Fachbereich 12
Informatik und Mathematik

Bachelorarbeit

Displacement-Mapping mit Tessellation-Shader

eingereicht bei

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung

eingereicht von

Marcel Gorlt
Marcel.Gorlt@googlemail.com

Eingereicht am:

16. September 2013

Betreuung durch:

Dr. Daniel Schiffner

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungs-kommission vorgelegt und auch nicht veröffentlich.

Darmstadt, den 16. September 2013

_____

(Marcel Gorlt)

Danksagung

An dieser Stelle möchte ich bei Michael Belwe bedanken, der meinen Fortschritt immer überwacht hat und dafür gesorgt hat, dass ich weiter mache. Ohne ihn wäre die Arbeit nicht zustande gekommen.

Des Weiteren danke ich meinem Betreuer Daniel Schiffner, der mir immer entscheidend weiterhelfen konnte, wenn ich nicht mehr weiter wusste.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die folgende Bachelorarbeit beschäftigt sich mit dem Thema des Displacement-Mappings unter Verwendung eines Tessellation-Shaders.
Ziel dieser Arbeit ist es, ein Programm zu entwerfen, welches einen visuellen Einblick in die Tessellation und in das Displacement-Mapping bietet. Des Weiteren soll die Leistung der Tessellation mit der des Geometry-Shaders und einem QuadTree-Verfahren verglichen werden.

Dazu wird zuerst beschrieben, welche Umgebung zur Implementierung verwendet wurde. Anschließend werden zunächst einige geläufige Mappingverfahren betrachtet.
Im nächsten Abschnitt wird dann auf die Tessellation und das Displacement-Mapping in Bezug auf Funktionsweise und Implementierung eingegangen.
Es folgt ein Vergleich der Performance der unterschiedlichen Mapping-Verfahren untereinander. In einem weiteren Vergleich wird die Performance der Tessellation (mit Displacement-Mapping) mit einem Geometry-Shader und einem QuadTree-Verfahren anhand eines Terrains durchgeführt.
Abschließend erfolgt die Beschreibung des Programms.

1.2. Einführung

Bei der Darstellung von virtuellen Welten, wie zum Beispiel Computerspielen, gibt es einen Punkt, der einen hohen Stellenwert einnimmt: Detailstreue. Es wird hart daran gearbeitet, Szenerien möglichst realistisch darstellen zu können, dabei aber möglichst ressourcensparend zu sein. Denn beim Echtzeitrendering ist es notwendig, etwa 25 Bilder pro Sekunde zu erzeugen, damit ein flüssiger Ablauf gewährleistet werden kann. Und solch ein Bild kann aus vielen unterschiedlichen Objekten und somit aus sehr vielen Informationen bestehen.

Die Geometrie eines Objektes kann man in drei Level aufteilen. Das erste Level ist die Makrostruktur, sie gibt die Form des Objektes an und besteht aus einem Gitternetz von Primitiven. Zum Beispiel der grundsätzliche Aufbau eines Kopfes.
Die Mesostruktur beinhaltet kleinere Details, am Beispiel des Kopfes - die Form der Nase oder Warzen im Gesicht usw.
Das dritte Level ist die Mikrostruktur, diese Struktur besteht aus den kleinen schwer unterscheidbaren Feinheiten. Die Mikrostruktur eines Objektes wird in der Regel durch eine Textur erzeugt. Am Beispiel des Kopfes wären hier die Haut, sowie Unreinheiten und weitere Makel gemeint[1].

Üblicherweise erzeugt man die Makrostruktur eines Objektes aus einem Polygongitter und legt per Mapping die Mikrostruktur in Form einer Textur darüber. Beim Normal-Mapping zum Beispiel wird dem Betrachter durch Lichtberechnungen eine Mesostruktur vorgegaukelt. Betrachtet man in einem solchen Fall die Mesostruktur von der Seite, so ist das Objekt flach. Dementsprechend hat diese Struktur auch keinen

[1] (László Szirmay-Kalos, 2006)

Einfluss auf die Umgebung, zum Beispiel Kollision und Schattenwurf. Dies bietet allerdings Vorteile beim Ressourcenverbrauch, da das Objekt keine Mesostruktur besitzt. Die CPU muss dementsprechend nur die Informationen der Makrostruktur berechnen.

Das Displacement-Mapping bedient sich hingegen keiner Illusion, sondern erzeugt die Mesostruktur, indem es anhand einer Height-Map (Höhenkarte) die Punkte des Polygongitters entsprechend der Normalen verschiebt. Damit dieser Vorgang vernünftige Resultate liefern kann, muss zunächst das Polygongitter verfeinert werden. Das Displacement-Mapping ist kein neues Verfahren, denn es existiert seit bereits mehr als sieben Jahren[[2]].
Allerdings ist die benötigte Modifizierung des Gitters über eine CPU Implementierung nicht sehr ressourcensparend und damit nicht vorteilhaft beim Echtzeitrendering von komplexen virtuellen Welten, die aus einer Vielzahl von Objekten und somit auch aus vielen Primitiven bestehen.

Erst nachdem solch eine Modifizierung mithilfe der hardwaregesteuerten und leistungsfähigen Grafikkarte ermöglicht wurde, konnte man über die Nutzung des Verfahrens im größerem Ausmaß nachdenken.
Zunächst wurde solch eine hardwaregestützte Modifizierung durch die Einführung des Geometry Shaders realisiert, welche die Shader-Stages der Grafikpipeline erweiterte. Der Geometry-Shader erzeugt hierbei neue Primitive, knickt jedoch bei der Performance schnell ein, wenn es sich um viele Objekte und damit auch um viele Primitive handelt. Erst mit der Entwicklung des Tessellation-Shaders, der zwei weitere Shader-Stages hinzufügt, soll die effektive Nutzung des Displacement-Mappings möglich sein. Denn im Gegensatz zum Geometry-Shader, bei dem definiert werden muss, wie das neue Primitiv erstellt werden soll, ist beim Tessellation-Shader nur ein Faktor anzugeben, mit dem ein Primitiv unterteilt wird.
Diese Hardwareimplementierung der Unterteilung soll dem Tessellation-Shader einen Leistungsvorsprung gegenüber dem Geometry-Shader geben, vor allem wenn es sich um eine hohe Anzahl an Primitiven handelt, was zweifellos beim Echtzeitrendering in Computerspielen der Fall ist.

Wie die beiden Verfahren im Detail funktionieren und ob die Kombination aus beiden eine effektive Benutzung ermöglicht, wird im Folgenden behandelt.

[2] (László Szirmay-Kalos, 2006)

1.3. State of the Art - Level of Detail (LoD)

LoD-Verfahren beschäftigen sich mit der dynamischen Veränderung des Detailgrades beim Echtzeitrendering einer Szenerie, oftmals mit dem Fokus auf Terrains, da sich Terrains leicht erstellen lassen und sehr komplex und großflächig sein können.
Es gibt verschiedene Verfahren, die versuchen ein Terrain möglichst realistisch und ressourcensparend darzustellen. Dabei soll die Darstellung so dynamisch wie möglich sein. Objekte nah am Betrachter sollen mit höherem Detailgrad dargestellt werden, weiter entfernte mit geringerem und nicht sichtbare Objekte gar nicht erst berechnet werden. Es gibt neben der hier vorgestellten Hardwaretessellation mit Displacement-Mapping weitere verschiedene Ansätze dies zu implementieren:

- CDLOD; GPU-basierendes Rendering, verwendet QuadTree [3]
- Seamless Patches for GPU-Based Terrain Rendering; verwendet ebenfalls QuadTrees, allerdings mit dem Ansatz ein Quad mit vier Triangles darzustellen, um einen flüssigeren Übergang zwischen Quads unterschiedlicher Detailstufe zu ermöglichen [4]
- GPU Ray-Casting; Verwendet als Basis ein QuadTree-Terrain, ersetzt aber die zu rendernden Teile mit einem GPU-basierten Ray-Casting-Algorithmus[5]

Die Verfahren haben eins gemeinsam, sie verwenden eine QuadTree-Struktur. Diese Struktur besteht aus einem Quad, welches rekursiv in vier gleichgroße Quads unterteilt wird. Je mehr rekursive Schritte, desto mehr Unterteilungen, desto mehr Details. Folgende Abbildung verdeutlicht die QuadTree Struktur anhand eines Beispiels.

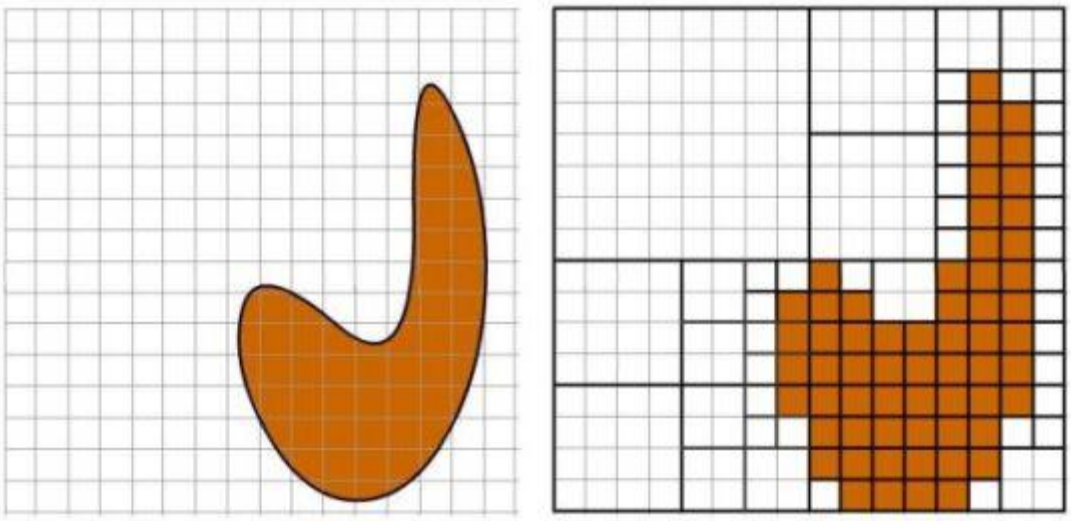

Abbildung 1-1: Links: Darstellung eines Objektes auf einer Karte
Rechts: Objekt in einer QuadTree Darstellung

[3] (Strugar, 2010)
[4] (Yotam Livny, 2007)
[5] (Christian Dick, 2009)

2. Grundlagen

Für das Verständnis dieser Arbeit sind Grundlagen in der graphischen Datenverarbeitung und der Funktionsweise von Shader nötig. Die Grundlagen der graphischen Datenverarbeitung werden als gegeben betrachtet. Da die neuen Shader-Stages einen erheblichen Aspekt dieser Arbeit darstellen, werden diese im kommenden Abschnitt näher erläutert.

2.1. Shader

Shader sind, was moderne Rechnersysteme betrifft, Module, die sich auf Grafikchips befinden und Bestandteil der Rendering-Pipeline (auch Grafik-Pipeline) sind. Sie werden beim Echtzeitrendering verwendet, um Objekten besondere Effekte zu verleihen, wie zum Beispiel metallisches Aussehen oder Wellen und andere Effekte bei Wasser.
Sie können über eine Programmierschnittstelle (API) individuell programmiert werden. Im Folgenden gehe ich von OpenGL als Schnittstelle aus, da diese zur Implementierung bei der Bearbeitung verwendet wurde. Eine weitere bekannte Schnittstellte ist Direct3D von Microsoft.

2.1.1. Vertex- und Fragment-Shader

Traditionell gibt es zwei programmierbare Shader-Stages. Zum einem den Vertex-Shader und zum anderen den Fragment-Shader, welche auch in dieser Reihenfolge abgearbeitet werden.

Der Vertex-Shader dient dazu, die Geometrie zu manipulieren und das Objekt somit zu verformen oder zu verschieben. Er arbeitet dabei jeden Vertex einzeln ab, so können zum Beispiel Wellenbewegungen erzeugt werden. Er ist jedoch nicht in der Lage, Vertices hinzuzufügen oder zu entfernen.

Die Aufgabe des Fragment-Shader besteht darin, die Fragmente zu verändern, er ist damit für das Aussehen eines Objektes verantwortlich. So verändert ein Fragment-Shader zum Beispiel die Darstellung der Textur und erweitert diese durch Beleuchtungseffekte um metallisches Aussehen.

2.1.2. Geometry-Shader

Mit OpenGL 3.2 war es möglich, den Geometry-Shader zu verwenden. Die Verwendung des Geometry-Shaders ist optional. Zuvor bestand das Rendering aus drei Schritten:

1. Transformation → Vertex-Shader
2. Beleuchtung → Fragment-Shader
3. Texturierung → Fragment-Shader

Mit dem Geometry-Shader kam ein weiterer Schritt zwischen dem Vertex- und dem Fragment-Shader dazu:

1. Transformation → Vertex-Shader
2. Hinzufügen/Entfernen von Vertices → Geometry-Shader
3. Beleuchtung → Fragment-Shader
4. Texturierung → Fragment-Shader

Der Geometry-Shader empfängt die Daten als Pakete vom Vertex-Shader, die Größe des Pakets richtet sich dabei nach der verwendeten Art der Primitive. Werden Triangles verwendet, erhält der Shader drei Vertices. Der Input ist demnach abhängig von den verwendeten Primitiven, der Output hingegen von der Implementierung des Shaders.
Die Output-Primitive müssen nicht den Input-Primitiven entsprechen, auch kann der Output gleich Null sein. Allerdings ist die Speicherreservierung abhängig von der Anzahl der maximalen Output-Primitive, die im Geometry-Shader definiert werden.
Wenn man zum Beispiel einen Geometry-Shader implementiert, der abhängig der Eigenschaften des Input-Primitivs eine variable Anzahl an Outputs erzeugt, dann wird in solch einem Fall immer der gleiche Speicherplatz reserviert. Übersteigt der Output den reservierten Speicherplatz, so fehlen Primitive in der Ausgabe.

Die Anzahl der Output-Primitive pro Input-Primitiv ist durch OpenGL beschränkt und beträgt aktuell maximal (OpenGL 4) 256 Vertices, dies entspricht etwa 85 Triangles ohne Mehrfachnutzung von Vertices.

Der Geometry-Shader wird auf jedes Primitiv einmal ausgeführt. Mit OpenGL 4 besteht die Möglichkeit ihn mehrmals pro Primitiv ausführen zu lassen. Anwendungsbeispiele sind:

- Weichzeichnen von Objekten
- Erzeugung von Schattenvolumen oder
- Erzeugung von Fell- und Haargeometrie.

Er ist nicht darauf ausgelegt, eine große Anzahl an Primitiven zu erzeugen. Für solche Zwecke wurde der Tessellation-Shader mit OpenGL4 beziehungsweise DirectX11 implementiert. Entsprechend ist der Shader nur auf modernen Grafikkarten verfügbar, die diesen Standard unterstützen.

2.1.3. Tessellation-Shader

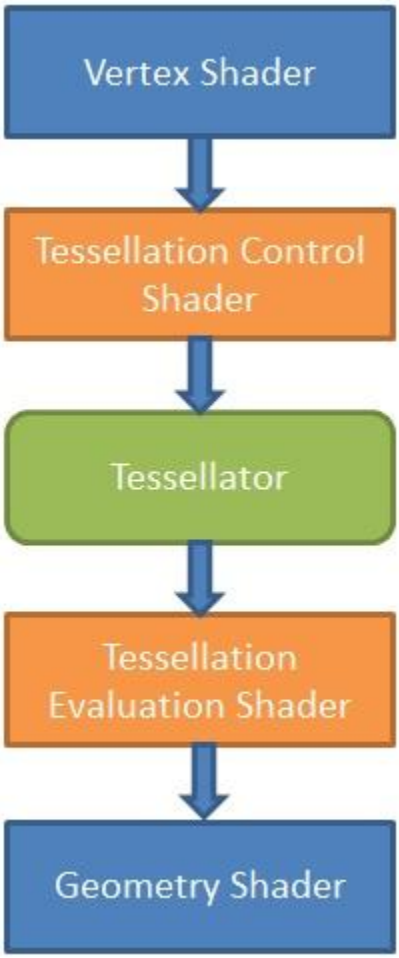

Abbildung 2-1 Shader-Stages

Im folgenden Abschnitt werden die Funktionen der einzelnen Schritte des Tessellation-Shaders erläutert, jedoch nicht wie Tessellation konkret funktioniert. Dies erfolgt in Kapitel 4.

Der Tessellation-Shader ist eine verbesserte Implementierung des Geometry-Shaders.
Er fügt zwei programmierbare Shader-Stages ein. Den Tessellation-Control-Shader (TCS) und den Tessellation-Evaluation-Shader (TES). Diese Stages werden zwischen dem Vertex-Shader und dem Geometry-Shader ausgeführt[6]. Die Abbildung 3-1 zeigt den Ablauf der Shader-Stages

Im Gegensatz zum Geometry-Shader, mit dem festgelegt werden kann, wie und wo neue Primitive entstehen, bleiben beim Tessellation-Shader die äußere Struktur und der Standort des Primitivs unverändert und es wird nur das innere Gitternetz verfeinert.
Die neuen Primitive befinden sich daher in derselben Ebene wie das Input-Primitiv. Das bedeutet, dass ein Betrachter keine Veränderung feststellt, außer in der Hinsicht, dass die Grafikkarte eine Mehrarbeit leistet. Erst im Zusammenhang mit Displacement-Mapping zeigt Tessellation seinen Nutzen.

Der TCS erhält als Input einen Patch, dieser besteht aus einer Liste von Vertices, die vom Vertex-Shader kommen.
Bei der Verwendung von Tessellation wird im Vertex-Shader in der Regel nur eine Transformation der Vertices in den World-Space durchgeführt. Die weitere Transformierung in den Screen-Space erfolgt nach dem TCS.

Ein Patch ist kein Primitiv, sondern ein Polygonnetz von der Größe der Anzahl der Vertices. Im Falle von Triangles besteht ein Patch aus drei Vertices. Des Weiteren spricht man in diesen Zusammenhang auch von Control-Points anstatt von Vertices.
Ein Patch kann aus bis zu 32 Control-Points bestehen.
Im Gegensatz zum Vertex-Shader ist dem TCS bekannt, welche Vertices zu einem Patch beziehungsweise Primitiv gehören.

Im TCS wird das Tessellation-Level bestimmt, welches angibt, wie oft ein Patch unterteilt werden soll.
Je nachdem, welches Template benutzt wird (Lines, Triangles oder Quads), gibt es unterschiedliche Anzahlen von Tessellation-Level-Variablen, die bestimmt werden

[6] (Hart, 2013)

müssen (Abbildung 3-2). So müssen für ein Triangle drei “Outer-Tess-Level” und ein “Inner-Tess-Level” angegeben werden.
Das Inner-Level gibt dabei an, wie viele neue Primitive im Inneren erstellt werden sollen. Entstehen Primitive durch die Angabe eines Inner-Level, so wird auf diese entstandenen Primitive wiederum das Outer-Level angewendet.

Das Outer-Level gibt an in wie viele Teile eine Kante unterteilt werden soll.
Die Bestimmung des Tessellation-Levels kann auf unterschiedliche Art und Weise erfolgen. Die einfachste Möglichkeit besteht darin, ihm einen festen Wert zuzuordnen.

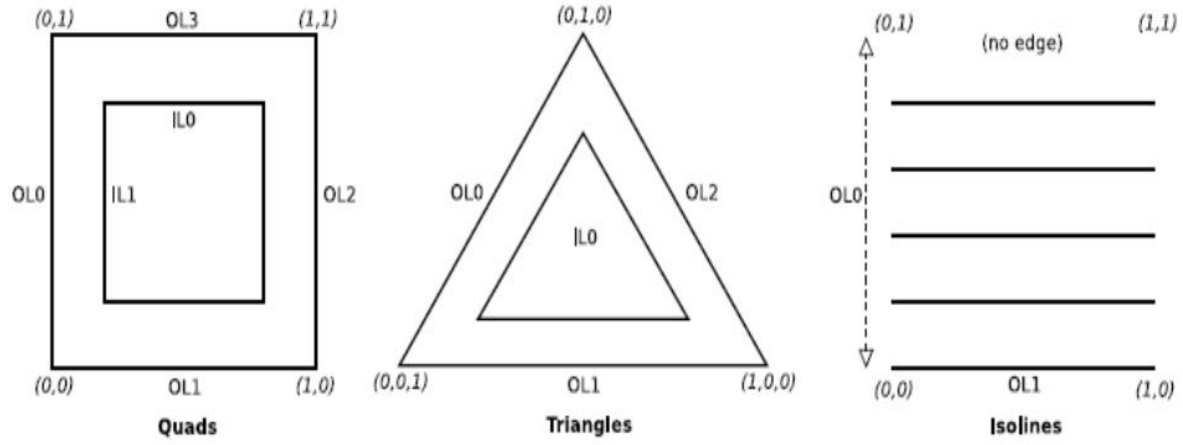

Abbildung 2-2-2: Templates des Tessellators; OL = Outer-Level; IL = Inner-Level

Eine andere Möglichkeit ist es, einen Algorithmus zu implementieren, der das Tessellation-Level in Abhängigkeit von der Distanz zwischen den Control-Points und der Kamera bestimmt. Je kleiner die Distanz, desto höher das Tessellation-Level, dadurch wird ein hohes Maß an Details erzeugt.

Der TCS wird einmal für jeden Control-Point des Patches ausgeführt.

Die Daten des Patches werden an den Tessellator weitergeleitet. Der Tessellator ist eine vom Anwender nicht-programmierbare Einheit, welche die Unterteilung entsprechend des Tessellation-Levels durchführt. Er wird einmal pro Patch ausgeführt, allerdings nicht auf den Patch, sondern auf ein Template (Schablone) des Patches. Der Tessellator arbeitet mit parametrisierten Koordinaten und normalisierten Werten, dass bedeutet es werden (u,v) beziehungsweise (u,v,w) Koordinaten und Werte zwischen 0 und 1 verwendet(Abbildung 3-2). Bei Triangles werden dementsprechend die Baryzentrischen (uvw) Koordinaten verwendet.

Der TES erhält diese parametrisierten Daten und berechnet entsprechend der Implementierung die endgültige Position der neuen Vertices, indem es die Koordinaten von Baryzentrischen (uvw) zu Kartesischen (xyz) übersetzt. Er wird für jeden Vertex ausgeführt, der vom Tessellator übergegeben wird.

Die Implementierung Displacement-Mappings erfolgt im TES.

2.2. Werkzeuge/Umgebung

Zur Implementierung des Programmierteils wurde die Engine von JMonkey verwendet, sie ist Open-Source und kann auf www.jmonkeyengine.org heruntergeladen werden. Sie verwendet als grafische Programmierschnittstelle OpenGL und die Entwicklungsumgebung entspricht der von NetBeans.
Der Tessellation-Shader steht erst mit OpenGL Version 4 zur Verfügung.
In der aktuellen Version (JME3) unterstützt die Engine lediglich OpenGL Standard bis Version 3.2. Dementsprechend steht zwar der Geometry-Shader zur Verfügung, jedoch nicht die Tessellation.
Um Tessellation-Shader verfügbar zu machen, muss die Engine und damit ihr Source-Code erweitert beziehungsweise modifiziert werden. Wird JMonkey von der Website heruntergeladen, erhält man den Source-Code in kompilierter Version, dieser kann damit nicht verändert werden.
Über einen SVN-Checkout mit einem entsprechendem Programm (GitHub) und folgender URL: ***http***://jmonkeyengine.googlecode.com/svn/trunk/, erhält man den Source-Code.

Die Modifizierung des Codes erfolgt in mehreren Dateien. Es werden die beiden neuen Shader-Stages als Token implementiert und anschließend mit den entsprechenden OpenGL Begriffen verknüpft, sodass JME erkennt, ob Tessellation-Shader angegeben wurden und welche OpenGL Version dafür verwendet werden muss.

Als Basis für die Modifizierung wurde ein bereits vorhandener Patch verwendet[7]. Da der Patch aus dem Jahr 2012 ist, funktioniert er nicht auf der aktuellen Version (August 2013) und musste daher entsprechend angepasst und aktualisiert werden.

Da die Engine permanent geupdatet wird, kann nicht garantiert werden, dass der Patch weiterhin funktioniert. Er muss eventuell neu angepasst werden.

Der Patch ist nur notwendig, wenn etwas mit JMonkey programmiert wird. Für das fertige und kompilierte Programm ist es ausreichend eine entsprechende Grafikkarte zu besitzen.

[7] (PasteBin, 2012)

2.3. Systemspezifikation

Für die Performancevergleiche wurde folgendes System verwendet:

- Betriebssystem: Windows 7
- Board: Gigabyte P55-USB3
- CPU: Intel Core i5 760 (2.8GHz)
- RAM: 4GB DDR3-1333
- Grafikkarte: Nvidia GeForce GTX460 (DX11, OpenGL 4.1)
    - Die GTX460 ist ein Modell aus dem Jahr 2010 und gehört damit zu einer der ersten DX11 Grafikkarten

Die Programme und Shader wurden mit JMonkey (Java) und in GLSL (OpenGL) geschrieben.

3. Mapping

Bei der einfachsten Variante des Mappings wird auf die Makrostruktur des Objektes eine Textur drübergelegt. Dadurch erhält das Objekt ein Aussehen. Durch Algorithmen lassen sich besondere Effekte erzeugen, die die Qualität des Aussehens verbessern.

In den folgenden Abschnitten werden drei Verfahren vorgestellt, die wie das Displacement-Mapping dazu verwendet werden eine Mesostruktur des Objektes zu erzeugen (Abbildung 4). Anders ausgedrückt, wird versucht, mit 2D-Texturen 3D-Objekte zu erstellen.

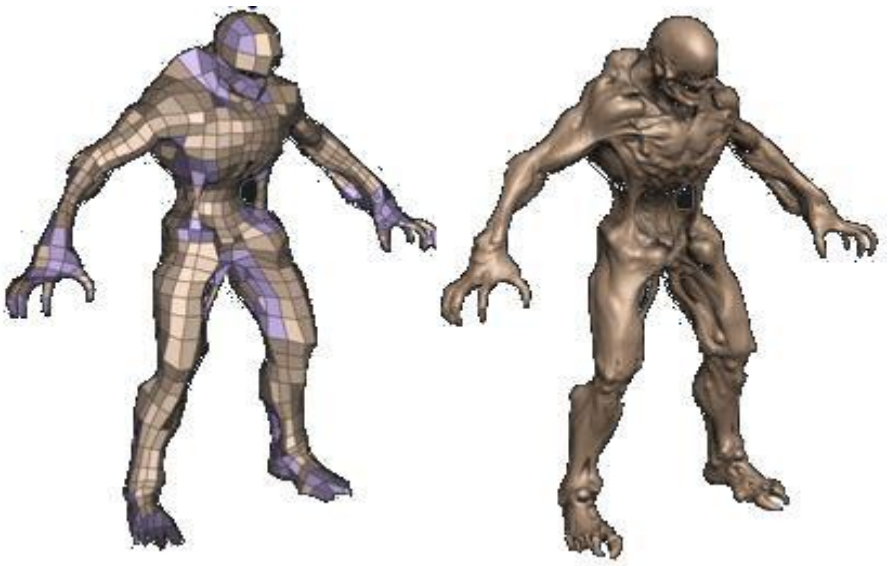

Abbildung 3-1: Links: Objekt in seiner Makrostruktur Rechts: Objekt mit Mesostruktur

3.1. Bump-Mapping

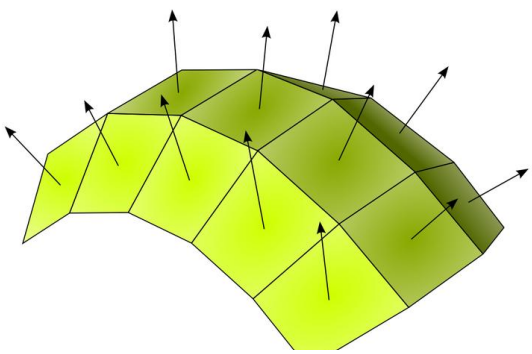

Abbildung 3-2 Darstellung von Normalvektoren

Mit dem Verfahren des Bump-Mappings, welches 1978 von James F.Blinn beschrieben wurde[8], ist es möglich der Oberfläche eines Objektes Unebenheiten (im Englischem “bumps”) und andere kleine Strukturelemente zu verleihen ohne dabei die Geometrie des Objektes zu verändern.

Dazu werden anhand einer Height-

[8] (Blinn, 1978)

Map die Oberflächennormalen des Objektes verändert. Anschließend werden diese veränderten Normalen für die Berechnung der Beleuchtung verwendet.
Die Normale einer (Polygon-)Fläche ist ein Vektor, der Senkrecht zu dieser Fläche steht und die Ausrichtung der Fläche in dem Raum angibt (siehe Abb. 3-2).
Um das Bump-Mapping zu verwenden, wird zunächst eine Height-Map in Form eines Graustufenbildes benötigt. Bei einem Graustufenbild hat jeder Punkt einen Wert zwischen 0 und 255. Je höher der Wert, desto höher liegt in der Regel der Punkt und desto heller ist die Farbe.

In der Berechnung der Beleuchtung des Objektes wird zunächst die Normale des Punktes anhand der Height-Map bestimmt und anschließend mit der des Objektes kombiniert. So werden Unebenheiten bei der Beleuchtung berücksichtigt, die nicht vorhanden sind und es entsteht die Illusion von Tiefe.

Abbildung 3-3 zeigt ein Beispiel, es wird ein Objekt bestehend aus zwei Polygonen (Triangles) verwendet. Aus der Textur, die auf das Objekt gelegt werden soll, wird ein Graustufenbild erstellt. Anschließend werden die Normalen modifiziert, das Ergebnis ist im Bild unten links der Abbildung 4-3 zu sehen. Im Vergleich mit der Textur sind die Bereiche zwischen den Steinen dunkler. Dies sind Schatten die die Tiefenillusion verursachen.

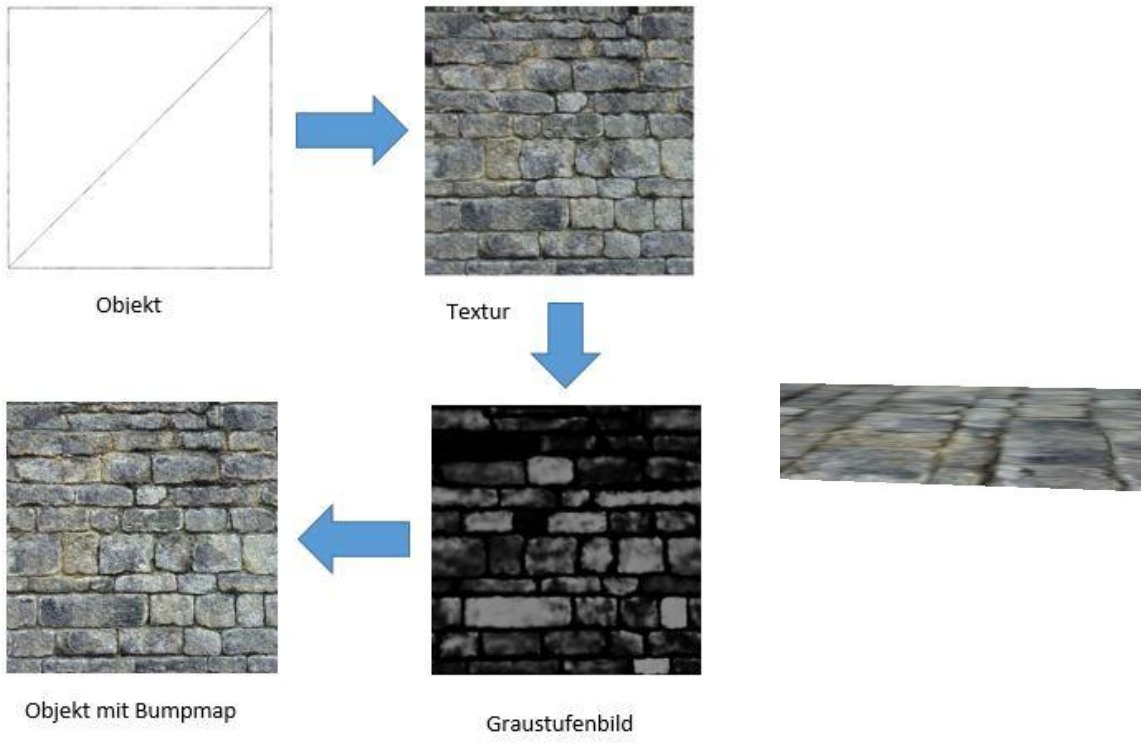

Abbildung 3-3 Links: Schritte des Bump-Mappings; Rechts: Seitenansicht des Ergebnis

Vorteil des Bump-Mappings ist, dass Objekte Polygonsparend erzeugt werden können und ihnen trotzdem noch ein detailliertes Aussehen verliehen werden kann. Des Weiteren können Graustufenbilder ohne großen Aufwand per Hand angefertigt werden.

Nachteil des Bump-Mappings ist unter anderem, dass die Silhouette und der Schattenwurf des Objektes unverändert bleiben. Das bedeutet, eine Kugel mit glatter Oberfläche, bei der durch Bump-Mapping Unebenheiten erzeugt wurden, ist der Schatten weiterhin ohne jegliche Unebenheiten. Des Weiteren nimmt der Effekt der Tiefe ab, je flacher der Betrachtungswinkel wird (Abbildung 7).

Ein weiterer Kritikpunkt ist die Verwendung eines Graustufenbildes. Das Format des Graustufenbildes ist auf 256 unterschiedliche Höhenstufen beschränkt, was bei Texturen mit vielen Höhenunterschieden zu Stufenbildung führt.
Auch funktioniert das Verfahren nur, genauso wie die folgenden, wenn eine Lichtquelle existiert, die sich außerhalb des Objektes befindet, da die Textur beleuchtet werden muss, damit ein Effekt zustande kommt. Eine innere Beleuchtung wirkt sich nicht auf die Oberfläche der Textur aus.

3.2. Normal-Mapping

Das Normal-Mapping(auch "Dot3-bump-Mapping" genannt) stellt eine Weiterentwicklung des Bump-Mappings dar und verwendet das gleiche Prinzip. Es wird eine Mesostruktur an der Oberfläche des Objektes erzeugt, indem die Oberflächennormale vor der Belichtungsberechnung verändert wird.

Die Idee des Normal-Mapping wurde von Cohen [9] und Cignoni[10] jeweils 1998 vorgestellt. Beide Publikationen beschäftigen sich mit der Theorie niedrigauflösende Objekte als hochauflösende darzustellen.

Der Unterschied zwischen Normal- und Bump-Mapping besteht darin, dass beim Normal-Mapping die Oberflächennormale auf Grundlage anderer Informationen verändert wird. Ein Graustufenbild, wie es beim Bump-Mapping verwendet wird, enthält Informationen über die Höhe eines Punktes, was bedeutet, dass in solch einem Fall auch die Normale nur in einer (z-)Achse verändert werden kann.

Beim Normal-Mapping wird kein Graustufenbild, sondern eine Normalmap verwendet. Eine Normalmap ist eine spezielle Textur bei der keine Farbwerte, sondern (xyz)-Koordinaten von Normalen in den RGB-Kanälen gespeichert werden. Das führt dazu, dass bei der Modifizierung der Oberflächen-Normalen nicht die Höhe verändert wird, sondern die Ausrichtung der Normale. Dadurch bleibt auch bei schrumpfendem Betrachtungswinkel ein gewisses Maß an Tiefe erhalten.
Abbildung 4-4 zeigt das Normal-Mapping anhand eines Beispiels.

Im Vergleich zum Bump-Mapping ist die Tiefe ausgeprägter. Bei flachem Betrachtungswinkel nimmt der Tiefeneffekt zwar ab, ist aber dennoch festzustellen. Das führt dazu, dass sich das Verfahren im Gegensatz zum Bump-Mapping auch bei

[9] (Jonathan Cohen, 1998)
[10] (P.Cignoni, 1998)

runden Geometrien lohnt. Wie auch beim Bump-Mapping bleibt die Geometrie des Objektes unverändert. Das bedeutet auch, dass sich die Silhouette des Objektes und damit auch der Schattenwurf auf die Umgebung nicht ändern.

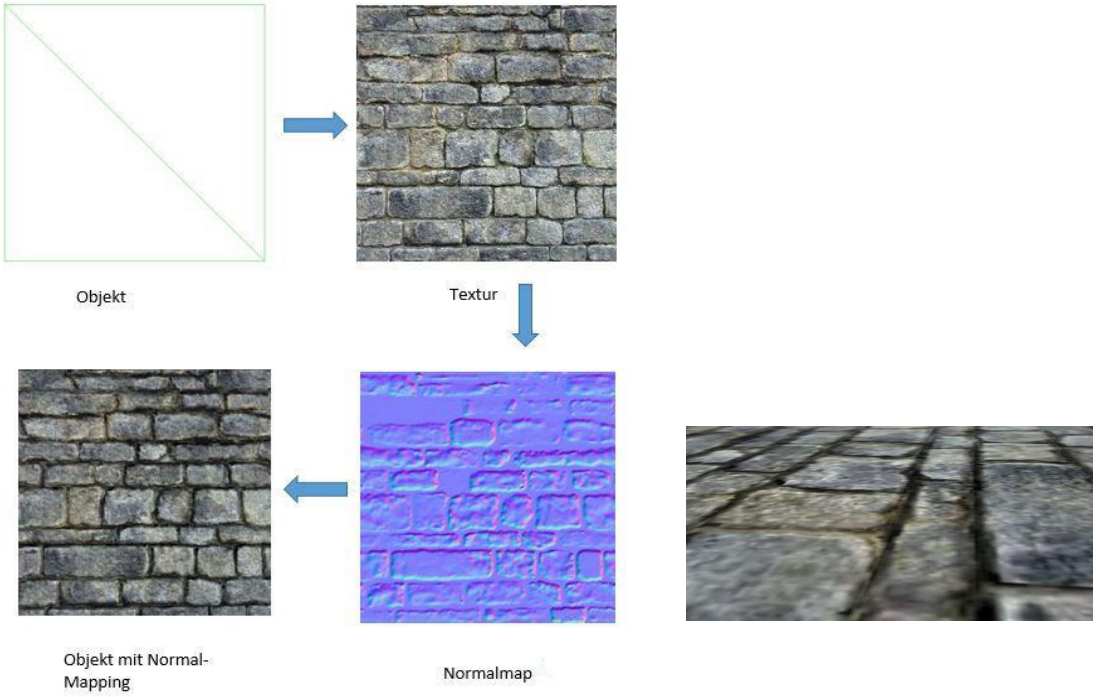

Abbildung 3-4 Links: Schritte des Normal-Mappings; Rechts: Seitenansicht

Zusammenfassend bietet das Normal-Mapping die gleichen Möglichkeiten wie das Bump-Mapping und mehr. Zwar benötigt das Normal-Mapping mehr Ressourcen, da die Normalmap umfangreicher ist, bietet jedoch ein sehr gutes Verhältnis zwischen Performance und Aussehen(siehe 5.3). Daher ist es eines der am meisten verwendeten Verfahren in der Spieleindustrie, vor allem nachdem Shadermodule großflächig verfügbar wurden und auch in Konsolen enthalten waren. Die PlayStation 3 (2006) und die XBOX360 (2005) verwenden bei ihren Spielen hauptsächlich Normal-Mapping.

Eine der Standardvarianten des Normal-Mapping ist es, zwei unterschiedliche Versionen eines Modells zu erstellen. Ein Modell ist dabei hochauflösend, während das andere niedrigauflösend ist. Die Differenz der beiden Versionen wird dann in einer Textur beschrieben. Im Spiel beziehungsweise der Anwendung wird dann die niedrigauflösende Version mit der hochauflösenden Textur verwendet.

3.3. Parallax-Mapping

Das Parallax-Mapping ist eine Erweiterung des Normal-Mappings und wurde 2001 von Tomomichi Kaneko eingeführt[11]. Das Grundprinzip ist identisch mit dem des Normal- und Bump-Mappings.

Anhand von Veränderungen der Oberflächennormale wird eine 3D-Struktur vorgegaukelt.
Das Parallax-Mapping kombiniert dabei die Varianten Normal- und Bump-Mapping, um die Illusion der Tiefe zu verbessern, insbesondere bei kleinem Betrachtungswinkel. Um dies zu erreichen, wird sowohl eine Normal-Map als auch ein Graustufenbild verwendet.

Das Format der Normalmap ermöglicht es, die Werte eines Graustufenbildes im Alpha-Kanal der Normalmap zu speichern, so wird nur eine Datei anstatt zwei benötigt.

In die Berechnung der neuen Oberflächennormalen gehen hier beide Werte ein. Jedoch ist der Einfluss der Grauwerte größer, je flacher der Winkel ist. Somit wird die Verschiebung verstärkt und der Abschwächung des Tiefeneffekts entgegengewirkt.
Ist der Offset, also die Verschiebung, zu hoch, so wird die Darstellung fehlerhaft (Abb. 4-5), da die Textur zu stark verzerrt wird.

Abbildung 3-5 Textur mit zu hohem Offset

Abbildung 3-6 Links: Parallax-Seite; Rechts: Steep-Parallax-Seite

Neben dem Parallax-Mapping gibt es noch das Steep-Parallax-Mapping.

Steep-Parallax-Mapping erweitert das Parallax-Mapping um die Verdeckungs-komponente. Beim Steep-Parallax-Mapping wird der Sehstrahl (Eye-Ray-Tracing) berücksichtigt, also das, was unsere Augen sehen würden. Das bedeutet, wenn ein Höhenfeld ein anderes Feld überlagert, so wird in diesem Fall auch nur das vordere Feld berücksichtigt und dargestellt[12]. Das führt dazu, dass bei kleinem Winkel der

[11] (Tomomichi Kaneko, 2001)
[12] (McGuire, 2005)

Eindruck entsteht, dass das Objekt nicht mehr flach wäre (Abb. 4-6), was aber der Fall ist, da die Geometrie unverändert bleibt.

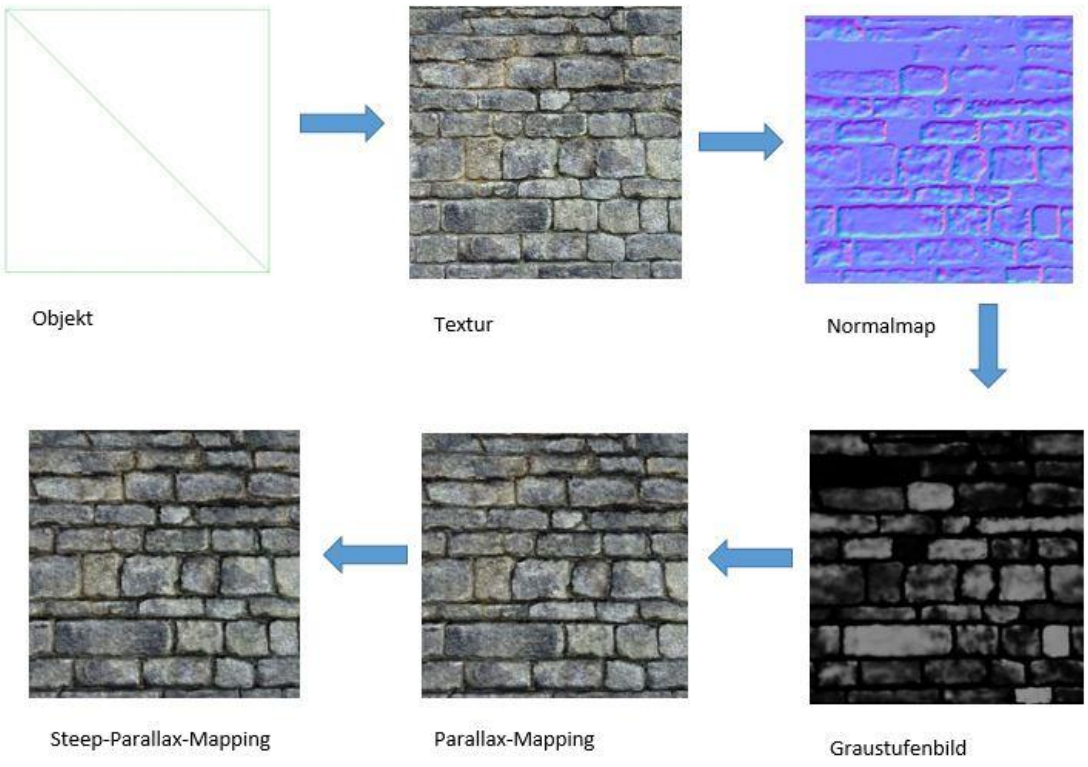

Abbildung 3-7 Ablauf des Parallax- und Steep-Parallax-Mapping

In Abbildung 4-7 ist Steep-Parallax- und das Parallax-Mapping abgebildet.

Zusammenfassend bestehen die Vorteile des Parallax-Mapping darin, dass selbst bei kleinem Winkel eine Tiefe erkennbar ist. Beim Steep-Parallax-Mapping ist diese Tiefe noch kräftiger aufgrund des Eye-Ray-Tracing.

Nachteile sind der erhöhte Berechnungsaufwand, sowie die unveränderte Geometrie.

4. Displacement-Mapping mit Hardware-Tessellation

In den folgenden zwei Abschnitten werden beide Verfahren erläutert. Da Tessellation das Werkzeug des Displacement-Mapping ist, wird zunächst auf die Tessellation eingegangen und ihre Implementierung betrachtet. Im darauffolgenden Abschnitt wird das Displacement-Mapping in seinen einzelnen Schritten erklärt. Im Anschluss werden ein paar visuelle Vergleiche vorgestellt und die Performance betrachtet.

4.1. Tessellation

Das Wort “Tessellation” beschreibt das Verfahren der Unterteilung einer Fläche in gleichgroße Teilflächen. Dabei bleiben Form und Größe der Fläche unverändert, da sich die Teilflächen in der gleichen Ebene befindet. Das Muster einer Bienenwabe (Sechseckgitter) stellt ein gutes Beispiel für Tessellation dar.

Erste hardwaregestützte Implementierungen von Tessellation erfolgten bereits 2005 unter anderem von Bunnell [13]. Hier hat man iterativ das Polygonnetz verfeinert und zunächst in temporäre Buffer auf der GPU gespeichert, anschließend wurde das finale Objekt gezeichnet. Dieses Verfahren erzeugte jedoch ein hohes Maß an Datentransfers zur und von der GPU, daher war die Effektivität durch den GPU Speicher limitiert.
Erst nachdem modernere GPUs den Zugriff auf Objekte innerhalb der GPU ermöglichten [14] und somit den Datentransfer der oben genannte Methode beseitigte, konnte man die Hardware-Tessellation effektiv implementieren, da man die Oberfläche eines Objektes direkt auswerten konnte.

Damit Hardware-Tessellation verwendet werden kann, wird zunächst eine Grafikkarte benötigt, die die OpenGL 4 beziehungsweise die DirectX11 Version unterstützt. Erst durch diese Versionen sind die zwei neuen Shader-Stages Verfügbar die die Tessellation ausführen können.

Die Tessellation in der graphischen Datenverarbeitung erfolgt in mehreren Schritten. Nachdem ein Objekt an die GPU weitergeleitet wurde und die Vertices den Vertex-Shader passiert haben, werden sie an den Tessellation-Control-Shader (TCS) weitergegeben. Im TCS werden dann je nach verwendeten Primitiven die Tessellation-Levels festgelegt (Abb. 5-1).

[13] (Bunnell, 2005)
[14] (Loop, 2013)

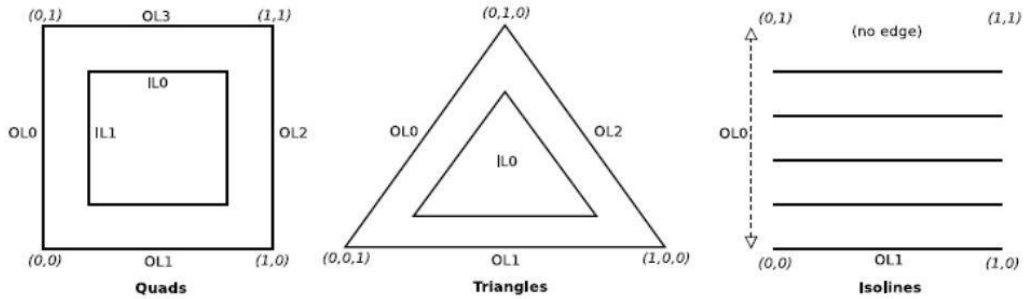

Abbildung 4-1

Entsprechend den Tessellation-Levels werden die Primitive unterteilt, unterstützt werden Tessellation-Levels von 1-64. Da das Tessellation-Level dynamisch angepasst werden kann, kann das Verfahren auch für LoD verwendet werden.

4.1.1. Isolines

Da für Isolines die Angabe von nur einem Tessellation-Level notwendig ist, sind sie der einfachste Fall bezüglich der Unterteilung. Die Linie wird entsprechend des Levels unterteilt. Bei einem Level von 2 wird die Linie halbiert und es wird in der Mitte ein neuer Vertex erstellt, bei drei wird sie gedrittelt und zwei neue Vertices entstehen. Dadurch lassen sich aus geraden Linien Kurven erzeugen, wenn die neuen Vertices im Tessellation-Evaluation-Shader (TES) entsprechend platziert werden.

4.1.2. Triangles

Bei Triangles sind insgesamt vier Tessellation-Level anzugeben. Ein Outer-Level für jeweils eine Kante und ein Innerlevel. Die Kanten werden entsprechend des Outer-Levels in gleich große Teile unterteilt. Folgende Bilderreihe (Abb. 5-2) zeigt die Tessellation bei der Veränderung des Outer-Levels, das Innerlevel bleibt unverändert. Es wird einmal ein neuer Vertex in der Mitte des Triangles erzeugt. Alle neu entstanden Vertices auf den Kanten verwenden dann den Vertex in der Mitte und bilden so neue Triangles, die für eine weitere Verarbeitung aufgrund des gemeinsamen Vertex in der Mitte nur wenig brauchbar sind.

Erst durch die Veränderung des Innerlevels entstehen neue Triangles innerhalb des Primitivs.

In der Praxis wird für gewöhnlich das Inner- und Outer-Level gleichermaßen verändert (Inner-Level = Outer-Level). Wird das Innerlevel erhöht dann richtet sich die Entstehung neuer Vertices nach der Unterteilung der Kanten. Bei einem Inner-Level von 2, werden die äußeren Kanten in zwei Teile aufgeteilt. Zieht man nun Linien ausgehend von den Punkten der Kanten, wo diese geteilt wurden, gibt es einen

Schnittpunkt innerhalb dieses Primitivs. Auf diesem Schnittpunkt wird ein neuer Vertex erzeugt.

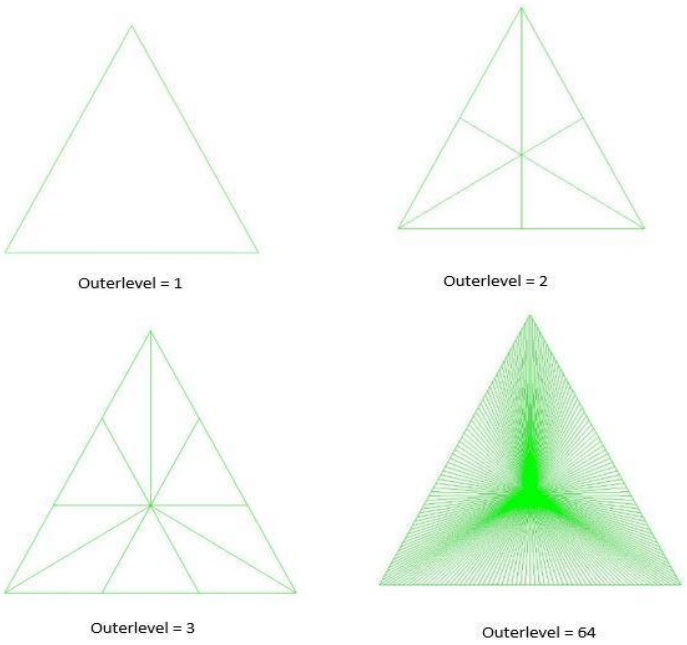

Abbildung 4-2 Tessellation Triangle, Inner-Level = 1

Das bedeutet, dass ein Inner-Level von 2 noch kein neues Triangle erzeugt, sondern, wie beim Outer-Level, nur einen einzelnen Vertex in der Mitte des äußeren Triangles.

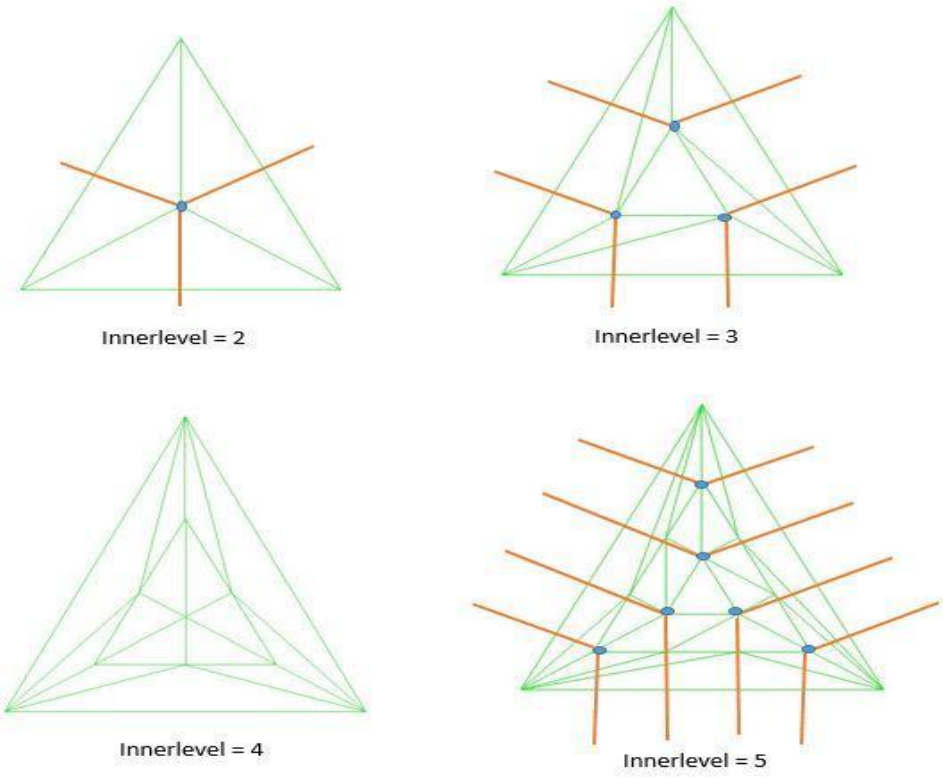

Abbildung 4-3 Bilderreihe Entstehung neuer Triangles

Erst ab einem Inner-Level von 3 entstehen drei Schnittpunkte, diese ergeben dann ein Triangle. Die drei äußersten Schnittpunkte ergeben die Eckpunkte des neuen Triangles. Alle weiteren Schnittpunkte und somit Vertices, werden als Outer-Level des neu entstandenen Triangles gewertet. Dadurch entsteht eine sukzessive Unterteilung in immer weitere Schichten, bis das innerste Triangle nur noch aus drei Eckpunkten oder einem Mittelpunkt besteht (Abb. 5-3).

Die nächste Bilderreihe (Abb. 5-4) zeigt die Unterteilung bei jeweils gleichen Inner- und Outer-Level. Es tauchen sich wiederholende Muster auf und es lassen sich aus einem Triangle bis zu 6144 Triangles erzeugen.

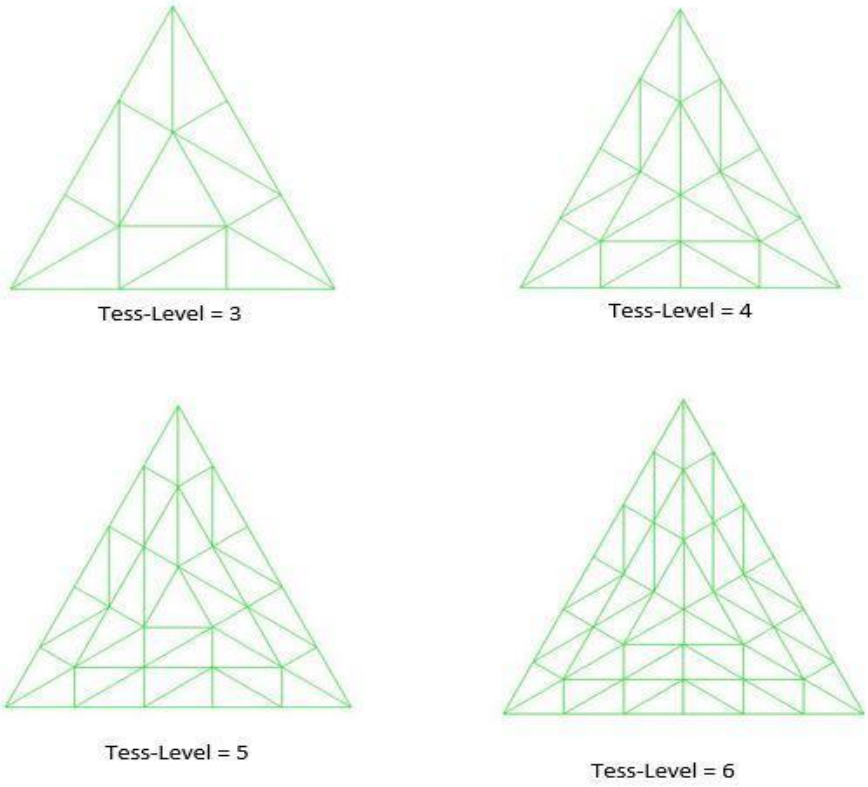

Abbildung 4-4 Outer- und Inner-Level gleichmäßig verändert

4.1.3. Quads

Bei Quads gibt es insgesamt sechs Tessellation-Level, vier Outer-Level und zwei Inner-Level. Wie bei den Triangles, geben die Outer-Level an, in wie viele Teile die Kanten unterteilt werden sollen.
Das Inner-Level bestimmt, wie viele Quads pro Reihe beziehungsweise pro Spalte entstehen. Jedoch geht dies nur bei ungeraden Werten perfekt auf, da sich immer ein Quad um den Mittelpunkt bildet(siehe Abb. 5-5).

Bei Quads, welche aus zwei Triangles aufgebaut sind, lassen sich aus einem Quad 3969 Quads (7938 Triangles) erzeugen.

Vergleicht man diese Werte mit denen vom Geometry-Shader ist dies ein enormer Unterschied. Beim Geometry-Shader können aus einem Primitiv 256 Vertices erstellt werden, was etwa 85 Triangles entspricht, sofern jeder Vertex nur einmal verwendet wird.

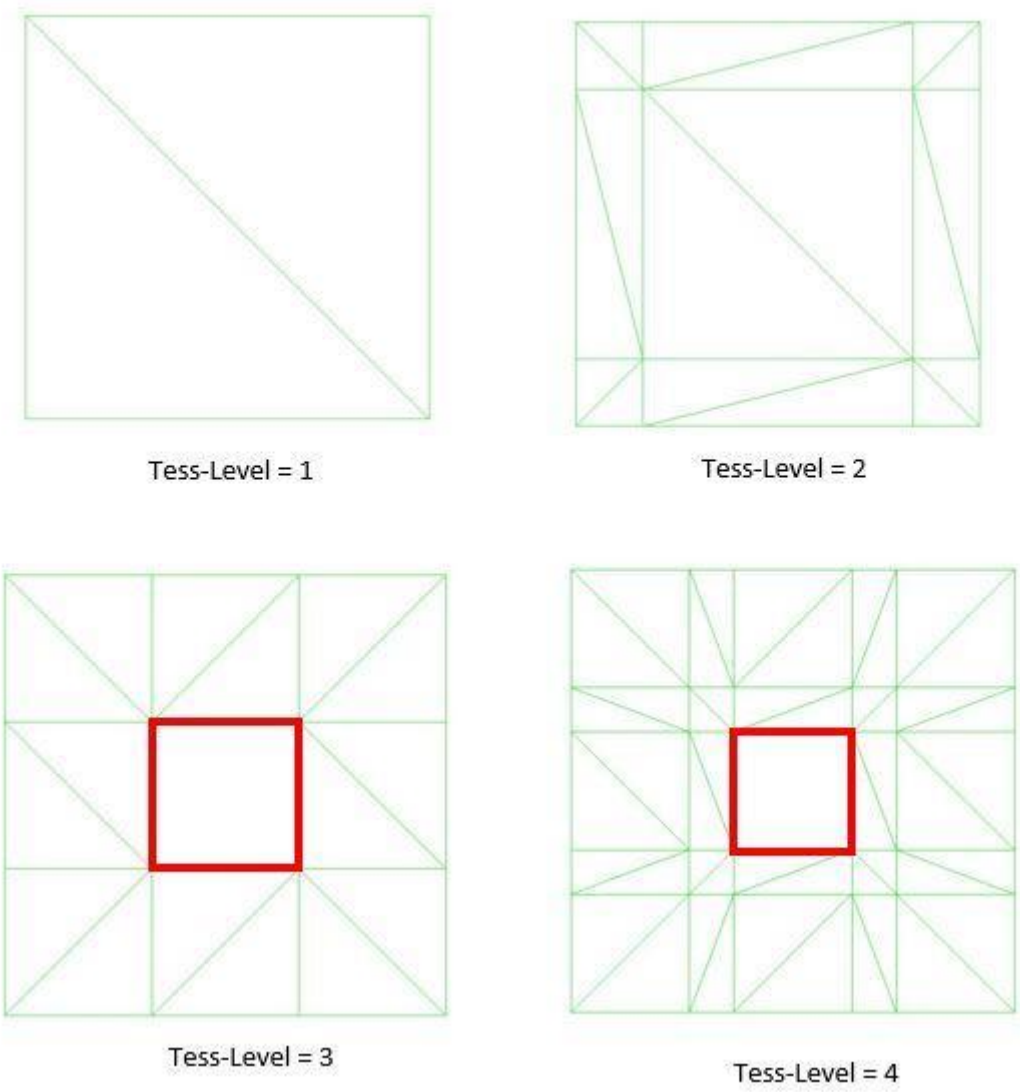

Abbildung 4-5 Entstehung neuer Primitive bei Quads, Rotes Viereck = Quad um Mittelpunkt

4.2. Displacement-Mapping

Beim Displacement-Mapping handelt es sich um ein State-of-the-Art Verfahren. Im Gegensatz zu den bisher beschriebenen Verfahren erzeugt das Displacement-Mapping keine Illusion von Tiefe. Das Prinzip des Displacement-Mapping liegt darin, anhand einer Mikrostruktur (Textur) die Makrostruktur zu verändern beziehungsweise zu verfeinern, um so eine Mesostruktur zu erzeugen.

Beim Bump-Mapping werden durch die Veränderung der Oberflächennormale im Grunde imaginäre Vertices verschoben, anhand dessen dann die Beleuchtungsberechnung durchgeführt wird und eine Illusion entsteht. Beim Displacement-Mapping findet hingegen eine Verschiebung von echten Vertices statt. Jedoch müssen diese Vertices erst erzeugt werden, also die Makrostruktur (das Gitternetz) des Objektes muss verfeinert werden. Die Erzeugung dieser neuen Vertices geschieht mithilfe der Hardwaretessellation.

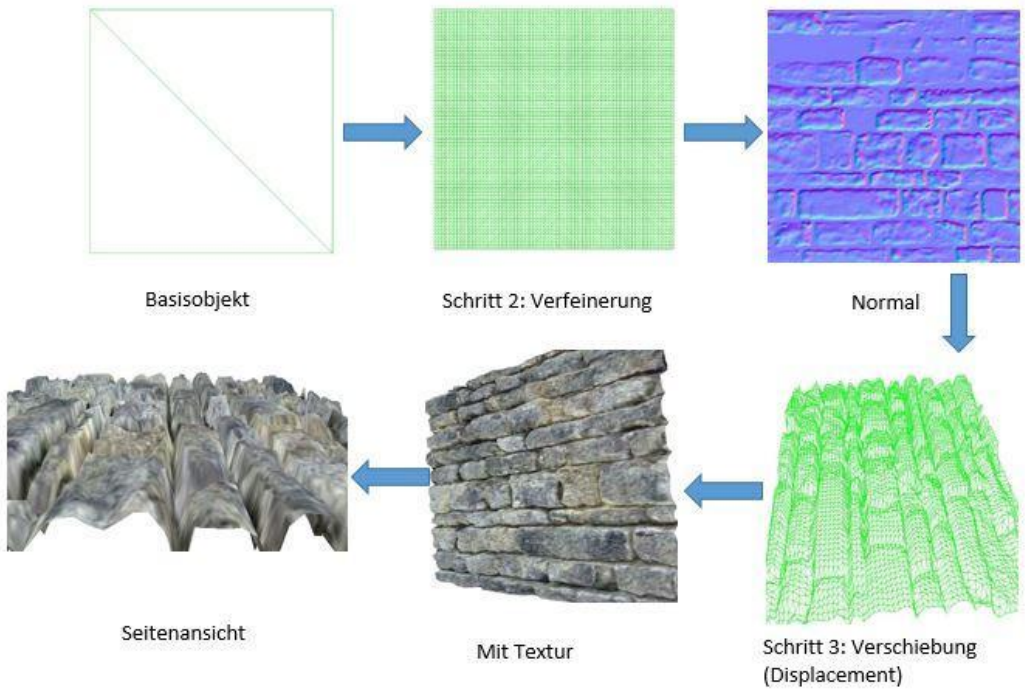

Abbildung 4-6 Ablauf des Displacement-Mappings

Die verschiedenen Schritte des Displacement-Mappings sind folgende:

1. Zunächst wird eine Height-Map benötigt. Entweder als Graustufenbild oder als Normalmap. Verwendet wird hier eine Normalmap, in deren Alpha-Kanal die Height-Map gespeichert ist. Die Erstellung der Normalmap erfolgt anhand der Textur die auf das Objekt gemapped werden soll.

2. In der Theorie besteht der zweite Schritt darin, die Height-Map auf das Objekt zu legen, und das Gitternetz dann im dritten Schritt an den Stellen zu verfeinern an denen Vertices verschoben werden müssten. In der Praxis jedoch besteht der zweite Schritt darin, dass Gitternetz zu verfeinern. Zum Beispiel durch Tessellation.
3. Die Vertices werden mithilfe der Height-Map verschoben und die Oberflächennormalen mit den Werten aus der Normalmap verrechnet und so neu ausgerichtet. Die Verwendung der Normalmap-Werte ist optional, sie führt jedoch zu optisch besseren Ergebnissen.
4. Die Textur wird auf das Objekt gelegt.

Wie in Abbildung 5-6 zu sehen ist, stellt das Displacement-Mapping eine gute Verbesserung dar, vor allem bei flachem Betrachtungswinkel. Was darauf begründet ist, dass die Geometrie des Objektes tatsächlich verändert wird. Dadurch wird sowohl die Silhouette als auch der Schattenwurf des Objektes beeinflusst.
In Bezug auf die Performance bietet das Displacement-Mapping gute Werte (siehe Abschnitt 6.).

4.3. Implementierung

4.3.1. Tessellation-Control-Shader

```
1.  layout(vertices = 3) out;
2.  in vec3 vPosition[];
3.  out vec3 tcPosition[];
4.  uniform float m_InnerTess;
5.  uniform float m_OuterTess;
6.  void main(){
7.  tcPosition[gl_InvocationID] = vPosition[gl_InvocationID];
8.  if (gl_InvocationID == 0) {
9.  gl_TessLevelInner[0] = m_InnerTess;
10. gl_TessLevelOuter[0] = m_OuterTess;
11. gl_TessLevelOuter[1] = m_OuterTess;
12. gl_TessLevelOuter[2] = m_OuterTess;
13. }
14. }
```

Abbildung 4-7 Programmcode eines simplen Control-Shaders

Dies ist der TCS für ein Triangle (Abb. 5-7). In Zeile 1 wird die Größe des Patches festgelegt. Anschließend werden Ein- und Ausgangsattribute angegeben. In diesem Fall wird weder die Normale des Patches, noch die Texturkoordinate verwendet, ausschließlich die Position des Vertex.
In Zeile 9-12 werden die Tessellation-Level festgelegt. Die Werte werden von der Anwendung an den TCS übergeben.

Für die Bestimmung der Level können beliebige Algorithmen verwendet werden. Im Anhang befindet sich eine Implementierung die das Level anhand der Entfernung bestimmt:

- Je höher die Distanz zwischen Kamera und Objekt, desto geringer das Level und damit desto niedriger die Detailstufe.

4.3.2. Tessellation-Evaluation-Shader

Folgender TES implementiert das Displacement-Mapping unter der Verwendung von Triangles (Abb. 5-8). Er ist nicht der zugehörige TES zum oben vorgestellten TCS.
Im TES wird zunächst angegeben welche Art von Primitiven der Tessellator weitergibt und in Zeile 2-7 die Ein- und Ausgabeattribute. Anschließend werden in Zeile 16-18 die Koordinaten umgerechnet. In Zeile 19-21 erfolgt das Displacement, dazu wird zunächst die Höhe und die Normale aus der Normalmap ausgelesen und dann mit den Werten des Vertex verrechnet, sodass sich die Position verändert. Die neue Position des Vertex wird dann an den Fragment-Shader weitergegeben (Zeile 23).

```
1.  layout(triangles, equal_spacing, ccw) in;
2.  in vec3 tcPosition[];
3.  in vec2 tcTexCoord[];
4.  in vec3 tcNormal[];

5.  out vec3 tePosition;
6.  out vec2 teTexCoord;
7.  out vec3 teNormal;

8.  uniform sampler2D m_NormalDepthMap;
9.  uniform float m_Influence;
10. uniform mat4 g_ViewProjectionMatrix;

11. vec2 interpolate2D(vec2 v0, vec2 v1, vec2 v2){
12. return vec2(gl_TessCoord.x) * v0 + vec2(gl_TessCoord.y) * v1 +
    vec2(gl_TessCoord.z) * v2;}

13. vec3 interpolate3D(vec3 v0, vec3 v1, vec3 v2){
14. return vec3(gl_TessCoord.x) * v0 + vec3(gl_TessCoord.y) * v1 +
    vec3(gl_TessCoord.z) * v2;}

15. void main(){
16. teTexCoord = interpolate2D(tcTexCoord[0], tcTexCoord[1],
    tcTexCoord[2]);
17. teNormal = interpolate3D(tcNormal[0], tcNormal[1],
    tcNormal[2]);
18. tePosition = interpolate3D(tcPosition[0], tcPosition[1],
    tcPosition[2]);

19. vec4 nd = texture2D(m_NormalDepthMap, teTexCoord);
20. vec3 displacement = tcNormal[0];
21. tePosition += displacement * m_Influence * nd.a;
22. teNormal = vec3(nd.x, nd.z, nd.y) * 2.0 - 1.0;

23. gl_Position = g_ViewProjectionMatrix * vec4(tePosition, 1.0);}
```

Abbildung 4-8 TES Implementierung für Displacement-Mapping

5. Performancevergleiche

5.1. Vergleich - Mappingverfahren

Im Folgenden werden die bisher beschriebenen Verfahren in ihrer Performance und dem Aussehen kurz verglichen. Für die Ausführung sind die vorimplementierten Shader von JMonkey verwendet worden.

JMonkey enthält ein “Statistic-Display” welches unter anderem die Framerate (Anzahl der Bilder pro Sekunde) anzeigt, da die Framerate jedoch permanent schwankt, wurden Durchschnittswerte über einem Zeitraum von 10 Sekunden genommen. Die Verfahren wurden auf ein Objekt bestehend aus zwei Triangles angewendet.

Tabelle 1

Verfahren	Frames per seconds (Durchschnittswert)
Texture-Mapping	3550
Bump-Mapping	2800
Normal-Mapping	2800
Parallax-Mapping	2400
Steep-Parallax-Mapping	1760
Displacement-Mapping	3300

Anhand der Werte aus Tabelle 1, lässt sich nachvollziehen, wieso Normal-Mapping nachwievor in großem Ausmaß verwendet wird. Es bietet im Vergleich zum Textur-Mapping eine starke Verbesserung.
Mit dem Parallax-Mapping wird dieses Ergebnis lediglich in Bezug auf die Seitenansicht verbessert (siehe Abb. 6-1 und 6-2). Daher wird das Normal-Mapping in der Regel als Basis für das Echtzeitrendering von Spielen verwendet und es gibt, sofern vom Hersteller implementiert, die Möglichkeit andere Verfahren wie das Parallax-Mapping zu aktivieren um den Detailgrad zu erhöhen.

Das Bump-Mapping und Normal-Mapping erreichen etwa gleiche Performancewerte, da in beiden Verfahren für jedes Pixel eine Berechnung durchgeführt wird, lediglich mit unterschiedlichen Werten (Normalmap/Graustufenbild).

Abbildung 5-1 Vergleich Seitenansicht

Abbildung 5-2 Vergleich Frontansicht

Das Displacement-Mapping bietet nicht nur ein gutes Bild, sondern auch einen sehr guten Performancewert. Dieser gute Wert kommt daher, dass es sich hier nur um ein einzelnes Objekt mit wenigen Triangles handelt. Daher besteht der Unterschied zwischen dem Texture-Mapping und dem Displacement-Mapping nur darin, dass beim Displacement-Mapping mehr Triangles angezeigt werden. Dies wiederrum bedeutet, je höher die Differenz der Anzahl der Triangles, desto schlechter die Performance.

5.2. Vergleich - Terrain

Mit einer dynamischen Anpassung des Tessellation-Levels und der effizienten Erzeugung von neuen Primitiven, bietet sich die Hardware-Tessellation auch als Verfahren zur Darstellung von Terrains an. Wie sie dabei im Vergleich zu anderen Verfahren abschneidet, wird im Folgenden untersucht.

Terrains eignen sich gut für einen Performancevergleich, da sich über ein Terrain ein großes Objekt mit vielen Details und Primitiven ohne großen Aufwand erzeugen lässt. Des Weiteren sollen Erhebungen direkt vor dem Betrachter und in größerer Entfernung angezeigt werden. Allerdings brauchen die entfernten Erhebungen nicht so detailgetreu, wie jene direkt vor dem Betrachter, dargestellt werden. Diese Bedingungen erfordern eine dynamische Einstellung des Detailgrades. Stellt man alles auf einem Detailgrad dar, kann die Performance stark darunter leiden.

Für die Erzeugung des Terrains wird nur eine Height-Map benötigt.

Es werden drei Verfahren miteinander verglichen:

1. Geometry-Shader
2. JME Implementierung für Terrains (QuadTree-Verfahren)
3. Displacement-Mapping mit Tessellation

Der Vergleich erfolgt in Bezug auf folgende Punkte:

- Zeitdauer vom Programmstart bis zur Anzeige
- Framerate
- Anzahl der Primitive

Die Werte in Tabelle 2 sind keine exakten Messungen, da sowohl die Framerate als auch die Anzahl der angezeigten Primitive abhängig ist von der Position der Kamera. Die verwendeten Werte entsprechen der jeweils gleichen Kameraposition, dabei erfolgte die Orientierung anhand eines charakteristischen Merkmals des Terrains. Des Weiteren ist der Wert für die Framerate zu der entsprechenden Angabe der Triangles entnommen worden. Daher erfolgt die Angabe der Werte in einem Durchschnittswert. Teilweise unterscheidet sich der Wert für die Framerate mit denen die auf den kommenden Abbildungen zu sehen sind. Der Unterschied wird dadurch verursacht, dass zum Zeitpunkt der Aufnahme das Programm nicht das aktive Fenster war und die Framerate somit stark gesunken ist.

Die Anzeige erfolgte jeweils im Wireframe-Modus.

Tabelle 2

Verfahren	Renderpass (Sekunde)	Framerate (fps)	Anzahl der Primitive (Triangles)
Geometry-Shader	<1	~70	180.000
QuadTree (ohne LoD)	~3	~68	~3.6 Mio
QuadTree (mit LoD)	~3	~615	~96.000
Tessellation(ohne LoD)	~1	~52	~5.5 Mio
Tessellation(mit LoD)	~1	~606	~235.000

5.2.1. Der Geometry-Shader

Als Implementierung für den Geometry-Shader wurde eine Variante verwendet, die der Tessellation entsprechen soll(siehe Anhang). Sie erzeugt so viele Primitive wie möglich und verschiebt die Vertices anschließend. Es wird kein Terrain erzeugt.

Die Anzeige der Ausgabe erfolgt nahezu instantan. Es wurde eine Basisfläche bestehend aus 2096 Triangles auf der CPU erstellt. Durch den Geometry-Shader ist die Anzahl der Triangles auf 180.000 angestiegen. Die Framerate ist auf 70 pro Sekunde gefallen, was für 180.000 Triangles ein schlechter Wert ist.
Der Geometry-Shader ist nicht dafür geeignet beziehungsweise vorgesehen, so viele Primitive zu erzeugen.

5.2.2. QuadTree

Hier wurde das QuadTree-Verfahren für Terrains von JME verwendet.
Es wurde ein Terrain mit einer Fläche von 2048x2048 und einer Patchsize von 64 erstellt. Das bedeutet die Grundstruktur besteht aus 64x64 Quads.
Im ersten Durchlauf wurde das Verfahren ohne LoD ausgeführt, der Detailgrad hat sich dementsprechend nicht geändert. Die Zeit bis das Rendering abgeschlossen ist, beträgt etwa drei Sekunden. Sie ist damit länger als beim Geometry-Shader, da die Quads zunächst auf der CPU erstellt und berechnet werden müssen. Erst dann können sie an die GPU weitergeleitet werden.

Mit ausgeschalteten LoD erkennt man bereits einen Leistungsunterschied im Vergleich zum Geometry-Shader, da dieser jedoch nicht für Terrains ausgelegt ist, ist der Vergleich unfair. Jedoch ist hier der Performanceeinbruch des Geometry-Shaders gut zu sehen. Bei nahezu gleicher Framerate ist der Unterschied in der Anzahl der Triangles enorm (Faktor 20).

Im nächsten Fall wurde das LoD eingeschaltet. Da die Grundstruktur identisch mit der vom ausgeschalteten LoD ist, ist auch die Zeit bis zur Ausgabe die gleiche. Mit LoD werden wesentlich weniger Triangles angezeigt, da der QuadTree in Abhängigkeit von der Entfernung zur Kamera die Quads unterteilt, dies ist in Abbildung 6-3 zu sehen. Das führt zu einem enormen Ansprung in der Leistung, jedoch auch zu einem Detailverlust.

5.2.3. Tessellation

Auch hier wurde wieder eine Fläche der Größe 2048x2048 erstellt, jedoch wurde hier eine Unterteilung der Grundfläche in 32x32 Quads vorgenommen.
Das Tessellation-Level steht im ersten Fall jeweils mit 64 auf dem Maximum. Die Zeitdauer bis zur Bildausgabe ist kürzer im Vergleich zum QuadTree, da auf der CPU wesentlich weniger Objekte erzeugt werden. Das Verhältnis zwischen Anzahl der Triangles und der Framerate ist nahezu gleichwertig mit dem des QuadTrees ohne LoD. Die Qualität der Objekte ist ebenfalls gleichwertig.
Der Unterschied in der Struktur kommt daher, dass die JME Implementierung für Terrains die Höhe anders errechnet.

Die Renderzeit ist mit eingeschaltetem LoD die gleiche wie zuvor, da die Anzahl der Objekte die auf der CPU erstellt werden, identisch sind. Bei eingeschalteten LoD wurde solange der LoD-Faktor verändert, bis entweder eine gleichwertige Anzahl an Primitiven oder eine ähnliche Framerate auftrat. Wie in der Tabelle zu sehen ist, stellt die Tessellation mehr Triangles bei nahezu gleicher Framerate dar. Des Weiteren stehen je nach Implementierung wesentlich mehr Detailstufen zur Verfügung, wodurch die Übergänge weicher sind (siehe Abbildung 6-3).

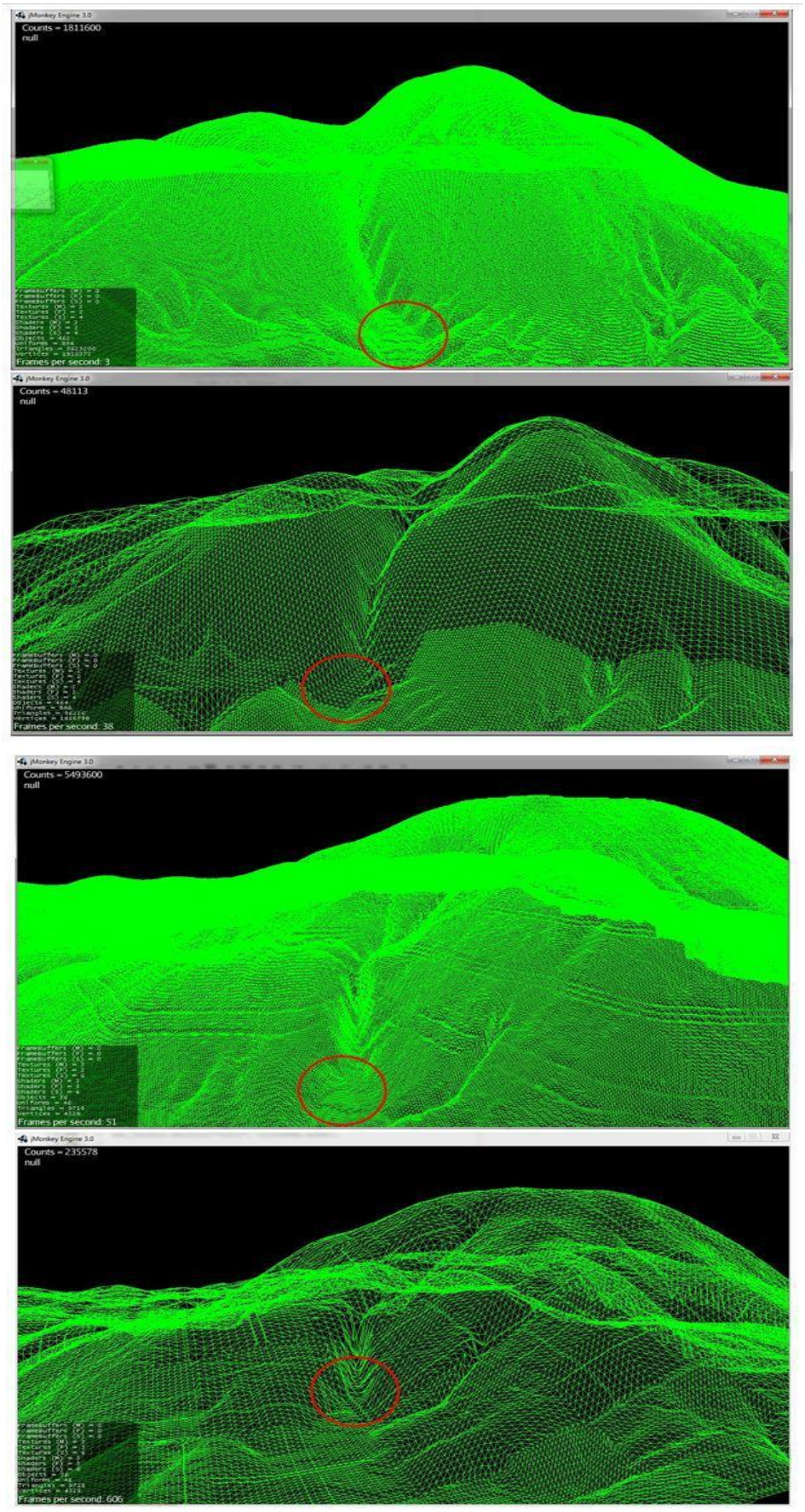

Abbildung 5-3 von oben nach unten: QuadTree ohne LoD, QuadTree mit LoD, Tessellation ohne LoD, Tessellation mit LoD

6. Programm

6.1. Konzept

Das Programm soll eine visuelle Einführung in das Thema Displacement-Mapping mit Tessellation ermöglichen. Dazu soll es die Möglichkeit geben sich den Vorgang der Tessellation mit und ohne Displacement veranschaulichen zu können. Über eine GUI soll zwischen den Beispielen gewechselt werden können und die Veränderung des Tessellation-Levels ermöglichen.
In Abbildung 7-1 ist eine Konzeptzeichnung der Oberfläche zu sehen.

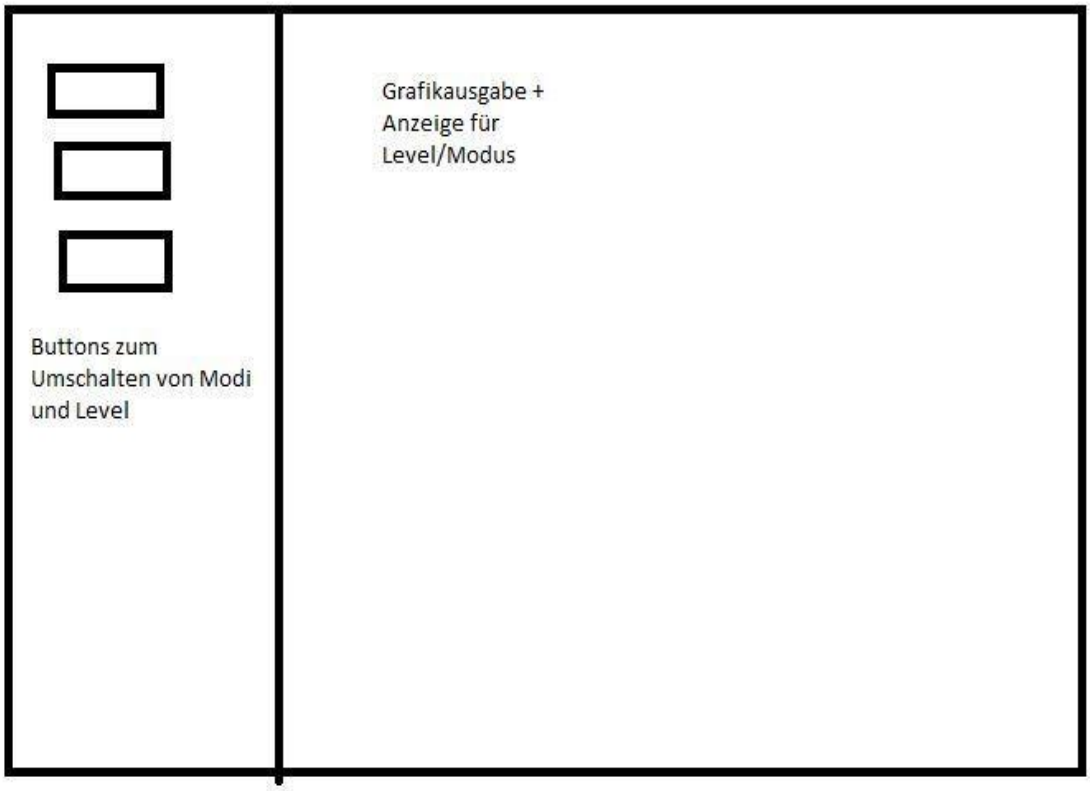

Abbildung 6-1 Konzeptzeichnung der Oberfläche

6.2. Umsetzung

Das Programm wurde vollständig in der IDE von JMonkey umgesetzt. Abbildung 7-2 stellt die finale Oberfläche dar.

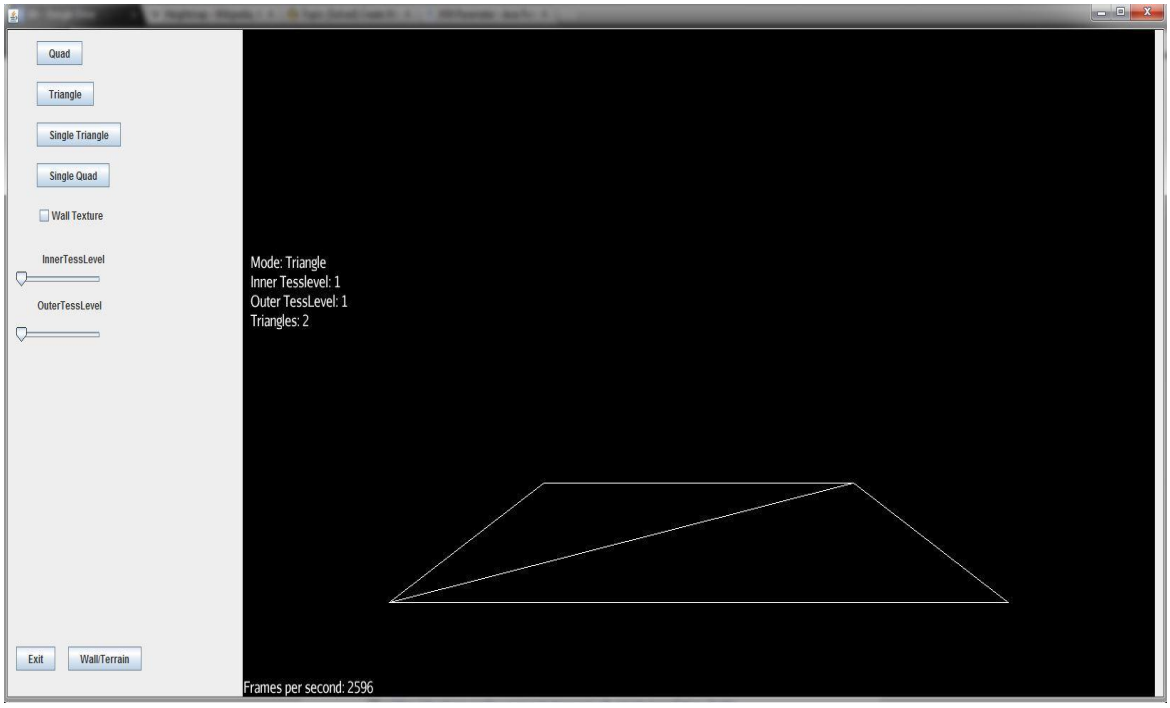

Abbildung 6-2 Programmoberfläche

Zunächst wurde eine Applikation erstellt, welche die Objekte und die entsprechenden Materialien erzeugt. In JME werden Shader über Materialdefinitionen verwendet und implementiert. Anschließend wurde die Applikation um ein JFrame erweitert, welches etwaige Actionbuttons zur interaktiven Benutzung zur Verfügung stellt.

Zwar bietet JME einen eigenen Trianglezähler, allerdings zählt dieser nur die Triangles die vorhanden sind, bevor das Objekt an die GPU weitergeleitet wird. Was dazu führt, dass bei der Verwendung von Tessellation sich der Zählerstand nicht ändert, da die neuen Triangles auf der GPU erzeugt werden.

Mit der Einführung des Geometry-Shader stellte OpenGL eine Funktion zur Verfügung, die durch GPU erzeugte Primitive zählt. Der Zähler wird dann über eine Query zusammen mit der Funktion “GL_PRIMITVES_GENERATED” implementiert.

Das Programm stellt folgende Features zur Verfügung gestellt:

- Tessellation ohne Displacement anhand eines einzelnen Quads und Triangles
- Tessellation mit Displacement anhand eines Objektes bestehend aus Triangles oder Quads
- Zwei verschiedene Displacementbeispiele → Terrain und Wand, durch einen Button kann zwischen beiden Modi gewechselt werden
- Dynamische Einstellung der Tessellation-Levels über Slidebars
- Anzeige einer Textur beim Modus “Wall” über ein Kontrollkästchen
- Eine Anzeige die Informationen über das derzeitige Tessellation-Level, den aktuellen Modus, sowie die Anzahl der Triangles des Objektes und die aktuelle Framerate angibt

7. Fazit

Die Ausgangsfrage, ob Displacement-Mapping mit Tessellation, ein positiver Fortschritt sei, ist eindeutig mit Ja zu beantworten.

Der graphischen Datenverarbeitung steht damit ein starkes Werkzeug für eine gute Entwicklung in Richtung realitätsnaher Darstellung zur Verfügung.

Es können wesentlich mehr Primitive erzeugt werden ohne die CPU zusätzlich zu belasten. Durch die hohe Anzahl an Tessellation-Level kann der Wechsel zwischen unterschiedlichen Detailgraden weicher dargestellt werden. Man erhält die Möglichkeit Objekte noch sparsamer auf der CPU zu erstellen und sie somit zu entlasten, da entsprechende Qualität dann durch die Tessellation erzeugt werden kann.

Nachteile sind jedoch, dass die Tessellation zum einem nur auf Grafikkarten zur Verfügung steht die DirectX11 unterstützen. Des Weiteren bedeutet jedoch eine höhere Anzahl an Primitiven auch eine Senkung der Performance und somit der Framerate.

Die folgende Abbildung zeigt den Vergleich einer Szene, wie sie in einem Spiel vorkommen kann, einmal mit und ohne Tessellation. Erzeugt wurden die Bilder mit dem Unigine Heaven Benchmark [15]. Hier wird der Effekt von Tessellation sehr gut deutlich.
Berücksichtigt man, dass die hier verwendete Grafikkarte aus eine der ersten Generationen der Tessellation-Grafikkarten ist, und bereits drei Jahre alt ist (Erscheinungsdatum: Juli 2010), sind selbst 30 Fps für solch eine Qualität ein guter Wert.
Innerhalb von drei Jahren sind viele neue und bessere Grafikkarten erschienen, die noch mehr Leistung zur Verfügung stellen um das Potential der Tessellation noch besser ausschöpfen zu können.

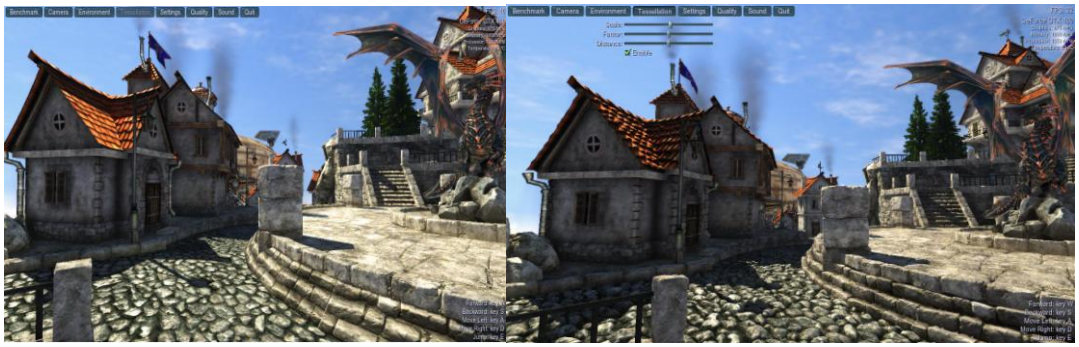

Abbildung 7-1 Links: ohne Tessellation Rechts: mit Tesselation

[15] (Unigine, 2013)

8. Abbildungsverzeichnis

9. Literaturverzeichnis

Blinn, J. (August 1978). Simulation of Wrinkled Surfaces. *Computer Graphics, Vol. 12(3)* , S. 286-292.

Bunnell, M. (2005). Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. In *GPU Gems 2* (S. 109-122).

Christian Dick, J. K. (2009). GPU Ray-Casting for Scalable Terrain Rendering.

Hart, E. (10. Januar 2013). *Nvidia Developer Zone*. Abgerufen am September 2013 von https://developer.nvidia.com/content/opengl-sdk-simple-tessellation-shader

Jonathan Cohen, M. O. (1998). Appearance-Preserving Simplification.

László Szirmay-Kalos, T. U. (2006). Displacement Mapping on the GPU - State of the Art. *Volume 25* , S. 1-24.

Loop, M. N. (2013). Analytic Displacement Mapping using Hardware Tessellation.

McGuire, M. M. (2005). Steep Parallax Mapping.

PasteBin. (Mai 2012). Von http://pastebin.com/mNQnTv5z abgerufen

Strugar, F. (11. Juli 2010). Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD).

Tomomichi Kaneko, T. T. (Dezember 2001). Detailed Shape Representation with Parallax Mapping.

Unigine. (2. Dezember 2013). Abgerufen am September 2013 von http://unigine.com/products/heaven/

Yotam Livny, Z. K.-S. (2007). Seamless Patches for GPU-Based Terrain Rendering.

10. Anhang

```
//TCS Implementierung - Kameraentfernung

layout(vertices = 4) out;

in vec2 vTexCoord[];
in vec3 vNormal[];
in vec3 vPosition[];

out vec2 tcTexCoord[];
out vec3 tcNormal[];
out vec3 tcPosition[];

uniform float m_LodFactor;      //Skalierungsfaktor für die LoD,
uniform mat4 g_ViewProjectionMatrix;  // je höher desto mehr
uniform vec4 g_ViewPort;              // wird skaliert


vec4 world2view(vec3 v) {
    vec4 v2 = g_ViewProjectionMatrix * vec4(v, 1.0);
    v2 /= v2.w;
    return v2;
}

vec2 view2screen(vec4 v) {
    return vec2(v.x * g_ViewPort.z * 0.5, v.y * g_ViewPort.w *
0.5);
}

bool offscreen(vec4 vertex){
    if(vertex.z < -0.5)
        return true;
    return any(lessThan(vertex.xy, vec2(-1.7)) ||
greaterThan(vertex.xy, vec2(1.7)));
}

float level(vec2 v0, vec2 v1) {
     return clamp(distance(v0, v1)/m_LodFactor, 1, 64);
}

void main()
{
    tcPosition[gl_InvocationID] = vPosition[gl_InvocationID];
     tcTexCoord[gl_InvocationID] = vTexCoord[gl_InvocationID];
     tcNormal[gl_InvocationID] = vNormal[gl_InvocationID];

    if (gl_InvocationID == 0) {
     vec4 s0 = world2view(vPosition[0]);
     vec4 s1 = world2view(vPosition[1]);
     vec4 s2 = world2view(vPosition[2]);
     vec4 s3 = world2view(vPosition[3]);

      if(offscreen(s0) && offscreen(s1) && offscreen(s2) &&
offscreen(s3)) { //sind die Vertices außerhalb des Sichtfeldes,
            gl_TessLevelInner[0] = 0; //ausblenden
```

```
            gl_TessLevelInner[1] = 0;
            gl_TessLevelOuter[0] = 0;
            gl_TessLevelOuter[1] = 0;
            gl_TessLevelOuter[2] = 0;
            gl_TessLevelOuter[3] = 0;
          } else {
            vec2 ss0 = view2screen(s0);
            vec2 ss1 = view2screen(s1);
            vec2 ss2 = view2screen(s2);
            vec2 ss3 = view2screen(s3);

                  float e0 = level(ss1, ss2);//anhand der Distanz
                  float e1 = level(ss0, ss1);// und dem LoDFactor
                  float e2 = level(ss3, ss0);// wird das Level
                  float e3 = level(ss2, ss3); //bestimmt

                  gl_TessLevelInner[0] = mix(e1, e2, 0.5);
                  gl_TessLevelInner[1] = mix(e0, e3, 0.5);
                  gl_TessLevelOuter[0] = e0;
                  gl_TessLevelOuter[1] = e1;
                  gl_TessLevelOuter[2] = e2;
                  gl_TessLevelOuter[3] = e3;
            }
    }
}

// es kann mit den Werten noch etwas rumgespielt werden, um zum
// Beispiel die Ausblendung etwas besser zu gestalten, damit die
// Objekte nicht zu früh ausgeblendet werden
```