das eine der drei Sphären. Um herauszufinden welche Sphäre eingedrungen ist, überprüft man den Tag des Kollisionsobjekts $other$. Danach wird die Methode $updateBaseObject(int\ index)$ ausgeführt und als Parameter der Sphärenindex übergeben. Diese aktualisiert den AssetBundleManager und das Basisobjekt bezüglich des aktuellen Index. Als erstes werden alle anderen Anfragen des Basisobjekts mithilfe der Funktion $removeRequest(LoadRequest\ lr)$ aus dem Queue gelöscht. Darauf prüft man, ob die BaseObject-Komponente das LOD Modell des Basisobjekts mit dem aktuellen Index bereits hält. Ist dies der Fall, so bewirkt die Methode $showInstance(int\ index)$ das Anschalten der MeshRenderer-Komponente des instanziierten Spielobjekts. Außerdem wird die Sichtbarkeit anderer instanziierter LOD Modelle abgeschaltet (Abbildung 5.10). Falls das benötigte Modell nicht vorhanden ist, stellt man mit

der Funktion $addRequest(LoadRequest\ lr)$ eine Ladeanfrage an den AssetBundleManager. Eine Ladeanfrage ist eine Struktur, welche den Namen des AssetBundles, den aktuellen Index des Basisobjekts und eine Referenz auf die BaseObject-Komponente selbst hält. Der AssetBundleManager testet, ob der Queue die Anfrage schon enthält und fügt sie diesbezüglich hinzu.

In jedem Frame wird für alle Spielobjekte die Methode $Update()$ aufgerufen. Die Komponente des AssetBundleManagers überschreibt diese und berechnet die Anzahl der nächsten möglichen Ladeoperationen. Für die nächsten Ladeanfragen im Queue führt man dann die asynchrone Funktion $LoadAssetBundle(LoadRequest\ lr)$ mit der jeweiligen Ladeanfrage als Parameter aus. Dabei wird der Name des benötigten AssetBundles mit dem angegeben Dateipfad der Objektdatenbank verbunden. Es entsteht der Pfad zur Datei, welche im nächsten Schritt parallel zum Hauptprozess geladen wird. Unity3D stellt für das Laden von AssetBundles eine eigene Funktion zur Verfügung, der nur noch der Dateipfad übergeben werden muss. Ist das AssetBundle vollständig geladen, so instanziiert man das enthaltene Spielobjekt und gibt dieses zusammen mit dem AssetBundle an das Basisobjekt zurück, welches die entsprechende Anfrage gestellt hatte. Die dafür zuständigen Funktionen $setInstance(Spielobjekt\ so)$ und $setBundle(AssetBundle\ ab)$ bietet die BaseObject-Komponente. Um das Verknüpfen der Texturen, Materialien und Meshes muss man sich hier nicht kümmern. Durch die internen IDs erledigt Unity3D diese Aufgabe automatisch. Als letztes wird die bearbeitete Anfrage aus dem Queue gelöscht. Ab der Rückgabe des AssetBundles und der Instanz, führt die BaseObject-Komponente wieder alle Aktionen aus. Diese setzt die aktuelle Instanz auf sichtbar und andere vorhandene Repräsentationen auf unsichtbar.

Verlässt eine Sphäre eine Grundform, so wird für das Spielobjekt der Grundform die Methode $OnTriggerExit(Collider\ other)$ aufgerufen, die man ebenfalls in der BaseObject-Komponente überschreibt. Es wird der Sphärenindex ermittelt und danach die Funktion $updateBaseObject(int\ index)$ ausgeführt. Als Parameter übergibt man den um Eins gesenkten Sphärenindex, solange dieser größer als Null ist. Falls der Index gleich Null ist und somit die größte Sphäre die Grundform verlassen hat, so zerstört eine andere Funktion $cleanUp()$ alle Instanzen und entlädt die passenden AssetBundles, deren Referenzen in der BaseObject-Komponente gespeichert sind. Abbildung 5.11 zeigt in einem Sequenzdiagramm noch einmal die Interaktion der BaseObject- und AssetBundleManager-Komponente zur Laufzeit.

Die InfoManager-Komponente erlaubt es dem Anwender zur Laufzeit die Daten in der Informationsdatenbank über ein bestimmtes Objekt abzurufen und zu verändern. Zu diesem Zweck lenkt man die Kamera in die Richtung eines Objekts und drückt die linke Maustaste. Die Komponente fängt den Mausdruck ab und schießt einen Strahl von der Kamera aus durch die Mitte des Bildes in die Szene. Trifft der Strahl auf ein Kollisionsobjekt, holt man sich die ID der getroffenen Grundform. Es wird eine HTTP-Anfrage erstellt und die ID mittels der CGI-Schnittstelle an das Webserverprogramm für den Abruf von Informationen geschickt. Das Programm holt sich aus der Datenbank alle Informationen des Objekts und

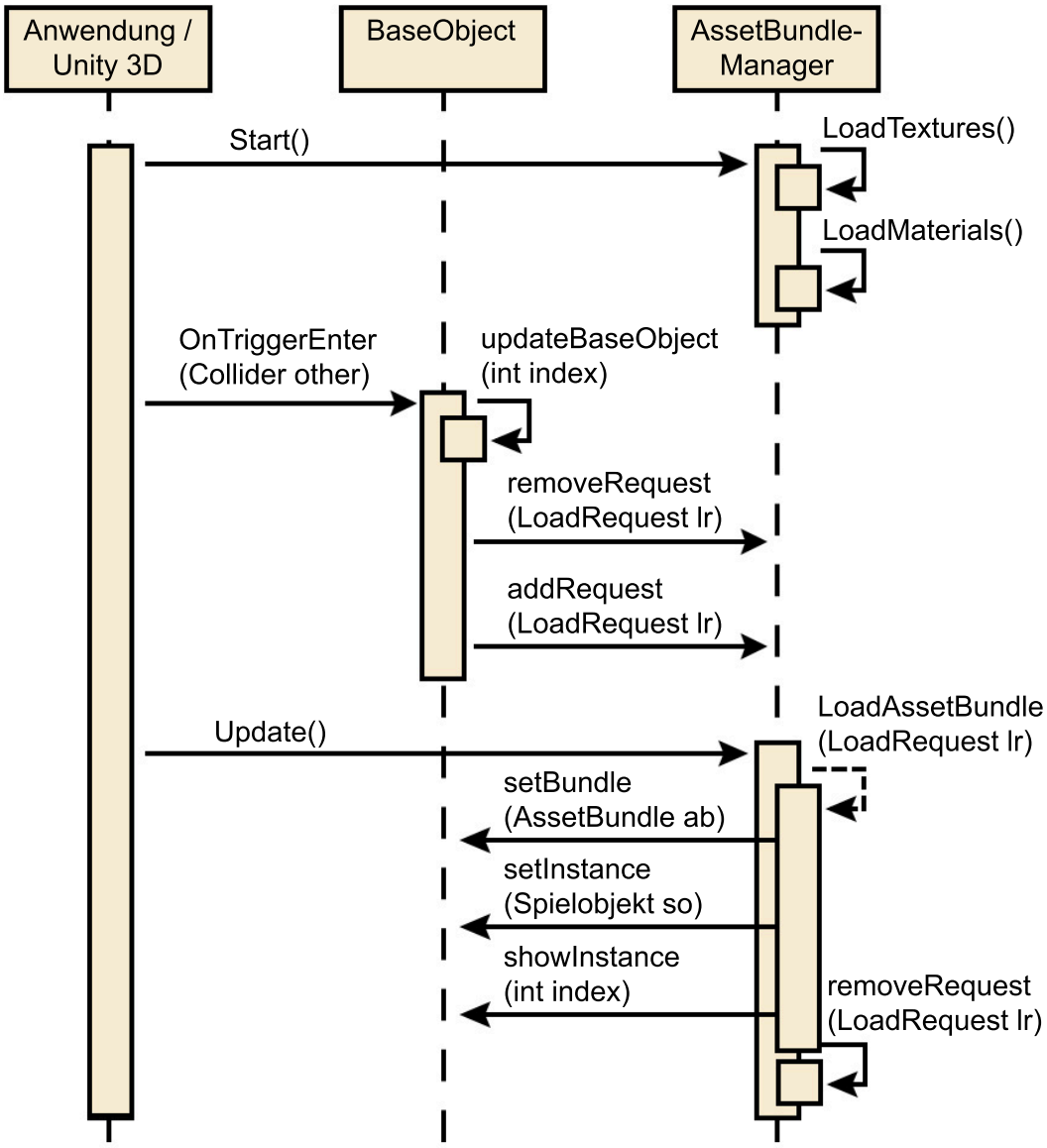

Abbildung 5.11: Das Sequenzdiagramm stellt den Aktionsverlauf für das Eindringen einer Sphäre in eine Grundform dar. Das benötigte LOD Modell ist in diesem Fall weder geladen noch instanziiert. Durchgezogene Pfeile stehen für synchrone Operationsaufrufe und gestrichelte Pfeile für asynchrone Operationsaufrufe.

schickt sie an die Anwendung zurück. Dort öffnet sich eine Oberfläche, in der die Daten angesehen und verändert werden können. Hier ist es auch möglich die Veränderungen wieder in die Datenbank zu speichern. Dabei wird erneut eine Anfrage mit den editierten Daten an das Programm zum Verändern von Informationen gesendet, welches sich dann um den Rest kümmert.

5.5 Zusammenfassung

Dieses Kapitel ging auf das Konzept und die Umsetzung des LOD und Szenenmanagementverfahrens sowie auf die Datenbank ein, welche die einzelnen Szenenobjekte mit Hintergrundinformationen verbindet. Ersteres kann in die drei Abschnitte Erstellung der Objektdatenbank, Szenenvorbereitung und Laufzeitprozess unterteilt werden. Die ersten zwei Abschnitte werden einmalig ausgeführt, bevor die Anwendung startet und bilden die Basis für das Laufzeitverfahren. Im Schritt der Objektdatenbankerstellung werden alle LOD Modelle mit einer eindeutigen ID in separaten Datei gespeichert. Dabei entnimmt man ihnen aufgrund der Redundanz alle Daten über Materialien und Texturen, für welche man ebenfalls einzelne Dateien mit eindeutiger ID anlegt. Die IDs der Materialien und Texturen werden dann im LOD Modell hinterlegt, um zusammengehörige Daten zur Laufzeit wieder zu verbinden. Innerhalb der Szenenvorbereitung hängt man drei Sphären mit unterschliedlichen Radien um die Kamera, welche sich analog mit ihr bewegen. Sie geben die Bereiche an, in denen die LOD Stufen geladen und angezeigt werden und dienen als Kollisionsobjekte. Für die Kamera wird ein Nebeleffekt gesetzt, der Objekte hinter der größten Sphäre vollständig verdeckt. Zuletzt werden die Grundformen ebenfalls als Kollisionsobjekte in der Szene platziert und angesichts einer kürzeren Rechenzeit eine hierarchische Datenstruktur für alle Kollisionsobjekte festgelegt.

Die Grundidee des Laufzeitverfahrens ist das Laden und Anzeigen des entsprechenden LOD Modells in der Objektdatenbank bei der Kollision einer Sphäre mit einer Grundform. Sobald eine Kollision stattfindet, wird eine Ladeanfrage mit der ID des zu ladenden LOD Modells an einen Queue geschickt. Dieser verarbeitet und verwaltet alle Anfragen bezüglich der aktuellen Umstände, wodurch unnötige Ladeoperationen verhindert werden. Die bedeutende Eigenschaft des Ladequeues ist, dass er eine benutzerdefinierte Menge an parallen Ladeoperationen ausführen kann, wodurch interaktive Frameraten erhalten bleiben. Ist ein Ladevorgang beendet, wird das enthaltende Modell in der Szene instanziiert, mit seinen vorgeladenen Materialien und Texturen verknüpft und für den Renderprozess freigegeben. Alle anderen bereits instanziierten LOD Modelle des selben Objekts setzt man auf unsichtbar. Verlässt die letzte Sphäre eine Grundform, so werden alle instanziierten Repräsentationen des Objekts zerstört und die Dateien entladen. Mithilfe des Queues werden die dringlichsten Modelle zuerst geladen und das Stadtmodell abhängig vom Kamerastandpunkt schrittweise an die aktuelle Situation angepasst. Außerdem wird das Szenenmanagementsystem mit dem LOD System verbunden, indem man nur jeweils das aktuell ausreichende LOD Modell eines Objekts lädt und anzeigt. Durch dieses Verfahren reduziert man die Rechenzeit zur Echt-

zeitdarstellung einer Szene mit zahlreichen Geometriemodellen sowie den dafür benötigten Speicherplatz. Umgesetzt wurde das Konzept mit der Game-Engine Unity3D. Die Software bietet für das Auslagern von Szeneninhalten und das dynamische Laden zur Laufzeit sogenannte AssetBundles an. Die Kollisionsberechnungen finden im Physiksystem der Middleware statt und die Anwendungslogiken bezüglich des LOD und Szenenmanagementsystems implementierte man durch eigens geskriptete Komponenten.

Für das Hinterlegen von Daten zu einzelnen Objekten wird eine zentrale Datenbank auf einem Webserver angelegt, die in der Anwendung als Wissenvermittlung dient. Dabei vernetzt eine Tabelle die ID eines Objekts mit verschiedenen Hintergrundinformationen und Verweisen auf die betroffenen Dateien in der Datenbasis. Den Zugriff auf die Datenbank, hinsichtlich des Abrufs, der Erstellung und Veränderung von Datensätzen, steuern Programme auf dem Webserver. Die Webserverprogramme können dann von mehreren Instanzen der Anwendung gleichzeitig aufgerufen werden. Zur Parameterübergabe wird die CGI-Schnittstelle verwendet.

Nachdem das Konzept und die Umsetzung des LOD und Szenenmanagementsystems erläutert wurde, geht das nächste Kapitel auf verschiedene Systemtests und die Ergebnisse ein. Die Eigenschaften des Verfahrens werden geändert und das System auf Performance, Speicherauslastung und Qualität geprüft.

Kapitel 6

Evaluation

In diesem Kapitel wird das entwickelte LOD und Szenenmanagementsystem evaluiert und die Ergebnisse vorgestellt. Dabei werden die verschiedenen Eigenschaftswerte des Verfahrens verändert, um die beste Einstellung für eine bestimmte Computerkonfiguration zu finden. Die Tests werden nicht mit dem internen Unity-Profiler ausgewertet, da die Ergebnisse aufgrund der zusätzlichen Rechenlast der Game-Engine stark vom realen Wert abweichen können. Stattdessen wird für jeden Versuch ein ausführbares Programm erstellt und die zu bestimmenden Werte mithilfe des Task-Managers und eigenen Skripten gemessen.

6.1 Entwicklungstests

Während der Konzeptentwicklung des LOD und Szenenmanagementsystems wurden ebenfalls Versuche durchgeführt, die zur weiteren Optimierung des Verfahrens beitrugen. Bevor im System die Sphären zur Kollisionsbestimmung dienten, kamen andere Geometrien zum Einsatz, welche sich auf den Sichtbereich der virtuellen Kamera beschränkten. Diese Kollisionsobjekte teilten das Kamerafrustum in drei Bereiche auf, wodurch nur für sichtbare Objekte Ladeanfragen ausgelöst wurden (Abbildung 6.1). Die Sphären hingegen spannen einen Raum auf, der größer ist als das Kamerafrustum selbst. Das heißt, es werden unsichtbare Modelle geladen, welche für die aktuelle Situation keine Bedeutung haben. Für den Renderprozess spielen die zusätzlichen Modelle keine Rolle, da sie ohnehin durch das View Frustum Culling ignoriert werden. Doch durch die zunächst überflüssig erscheinenden Ladeoperationen geht vor allem Speicherplatz und Performance verloren. Aus diesem Grund verwendete man zunächst Kollisionsobjekte, die dem Kamerafrustum entsprachen. Zur Laufzeit zeigte sich jedoch ein Problem. Aufgrund des eingeschränkten Sichtbereiches fielen schon bei kleinen Rotationen der Kamera viele Ladeoperationen an (Abbildung 6.1). Dazu kamen noch die Ladeanfragen, welche durch eine Translation des virtuellen Charakters ausgelöst wurden. Außerdem werden Ladeanfragen durch das Kamerafrustum selbst initiiert. Folglich wird ein Modell erst geladen, wenn es bereits sichtbar sein sollte. Durch die vielen Ladeanfragen und der Ladedauer ist es jedoch nicht möglich alle benötigten Dateien innerhalb des nächsten Frames zu laden. Dies führt zu einer deutlich sichtbar verzögerten Darstellung von Objekten. Mit der Verwendung einer Sphäre lösen Rotationen keine Ladeanfragen aus. Dies entspricht

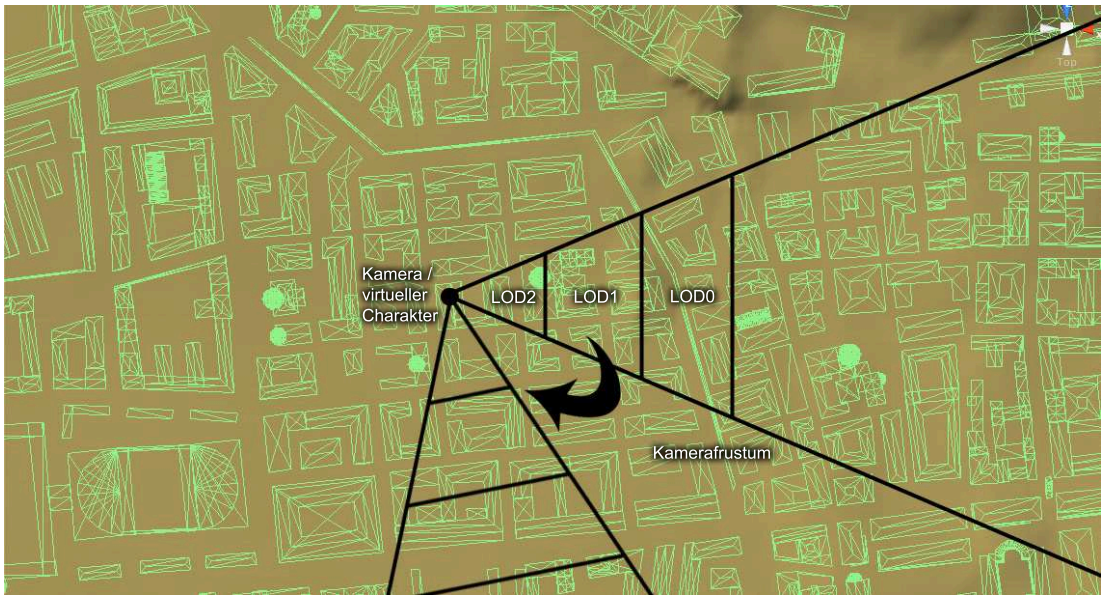

Abbildung 6.1: Hier wird anstatt der Sphären das Kamerafrustum als Grenzbereich für die LOD Modelle verwendet. Bei einer Rotation der Kamera, kollidiert das Frustum jedoch mit zu vielen Grundformen, deren Modelle nicht bis zum nächsten Frame geladen werden können, was eine verzögerte Darstellung zur Folge hat.

auch dem Sinn eines LODs, denn lediglich Translationen verändern die Distanz und somit auch die nötige Detailstufe eines Objekts. Auf diese Weise lädt man zwar für das aktuelle Frame unnötige Modelle, doch es wird für Situationen in naher Zukunft vorgesorgt.

6.2 Versuchsaufbau

Nach der Umsetzung des Konzepts mit der Game-Engine Unity3D soll zuerst der Gewinn für die Anwendung durch das LOD und Szenenmanagementsystems überprüft werden. Dafür vergleicht man die Echtzeitdarstellung des Stadtmodells mit und ohne Optimierungen. Danach werden verschiedene Eigenschaftswerte des entwickelten Verfahrens verändert und die Auswirkungen auf die Anwendung beobachtet. Alle nachfolgenden Tests wurden mit einer Intel Core Duo CPU T7300 mit 2.0 GHz pro Prozessorkern und 2 GB adressierbaren Arbeitsspeicher auf dem Betriebssystem Windows Vista 32-Bit durchgeführt. Als Grafikkarte wurde eine ATI Mobility Radeon HD 2400 XT mit einem Grafikspeicher von 128 MB verwendet. Für die Grafikeinstellungen benutzte man eine Auflösung von 1024 auf 768 Pixel und das von Unity3D vorgegebene Profil mit guter Grafikqualität, welches unter anderem Pixel Lighting und Soft Shadows einschaltet. Um die verschiedenen Einstellung vergleichen zu können, werden die Kriterien durchschnittliche Bildrate pro Sekunde (FPS = Frames per second), durchschnittlicher Arbeitsspeicher und Bildqualität bestimmt. Hierbei wird eine Anwendung

nicht mehr als interaktiv eingestuft, wenn die durchschnittlichen FPS unter 25 Hz sinken. Die Bildqualität wird zwecks einer besseren Vergleichbarkeit durch ein rechnerisches Verfahren festgelegt, bei dem der Bereich für bessere LOD Modelle einen höheren Faktor bekommt als der Bereich gröberer Modelle:

$$Bildqualität = 3 \cdot Radius^2_{LOD2} + 2 \cdot (Radius^2_{LOD1} - Radius^2_{LOD2}) + 1 \cdot (Radius^2_{LOD0} - Radius^2_{LOD1})$$

Die rein optische Qualitätswahrnehmung verhält sich analog zum berechneten Wert. Allgemein nimmt die Bildqualität zu, umso weiter man in die Szene hineinschauen kann und je mehr hochauflösende Modelle zu sehen sind. Alle Versuchsaufbauten, mit welchen die Vergleichswerte gemessen werden, bestehen aus einem ausführbaren Programm, in dem sich die virtuelle Kamera immer am selben Startpunkt in der Szene befindet. Ist das Programm fertig geladen, wird die Kamera bis zu einem festgelegten Punkt in eine Richtung bewegt und darauf der Test beendet. Somit gelten für alle Versuche die gleichen äußeren Umstände. Start- und Endpunkt werden in einen Stadtbezirk gelegt, der im Vergleich zu anderen Regionen durchschnittlich mehr Objekte enthält. Demzufolge wird die Anwendung für einen der leistungs- und speicherintensivsten Bereiche getestet.

6.3 Versuchsdurchführung und Ergebnisse

Im ersten Versuch wird auf alle entwickelten Optimierungen verzichtet und die Echtzeitszene solange mit LOD2 Modellen gefüllt bis die Bedingung der Interaktivität nicht mehr erfüllt ist. Hier wird schnell die Grenze der Darstellung von zahlreichen Objekten deutlich. Ab ca. 200 LOD2 Modellen sinken die FPS unter 25 Hz. Der lange Renderprozess verschuldet den Abbruch und nicht der benötigte Arbeitspeicher von 900 MB. Eine optisch gute Bildqualität ist nur innerhalb des ausmodellierten Bereichs gegeben. Verlässt man diesen, fällt die Qualität aufgrund des fehlenden Szenenmanagements auf 0. Hier wird nicht die oben genannte Formel verwendet, da gar keine Objekte zu sehen sind. Für den zweiten Versuch wird deshalb nur das Szenenmanagementsystem aktiviert, indem man die Sphären $LOD0$ plus $LOD1$ ausschaltet und den Radius der Sphäre $LOD2$ variiert. Abbildung 6.2 zeigt die Ergebnisse der Tests. Im Gegensatz zum ersten Versuch ist jetzt die Bildqualität über die ganze Szene gegeben und nicht nur in einem Teilbereich. Der Einsatz des Szenemanagementsystems macht die Darstellung einer Szene mit zahlreichen Modellen ohne Ladeunterbrechungen erst möglich. Dabei ist ein sichtbarer Bereich mit dem Radius von 50 Einheiten und einer Framerate von 29 Hz möglich.

Die nächsten Versuche sollen die Performance, Speicherauslastung und Bildqualität des kompletten LOD und Szenenmanagementsystems überprüfen. Von vier Eigenschaften hängen die genannten Kriterien ab. Bezüglich des Systems sind davon zwei intern und zwei extern. Zu den internen Eigenschaften gehören die drei Sphärenradien sowie die maximale Anzahl paralleler Ladeoperationen (MAPL), welche sich beide im System durch den Benutzer

1.) Versuch: keine Optimierungen

LOD2 Modelle	Arbeitsspeicher	FPS	Bildqualität	Bereich
200	900 MB	< 25 Hz	ca. 10.000	ausmodelliert
200	900 MB	184 Hz	0	leer

2.) Versuch: mit Szenenmanagementsystem

Radius LOD2	Arbeitsspeicher	FPS	Bildqualität
20	505 MB	48 Hz	1.200
30	535 MB	40 Hz	2.700
40	575 MB	35 Hz	4.800
50	602 MB	29 Hz	7.500
60	655 MB	23 Hz	10.800

3.) Versuch: mit Optimierungen (LOD und Szenenmanagementsystem)

Radius LOD2/1/0	Arbeitsspeicher	FPS	Bildqualität
20/40/60	625 MB	40 Hz	5.600
30/60/90	680 MB	34 Hz	12.600
40/80/120	780 MB	25 Hz	22.400
50/100/150	970 MB	20 Hz	30.000

4.) Versuch: mit Optimierungen, Radien LOD2/1/0 = 30/60/90, Bildqualität = 12.600

MAPL	Arbeitsspeicher	FPS	Ladeoperationen	Ladeanfragen
100	690 MB	33 Hz	5	0
50	690 MB	33 Hz	5	0
20	690 MB	33 Hz	5	0
10	690 MB	33 Hz	5	0
5	690 MB	35 Hz	5	2
3	660 MB	37 Hz	3	40
1	600 MB	41 Hz	1	60

5.) Versuch: mit Optimierungen, Radien LOD2/1/0 = 30/60/90, Bildqualität = 12.600

Laufgeschw.	Arbeitsspeicher	FPS	Ladeoperationen
8	690 MB	33 Hz	5
12	690 MB	31 Hz	8
16	690 MB	29 Hz	12

Abbildung 6.2: Testergebnisse des LOD und Szenenmanagementsystems.

anpassen lassen. Die externen Eigenschaften sind die Laufgeschwindigkeit des virtuellen Charakters und die Objektdichte. Die Objektdichte gibt an wie viele Objekte sich innerhalb eines bestimmten Raumes befinden. Sie variiert in unterschiedlichen Bereichen und lässt sich nicht verändern, da sie von der Szene selbst vorgegeben wird. Ist die Objektdichte größer, müssen mehr Ladeoperationen pro Frame ausgeführt werden, was die Performance verschlechtert. Das gleiche Prinzip gilt für die Charaktergeschwindigkeit. Bewegt man sich schneller fort, so löst dies automatisch mehr Ladeanfragen aus, die bearbeitet werden müssen. Um einen natürlichen Eindruck zu gewährleisten, ist der Standardwert der Laufgeschwindigkeit für die folgenden Versuche gleich 8 Einheiten pro Sekunde (E/s). Der Standardwert für die MAPL ist 100. Im Bezug auf die Objektdichte des Stadtmodells ist dies ein hoher Wert, wodurch der Queue alle Ladeanfragen sofort bearbeitet.

Im dritten Versuch werden nun die Sphärenradien verändert. Man stellt fest, dass bei einer interaktiven Framerate von 25 Hz eine Bildqualität von 22.400 erreicht wird. Vergleicht man diese Werte mit dem letzten Ergebnis aus dem zweiten Versuch (vergleichbarster Wert: FPS = 23 Hz), lässt sich erkennen, dass mithilfe des LOD Systems bei gleicher Anzahl der FPS eine Steigerung der Bildqualität von über 107% möglich ist. Dies geht jedoch auf die Kosten des Speicherplatzes, da für viele Objekte mehrere LOD Modelle geladen sind (Abbildung 6.2). Im vierten Versuch werden die MAPL variiert und zusätzlich die durchschnittliche Anzahl an aktuellen Ladeoperationen sowie wartenden Ladeanfragen pro Frame gemessen. Hier ist zu sehen, dass sich bei sinkender MAPL keine Veränderungen zeigen bis die MAPL den Wert der durchschnittlichen Ladeoperationen pro Frame erreicht hat. Ab dann müssen Ladeanfragen auf ihre Ausführung warten, was eine steigende Anzahl an FPS zur Folge hat. Der benötigte Arbeitsspeicher sinkt, aber auch die Bildqualität kann um einen nicht messbaren Wert abfallen, da Modelle erst verspätet angezeigt werden. Findet man den optimalen Wert für die MAPL (hier MAPL=5), verbessern sich die FPS nochmals um ca. 3%. Dieser Wert hängt allerdings von der Objektdichte der Region ab, welche sich innerhalb der Szene verändert. Aus diesem Grund wird ein Bereich mit der höchsten Objektdichte getestet.

Zuletzt wird die Auswirkung der Laufgeschwindigkeit auf die Anwendung geprüft. Abbildung 6.2 zeigt die Ergebnisse bei unterschiedlichen Geschwindigkeiten. Der Test beweist, dass eine höhere Laufgeschwindigkeit aufgrund von mehr Ladeoperationen zu einer schlechteren Performance führt. Dabei hat die Laufgeschwindigkeit mit 8 E/s den natürlichsten Laufeindruck hinterlassen. Hinsichtlich des Testcomputers sind folgende optimale Einstellungen unter Berücksichtigung der Interaktivitätsbedingung gewählt worden: Radien = 40/80/120, MAPL = 5, Laufgeschwindigkeit = 8. Mit diesen Einstellungen erreicht man durchschnittlich 26 FPS und eine Speicherauslastung von 790 MB.

Ein Problem besteht allerdings über alle Versuche hinweg. Der Effekt des Poppings wird unabhängig von den Sphärenradien vom Benutzer wahrgenommen. Dies ist auf die Modellbasis zurückzuführen, deren prozedurale Generierung nicht optimal ist und die Anpassungen der Erstellungsregeln nicht im Rahmen dieser Arbeit enthalten sind. Der visuelle Unterschied zwischen zwei aufeinanderfolgenden LOD Modellen ist oft so groß, dass auch bei weiten Distanzen der LOD Austausch zu bemerken ist. Die Positionen und Größen der Fen-

Abbildung 6.3: Fehler bei der LOD Modellerstellung. Oben: eine Hausreihe von LOD0 Modellen. Unten: die LOD1 Modelle der selben Hausreihe.

ster stimmen nie überein, die Farben und Texturen ändern sich teilweise vollständig und auch das hinzufügen großer Geometrien wie eines Balkons ist deutlich sichtbar (Abbildung 6.3). Diese Problematik ist jedoch nicht mit dem LOD und Szenenmanagementsystem verbunden und lässt sich dadurch auch nicht lösen.

Der Vergleich mit anderen existierenden LOD oder Szenenmanagementsystem ist schwierig, da die Verfahren nicht veröffentlicht werden. Eines der neusten Computerspiele, welches sich in einer großen Stadt mit hoher Grafikqualität abspielt, ist Grand Theft Auto 4[1]. Dort kommen vergleichbare Systeme zum Einsatz, welche interaktive Frameraten garantieren. Die Anforderungen an einen Computer sind entsprechend hoch. Es wird ein Arbeitsspeicher von 2 GB, 18 GB Fesplattenspeicher und eine Grafikkarte mit einem 512 MB großen Grafikspeicher verlangt. Die Anforderungen würde der verwendete Testcomputer nicht bestehen. In einem Spiel laufen andererseits mehr Prozesse ab, als nur die Darstellung der Szene, was einen direkten Vergleich unmöglich macht.

6.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass das entwickelte Szenenmanagementsystem die Darstellung einer großen Szene mit vielen Objekten erst möglich macht. In Verbindung mit dem

[1] http://www.rockstargames.com/IV/

LOD System lässt sich bei gleicher Anzahl der FPS eine Steigerung der Bildqualität von über 107% realisieren. Aufgrund deutlicher visueller Unterschiede der LOD Modelle eines Objekts ist allerdings bei jeder Distanz der LOD Austausch zu erkennen. Das letzte Kapitel greift noch einmal die wichtigsten Inhalte der Arbeit auf und gibt einen Ausblick auf mögliche Verbesserungen.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wird auf der Grundlage eines Stadtmodells mit zahlreichen Objekten ein „Level of Detail“ mit einem Szenenmanagementsystem verknüpft, um erstens die Darstellung der vollständigen Szene ohne Ladeunterbrechungen zu ermöglichen und zweitens den Renderprozess zu beschleunigen. Durch die gewonnene Zeit kann man zusätzliche Modelle darstellen, die wiederum die Bildqualität verbessern. Dabei wird das Kriterium der Interaktivität, welches man hier auf durchschnittlich 25 Bilder pro Sekunde setzt, nicht verletzt. Außerdem wird zum Zweck der Wissensvermittlung eine Datenbank aufgebaut, in der sich diverse Informationen zu einzelnen Objekten aus der fertigen Anwendung heraus hinterlegen, verändern und abrufen lassen.

Das Stadtmodell des alten Roms, welches dafür zum Einsatz kommt, entstand überwiegend durch die prozedurale Generierung von Gebäudemodellen. Bei dieser Erstellung von insgesamt über 22.000 Objekten wird jeweils zuerst die Grundform, also die äußeren Umrisse, des Hauses mit wenigen Vertexpunkten erzeugt. Darauf bestimmen benutzerdefinierte Regeln die Erstellung von drei weiteren Modellen für das selbe Objekt. Dieser prozedurale Vorgang teilt die Grundformen, insbesondere die Hauswände, in Kacheln und füllt diese mit Texturen von Wänden, Fenstern und Türen. Dabei wird für alle Modelle aus dem selben Texturvorrat gegriffen. Diese Eigenschaften der Datenbasis werden im entwickelten System berücksichtigt und führen zur Optimierung des Speichers sowie der Performance. Die pro Objekt generierten Modelle sind bereits LOD Stufen des Hauses in drei unterschiedlich hohen Auflösungen. Sie werden für ein kollisionsbasiertes, diskretes LOD System verwendet, dass die Modelle bei verschiedenen Distanzen zur virtuellen Kamera anzeigt. Dabei wird jedoch nicht der aktuelle Abstand aller Objekte zur Kamera berechnet, da dies zu einem frühzeitigen oder verspäteten LOD Austausch führen kann, den der Benutzer schneller wahrnehmen würde. Das entwickelte LOD System, wie auch das Szenenmanagement, basieren auf Kollisionsereignissen mit Platzhaltern für die Szenenobjekte.

Das LOD und Szenenmanagementsystem besteht aus den Phasen der Objektdatenbankerstellung, Szenenvorbereitung und dem Laufzeitpozess. Während die ersten beiden Phasen nur einmalig ausgeführt werden müssen, findet der Laufzeitprozess bei jedem Start der An-

wendung statt. Bei der Erstellung der Objektdatenbank werden redundante Daten, hier die Materialien und Texturen, von den Modellen getrennt und in separate Dateien mit einer eindeutigen ID gespeichert. Mit den einzelnen LOD Modellen wird ebenso verfahren. Dabei werden die benötigten IDs der Materialien oder Texturen in der davon abhängigen Datei gehalten. Zur Laufzeit der Anwendung kann man so alle Daten wieder zusammenbringen. Im Abschnitt der Szenenvorbereitung wird die leere Szene neben dem Gelände mit den Grundformen aller Gebäude gefüllt. Sie werden in der Anwendung nicht gerendert, sondern dienen als Kollisionsobjekte. Zusätzlich werden drei Sphären mit unterschiedlichen Radien als Kollisionsobjekte um die Kamera gehängt, die sich analog zu ihr bewegen. Die Sphären definieren die Bereiche, in dem die jeweiligen LOD Stufen geladen plus angezeigt werden und besitzen auch eine entsprechende ID. Ein Nebeleffekt der virtuellen Kamera lässt Objekte hinter der größten Sphäre vollständig verschwinden. So brauchen zur Laufzeit auch keine Modelle außerhalb der Sphären geladen und angezeigt werden. Zuletzt wird eine hierarchische Datenstruktur für alle Kollisionsobjekte angelegt, welche die benötigten Schnittberechnung wesentlich beschleunigt.

Die Grundidee des Laufzeitverfahrens ist das Laden und Anzeigen des entsprechenden LOD Modells in der Objektdatenbank bei der Kollision einer Sphäre mit einer Grundform. Sobald eine Kollision stattfindet, wird eine Ladeanfrage mit der ID des zu ladenden LOD Modells an einen Queue geschickt. Dieser verarbeitet und verwaltet alle Anfragen bezüglich der aktuellen Umstände, wodurch unnötige Ladeoperationen verhindert werden. Die bedeutende Eigenschaft des Ladequeues ist, dass er eine benutzerdefinierte Menge an parallelen Ladeoperationen ausführen kann, wodurch interaktive Frameraten erhalten bleiben. Da die Renderpipeline bereits auf Softwareebene startet, lässt sich diese durch parallele Programmierung erheblich beschleunigen. Während der Hauptprozess, also die Anwendung, weiter läuft, können im Hintergrund benötigte Daten geladen werden. Ist ein Ladevorgang beendet, wird das enthaltende Modell in der Szene instanziiert, mit seinen vorgeladenen Materialien und Texturen verknüpft und für den Renderprozess freigegeben. Alle anderen bereits instanziierten LOD Modelle des selben Objekts setzt man auf unsichtbar. Verlässt die letzte Sphäre eine Grundform, so werden alle instanziierten Repräsentationen des Objekts zerstört und die Dateien entladen, was Speicherplatz für neue Modelle freilässt. Mithilfe des Queues werden die dringlichsten Modelle zuerst geladen und das Stadtmodell abhängig vom Kamerastandpunkt schrittweise an die aktuelle Situation angepasst. Außerdem wird das Szenenmanagementsystem mit dem LOD System verbunden, indem man nur jeweils das aktuell ausreichende LOD Modell eines Objekts lädt und anzeigt. Durch dieses Verfahren reduziert man die Rechenzeit zur Echtzeitdarstellung einer Szene mit zahlreichen Geometriemodellen sowie den dafür benötigten Speicherplatz.

Für das Verknüpfen von Informationen mit einzelnen Objekten wird eine zentrale Datenbank auf einem Webserver angelegt, die in der Anwendung als Wissenvermittlung dient. Dabei vernetzt eine Tabelle die ID eines Objekts mit verschiedenen Hintergrundinformationen. Den Zugriff auf die Datenbank, hinsichtlich des Abrufs, der Erstellung und Veränderung von Datensätzen, steuern Programme auf dem Webserver. Die Webserverprogramme können dann von mehreren Instanzen der Anwendung gleichzeitig aufgerufen werden. Somit ist die

Informationsdatenbank für jeden Benutzer aktuell und zugleich aus der Anwendung heraus editierbar.

Umgesetzt wird das Konzept mit der Game-Engine Unity3D. Die Software bietet für das Auslagern von Szeneninhalten und das dynamische Laden zur Laufzeit sogenannte Asset-Bundles an. Die Kollisionsberechnungen finden im Physiksystem der Middleware statt und die Anwendungslogiken bezüglich des LOD und Szenenmanagementsystems implementiert man durch eigens geskriptete Komponenten. Um das ganze Verfahren zu testen, werden mithilfe der Game-Engine ausführbare Programme erstellt, in denen man die Eigenschaften des Systems verändert und die Auswirkungen auf die Anwendung hinsichtlich der Performance, Speicherauslastung und Bildqualität beobachtet. Ohne jegliche Optimierungen ist bei ca. 200 hochaufgelösten Modellen keine Interaktivität mehr möglich. Das heißt die Bildrate pro Sekunde sinkt unter 25 Hz. Mithilfe des Szenenmanagmentsystems ist die Darstellung der kompletten Szene mit einem Radius von 60 Einheiten bei einer Framerate von 23 Hz möglich. Die errechnete Bildqualität dafür liegt bei 10800 Einheiten und der benötigte Arbeitsspeicher ist 655 MB. Durch die Verbindung mit dem LOD System kann bei einer Framerate von 25 Hz mehr als die doppelte Bildqualität erlangt werden. Diese Verbesserung geht jedoch leicht auf die Kosten des Speicherplatzes, da für viele Objekte mehrere LOD Modelle geladen sind.

Das Hauptproblem der fertigen Anwendung ist der LOD Austausch, welcher auf jeder Distanz deutlich sichtbar ist. Die Problematik geht auf die prozedurale Erstellung der Modelle zurück und lässt sich durch das entwickelte System nur bedingt beheben. Eine Lösung den Poppingeffekt zu mindern ist das diskrete LOD System um ein Blending-Verfahren zu erweitern. Die Darstellungen der auszutauschenden Modelle werden dann über einen kleinen Zeitraum miteinander verblendet bis das neue Modell vollständig sichtbar ist. Da innerhalb des Zeitraums jedoch beide Objekte gerendert werden, sinkt die Performance dementsprechend. Das eigentliche Problem der stark abweichenden Repräsentationen eines Objekts wird dadurch nicht gelöst. Hierfür wäre eine Regelverbesserung der prozeduralen Modellgenerierung nötig, sodass sich Fenster plus Türen immer an der gleichen Stelle in den LOD Modellen befinden und die Farbe des Hauses konstant bleibt.

Außerdem erlaubt ein diskretes LOD die Benutzung von Vertexpufferobjekten, also die mehrfache Instanziierung des selben Modells in der Szene. Die Ausnutzung dieses Vorteils würde den benötigten Speicherplatz reduzieren und die Performance erhöhen. Momentan ist jedes Objekt Roms ein Unikat. Bei 20.000 Objekten fällt es dem Benutzer jedoch nicht auf, wenn sich Modelle selten wiederholen. Allein die zweifache Instanziierung aller Objekte innerhalb der Sphären würde den Speicherverbrauch halbieren und den Renderprozess deutlich beschleunigen. In der Informationsdatenbank könnten dann die erforderlichen Daten wie Position und Rotation zu den Instanzen gehalten werden.

Anhang A

Inhalte der DVD

Auf der beiliegenden DVD sind die schriftliche Ausarbeitung der Bachelorarbeit sowie der Quellcode aller erstellten Programme enthalten. Die DVD ist in folgende Verzeichnisse untergliedert:

- **Anwendung**

  Hier befindet sich das ausführbare Programm mit dem entwickelten LOD und Szenenmanagementsystem. Damit die Anwendung einwandfrei funktioniert, müssen die Ordner `Anwendung` und `Data` im gleichen Verzeichnis liegen. Außerdem sollten beide Ordner auf die Festplatte kopiert werden.

- **Bachelorarbeit**

  Das Verzeichnis beinhaltet die Bachelorarbeit als PDF-Dokument.

- **Data**

  Hier befindet sich die Objektdatenbank, welche alle Texturen, Materialien und LOD Modelle in AssetBundles hält.

- **Dokumentation**

  In diesem Ordner ist die Dokumentation des Quellcodes für das LOD und Szenenmanagementsystem sowie für weitere Skripte enthalten.

- **Informationsdatenbank**

  Das Verzeichnis umfasst die Programme des Webservers und die SQL-Anweisungen zum Aufbau der Informationsdatenbank.

- **RomeRebornUnity**

  In diesem Verzeichnis befindet sich das Unity3D-Projekt mit dem entsprechenden Quellcode für die Komponenten und zusätzliche Skripte. Das C# -Projekt ist hier ebenfalls zu finden.

Literaturverzeichnis

[AM99] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms. Technical report, Chalmers University of Technology, 1999.

[BC96] Dani Lischinski David Salesin John Snyder Bradford Chamberlain, Tony DeRose. Fast rendering of complex environments using a spatial hierarchy. Technical report, University of Washington and Microsoft Research, 1996.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. Technical report, Stanford University, 1975.

[Bri09] Manfred Brill. *Virtuelle Realität*. Springer-Verlag Berlin Heidelberg, 2009.

[Bro97] Brockhaus. *Brockhaus . Die Enzyklopädie*. F.A. Brockhaus GmbH Leipzig-Mannheim, 1997.

[Cho02] Li Yiu Chong. Virtual reality and computer games. Technical report, School of Creative Media, City University of Hong Kong, 2002.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. Technical report, University of California at Santa Cruz, 1976.

[Cor03] NVIDIA Corporation. Using vertex buffer objects. Technical report, NVIDIA Corporation, 2003.

[CZ02] Konrad Karner Christopher Zach. Fast event-driven refinement of dynamic levels of detail. Technical report, VRVis Research Center, 2002.

[Dis03] Hartwig Distler. *Wahrnehmung in Virtuellen Welten*. PhD thesis, Justus-Liebig-Universität Gießen, 2003.

[DLH03] J. Cohen A. Varshney B. Watson D. Luebke, M. Reddy and R. Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[FE96] Amitabh Varshney Francine Evans, Steven Skiena. Optimizing triangle strips for fast rendering. Technical report, State University of New York at Stony Brook, 1996.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[FRG04] Oscar Ripolles Francisco Ramos, Miguel Chover and Carlos Granell. Continuous level of detail on graphics hardware. Technical report, Universitat Jaume I, Depto. Lenguajes y Sistemas Informaticos, 2004.

[Fri08] Bernard Frischer. The rome reborn project. how technology is helping us to study history. Technical report, Institute for Advanced Technology in the Humanities, University of Virginia, 2008.

[Hav04] Herman J. Haverkort. Introduction to bounding volume hierarchies. Technical report, institute of information and computing sciences, utrecht university, 2004.

[Hee05] Rainer Heers. *Being There. Untersuchungen zumWissenserwerb in virtuellen Umgebungen*. PhD thesis, Eberhard-Karls-Universität Tübingen, 2005.

[Hop96] Hugues Hoppe. Progressive meshes. Technical report, Microsoft Research, 1996.

[Kra05] Carsten Krauter. Vergleich zweier effizienter continuous level-of-detail-algorithmen zur dreidimensionalen geländetriangulierung. Technical report, Fachhochschule Regensburg, 2005.

[Kra07] Robert Krause. Seminar game development engines and middleware. Technical report, Universität der Bundeswehr München, 2007.

[LC99] Tsai-Yen Li and Chih-Wei Chiang. Data management for visualizing large virtual environments. Technical report, Computer Science Department, National Chengchi University, 1999.

[Lev02] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. Technical report, University of California at Berkeley, 2002.

[LJ02] Michael Lewis and Jeffrey Jacobson. Game engines in scientific research. Technical report, Communications of the ACM, 2002.

[MB05] Manfred Brill Michael Bender. *Computergrafik, Ein anwendungsorientiertes Lehrbuch*. Hanser Verlag, 2005.

[MD97] David E. Sigeti Mark C. Millery Charles Aldrich Mark B. Mineev-Weinstein Mark Duchaineau, Murray Wolinsky. Roaming terrain: Real-time optimally adapting meshes. Technical report, Los Alamos National Laboratory, 1997.

[MG07] Michael Wimmer Markus Giegl. Unpopping: Solving the image-space blend problem. Technical report, Vienna University of Technology, 2007.

[PM06] Simon Haegler Andreas Ulmer Luc Van Gool Pascal Müller, Peter Wonka. Procedural modeling of buildings. Technical report, ACM SIGGRAPH, 2006.

[PM07] Simon Haegler Andreas Ulmer Luc Van Gool Pascal Müller, Peter Wonka. Image-based procedural modeling of facades. Technical report, ACM SIGGRAPH, 2007.

[Pri03] Rogers Y. Scaife M. Price, S. Using tangibles to promote novel forms of playful learning. Technical report, University of Sussex, Brighton, 2003.

[RGL05] Carla M. D. S. Freitas Rodrigo G. Luque, Joao L. D. Comba. Broad-phase collision detection using semi-adjusting bsp-trees. Technical report, Instituto de Informatica, UFRGS, 2005.

[SC97] Seth Teller Satyan Coorg. Real-time occlusion culling for models with large occluders. Technical report, MIT Laboratory for Computer Science, 1997.

[SJT91] Carlo H. Sequin Seth J. Teller. Visibility preprocessing for interactive walkthroughs. Technical report, University of California at Berkeley, 1991.

[SZ04] Oliver Düvel Stefan Zerbst. *3D Game Engine Programming*. Thomson Course Technology, 2004.

[TAFT92] Carlo H. Sequin Thomas A. Funkhouser and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. Technical report, University of California at Berkeley, 1992.

[TAMH08] Eric Haines Tomas Akenine-Möller and Naty Hoffman. *Real-Time Rendering*. A.K. Peters Ltd., 3 edition, 2008.

[TR07] Gudula Rünger Thomas Rauber. *Parallele Programmierung*. Springer Verlag, 2007.

[Vor03] Martin Vorländer. *CGI kurz und gut*. O'Reilly Verlag, 2003.

[WAR01] William J. Brown William A. Ruh, Francis X. Maginnis. *Enterprise Application Integration*. John Wiley and Sons, 2001.

[Zöb08] Dieter Zöbel. *Echtzeitsysteme, Grundlagen der Planung*. Springer-Verlag Berlin Heidelberg, 2008.