TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Medieninformatiker

Effiziente Datenübertragung von Modellen und Texturen für die Verwendung in WebGL

Stefan Wagner
(Geboren am 10. Dezember 1983 in Pirna)

Betreuer:
Prof. Dr. rer. nat. Stefan Gumhold (TU Dresden)
Dipl.-Medieninf. Andreas Stahl (TU Dresden)
Michael Kopietz (Crytek GmbH)

Dresden, 1. Oktober 2012

Aufgabenstellung

Durch WebGL ist es möglich, grafikintensive Anwendungen, zum Beispiel Spiele und 3D-Objektkataloge, über den Webbrowser anzubieten. Diese benötigen allerdings oft komplexe 3D-Modelle und Texturen, welche zu langen Ladezeiten bei der Übertragung dieser führen. Der Anwender ist es jedoch aus dem Webumfeld gewohnt, dass Webanwendungen sofort oder zumindest in sehr kurzer Zeit zur Verfügung stehen.

Ziel dieser Diplomarbeit ist die Evaluierung von Kompressionstechniken, um 3D-Modelle und Texturen effektiv zu übertragen. Die dabei untersuchten Techniken sollen insbesondere im Rahmen von WebGL Anwendbarkeit finden. Alle Methoden der Datenübertragung sollen an einfachen und komplexen Objekten und verschiedenen Szenen mit wenigen und vielen dieser evaluiert werden. Für wiederholte Aufrufe sollen die Modelle im Cache des Anwenders gespeichert werden. Dafür sollen existierende HTML5 Techniken bezüglich ihrer Verfügbarkeit und Eignung für 3D-Daten evaluiert werden.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Effiziente Datenübertragung von Modellen und Texturen für die Verwendung in WebGL

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 1. Oktober 2012

Stefan Wagner

Kurzfassung

Diese Arbeit evaluiert Methoden und Formate, welche für die Kompression der Geometriedaten verwendet werden können. Einige von ihnen erreichen eine Kompressionsrate von 1 zu 2 gegenüber dem mit GZIP komprimierten Binärformat. Diese lassen sich für eine progressive Vorschau während des Streaming nutzen, indem das Modell und seine Level of Detail (LOD) Stufen komprimiert und einzeln übertragen werden. Um eine Vorschau ohne zusätzliche Daten zu erreichen, wird erläutert, wie sich das Ursprungsmodell durch Progressive Meshes nach Hoppe et al. während der Übertragung beim Client zusammensetzen lässt. Ebenfalls wird dafür der Progressive-Modus von PNG und JPEG über die Technik Image Geometry evaluiert.

Da die Dekodierungsgeschwindigkeit der Formate bei sehr schnellen Verbindungsgeschwindigkeiten eine wichtige Rolle spielt, werden diese Zeiten gegenübergestellt. Die Texturen belegen einen großen Anteil des Speicherplatzes, sind aber nicht ohne weiteren Qualitätsverlust komprimierbar. Deshalb wird alternativ eine Übertragung über den progressiven Modus von JPEG evaluiert. Im Abschluss werden die gegebenen Möglichkeiten untersucht, um die Modelle persistent oder temporär im Cache des Nutzers abzulegen.

Abstract

This thesis evaluates methods and formats to compress 3D models. Some of those formats achive a compression ratio of 1 to 2 compared to the GZIP binary format. Those can be used to compress the model and the individual level of details (LOD). The LODs can later be used to generate a progressive preview, while the user is streaming the model. The progressive mesh technique from Hoppe et al. is also evaluated to achieve such a preview without sending additional data. The models can also be streamed with the progressive mode of JPEG and PNG by using the image geometry technique.

The decode time is a large part of the overall time if the user has a very fast connection, therefore those decode times where measured and compared. The size of the textures is larger than the size of the compressed models, but it isn't possible to compress them any further without decreasing the quality, therefore a streaming method for the textures which uses progressive JPEGs is evaluated. The last part evaluates the given possibilities to store the downloaded 3D models and textures at the client.

Inhaltsverzeichnis

1 Einleitung **5**

2 Verwandte Arbeiten **7**
2.1 Progessive Meshes . . . 7
2.2 Komprimierung von 3D-Geometriedaten . . . 7

3 Modellformate in WebGL **9**
3.1 Modellformate . . . 9
3.1.1 Crytek Geometry Format . . . 10
3.1.2 Wavefront Object Format . . . 11
3.2 JSON . . . 11
3.3 Testmodelle . . . 12

4 Kompression **17**
4.1 Basistechniken . . . 17
4.1.1 Delta-Kodierung . . . 17
4.1.2 Huffmann-Kodierung . . . 17
4.1.3 LZ77 . . . 18
4.1.4 Deflate . . . 19
4.2 HTTP-Kompression . . . 19
4.3 Sortierung der Daten . . . 20
4.3.1 Cache-Optimierung . . . 20
4.3.2 Attribute versetzt abspeichern (Element Interleaving) . . . 21
4.3.3 Bytes versetzt abspeichern (Byte Interleaving) . . . 22
4.4 Quantisierung . . . 23

5 Progressive Meshes **27**
5.1 Zerlegung des Meshes . . . 27
5.2 Datenstruktur . . . 28
5.3 Half Edge Collapse . . . 28
5.4 Vertex Split . . . 29
5.5 Bewertung . . . 31

6 Komprimierung von 3D-Modellen **35**
6.1 Schrittweise Komprimierung . . . 35
6.1.1 Schritt 1: Optimierung des Index nach Forsyth . . . 36
6.1.2 Schritt 2: Delta-Kodierung des Index . . . 36
6.1.3 Schritt 3: Attribute quantisieren und als 16 bit Short-Variable abspeichern . . . 37
6.1.4 Schritt 4: Versetztes Abspeichern der Attribute . . . 37
6.1.5 Schritt 5: Transformation der Attribute mittels Delta-Kodierung . . . 37
6.1.6 Bewertung . . . 37
6.2 OpenCTM . . . 41
6.2.1 Aufbau des OpenCTM Formates . . . 41
6.2.2 RAW Modus . . . 42
6.2.3 MG1 Kompression . . . 42

6.2.4 MG2 Kompression . . . 43
6.2.5 Bewertung . . . 44
6.3 WebGL Loader . . . 45
6.3.1 UTF-8 zur Speicherung binärer Daten . . . 45
6.3.2 Kompressionsalgorithmus . . . 47
6.3.3 Bewertung . . . 47

7 Image Geometry 49
7.1 Algorithmus . . . 49
7.1.1 Kodierung . . . 49
7.1.2 Dekodierung . . . 50
7.2 Komprimierung mit dem JPEG und PNG Format . . . 50
7.3 Progressive Vorschau des Modells über Streaming . . . 51
7.4 Streaming Animations . . . 56
7.5 Bewertung . . . 58

8 Download- und Dekodierungsgeschwindigkeit 63
8.1 Kompressionsverhältnis . . . 63
8.2 Dekodierungsgeschwindigkeit . . . 65
8.3 Gesamtzeit zur Anzeige der Modelle . . . 67

9 Texturen 69
9.1 Textur-Streaming . . . 69
9.2 Bewertung . . . 71

10 Caching 73
10.1 Browser Cache . . . 73
10.2 Application Cache . . . 75
10.2.1 Bewertung . . . 75
10.3 File System API . . . 76
10.3.1 Bewertung . . . 77
10.4 Web Storage . . . 77
10.4.1 Bewertung . . . 78
10.5 Web SQL Database . . . 78
10.5.1 Bewertung . . . 79
10.6 IndexedDB . . . 79
10.6.1 Bewertung . . . 80
10.7 Einsatz der Caching Schnittstellen . . . 80

11 Fazit und Ausblick 83

Literaturverzeichnis 85

Abbildungsverzeichnis 89

Tabellenverzeichnis 91

Listings 93

A Strukturierung der Vertex Buffer Objects 95
A.0.1 Structure of Arrays . . . 95
A.0.2 Array of Structures . . . 95

B Zusatztabellen **97**

C CD Inhalt **99**

1 Einleitung

WebGL erlaubt es, 3D-Grafiken hardwarebeschleunigt ohne die Verwendung von externen Erweiterungen auf Internetseiten anzuzeigen. Notwendig ist dafür lediglich ein Browser, der den WebGL Standard unterstützt. Damit können komplexe Anwendungen, welche vorher nur auf dem Desktop möglich waren, direkt im Browser ausgeführt werden.

Mit komplexer werdenden 3D-Anwendungen steigt gleichzeitig die Komplexität der 3D-Szenen. Immer mehr 3D-Modelle und höher aufgelöste Texturen werden benötigt, um einen noch realistischeren Eindruck zu erschaffen. In modernen Spielen benötigt ein einzelnes Level oft mehr als 400 MB an reinen Geometriedaten und Texturen. In der Regel geht der Entwickler davon aus, dass sich alle Modelle und Texturen zur Laufzeit bereits auf der Festplatte oder zumindest der DVD des Anwenders befinden. Diese Annahme impliziert, dass die Daten vor dem Start einer 3D-Anwendung erst komplett vom Nutzer heruntergeladen werden müssen. Allerdings kollidiert dies mit der Erwartung des Nutzers, sobald er sich im Webumfeld befindet, wo er es gewohnt ist, dass sich Seiten sehr schnell öffnen und wenig Wartezeiten in Form von Ladebalken abverlangen. Im schlimmsten Fall müssen die Daten wiederholt heruntergeladen werden, sobald der Anwender die Seite nochmals besucht.

Das Ziel ist es somit, 3D-Modelle möglichst kompakt und effektiv zu übertragen, um so die Wartezeit des Nutzers auf ein Minimum zu reduzieren. Im besten Fall erhält der Nutzer bereits sehr früh ein Feedback in Form einer Vorschau des 3D-Modells. Dies kann zum Beispiel eine weniger detailliertere Version dessen sein. Diese kann, zum Beispiel während des Download, aus den bereits übertragenen Daten errechnet werden, ohne das zusätzliche Daten übertragen werden müssen. Ein anderer Ansatz übermittelt im Vorfeld gröber aufgelöste Modelle und schickt diese in aufsteigender Reihenfolge nacheinander. Damit entsteht zwar ein zusätzlicher Overhead, allerdings ist die Qualität der Vorschau meist wesentlich besser. In dieser Arbeit werden beide Ansätze mit verschiedenen Methoden evaluiert.

Damit ein erneutes Herunterladen der Daten beim Client möglichst entfällt, werden die Modelle im Cache gespeichert werden. Mit HTML5 ist es möglich verschiedene Schnittstellen für die Speicherung von Daten zu nutzen. Welche am besten zum Speichern der Daten geeignet ist, hängt vom jeweiligen Anwendungsfall an. Diese Arbeit analysiert die möglichen Schnittstellen und beschreibt, für welche Fälle diese geeignet sind.

Bei der Evaluation der hier vorgestellten Techniken sind folgende Anforderungen zu beachteten. Die Modelldaten stammen, damit verschiedene Fälle abgedeckt werden, aus zwei unterschiedlichen Anwendungsgebieten. Bei der Kompression der Modelle muss ein Kompromiss zwischen einer schnellen Dekodierungszeit und einem guten Kompressionsverhältnis abgewogen werden. Dabei empfiehlt es sich möglichst wenig der Dekodierung mittels JavaScript auszuführen, da die Geschwindigkeit andernfalls vom verwendeten Browser abhängig ist.

Diese Diplomarbeit wurde in Zusammenarbeit mit der Firma Crytek GmbH in Frankfurt am Main durchgeführt. Die verwendeten Techniken werden an 3D-Modellen der Firma erprobt, um somit eine realitätsnahe Einschätzung der Performance zu ermöglichen. Diese Arbeit geht davon aus, dass der Leser bereits grundlegende Kenntnisse in WebGL, JavaScript und HTML5 besitzt. Insbesondere sollte die Darstellung von 3D-Modellen mittels Vertex Buffer Objects (VBO) in WebGL, das Datenaustauschformat JSON und die Darstellung von Bildern mittels JavaScript und dem Canvas Element bereits bekannt sein.

2 Verwandte Arbeiten

2.1 Progessive Meshes

Hoppe et al. [Hop96] untersuchten in ihrer Arbeit über Progressive Meshes, wie 3D-Modelle stufenlos vereinfacht werden können. Der vorgestellte Algorithmus vereinfacht diese, indem schrittweise Kanten des Modells entfernt werden. Diese Vereinfachung wird protokolliert und kann anschließend in der entgegengesetzten Reihenfolge schrittweise durchgeführt werden. Dies soll es ermöglichen, sehr komplexe Szenen darzustellen, da die 3D-Modelle nur in ihrer vollen Auflösung angezeigt werden, wenn diese zum Beispiel nah am Sichtfeld des Nutzers platziert sind. Ebenfalls kann diese Technik für den progressiven Transfer von 3D-Modellen verwendet werden, indem die aufgezeichneten Vereinfachungen übertragen und anschließend wieder zusammengesetzt werden.

Lee et al. [ML10] und Fogel et al. [FCO12] untersuchten, wie Progressive Meshes im Webumfeld effektiv übertragen werden können. Sie verwendeten Metriken, wie die Verbindungsgeschwindigkeit, die Leistungsfähigkeit und den aktuellen Bildausschnitt des Clients, um die Übertragung zu optimieren. Dieser Ansatz wurde als Erweiterung für das Framework X3DOM [Beh11] implementiert, jedoch nicht veröffentlicht. Sawicki und Chaber [SB12] untersuchten ebenfalls, wie Progressive Meshes ohne den Einsatz von Erweiterungen übertragen werden können. Sie implementierten dabei einen prototypischen 3D-Modell-Viewer, welcher die Technik von Hoppe et al. verwendet, um ein 3D-Modell schrittweise zu übertragen. Die Übertragung des 3D-Modells ist in der prototypischen Implementation jedoch relativ langsam, da eine Optimierung der Daten fehlt. Die progressiven Zwischenstufen werden im Text-Format mit unnötigen Overhead versendet. Dazu kommt eine langsame Rendergeschwindigkeit des 3D-Modells, da der Renderer ebenfalls nicht weiter optimiert wurde.

In der vorliegenden Arbeit werden ebenfalls 3D-Modelle mit der Technik von Hoppe et al. progressive übertragen. Damit die Technik in einer vorhandenen Anwendung integriert werden kann, wurden keine zusätzlichen Erweiterungen verwendet und diese ohne die Verwendung eines externen Frameworks implementiert. Die Technik wird dabei für niedrig aufgelöste 3D-Modelle aus dem Spielebereich erprobt.

2.2 Komprimierung von 3D-Geometriedaten

Touma und Gotsman [TG98] führten in ihrer Arbeit eine Geometriedatenrepräsentation ein, mit welcher eine hohe Kompressionsrate erreicht werden kann. Die Komprimierung von Geometriedaten ist darüber hinaus ein Thema in weiteren Arbeiten, welche meist jedoch nicht für sämtliche 3D-Modelle anwendbar sind. Des Weiteren werden oft aufwendige Algorithmen verwendet, welche in JavaScript nur relativ langsam ausgeführt werden können. Dadurch entstehen hohe Dekodierungszeiten, welche unter Umständen den Vorteil der Komprimierung negieren.

In der vorliegenden Arbeit werden deshalb die Restriktionen, welche aus der Verwendung von WebGL und JavaScript entstehen, beachtet. Die verwendeten Algorithmen verwenden, wenn möglich, native Funktionen, wie zum Beispiel die Komprimierung des Browsers.

Die Geometriedaten eines 3D-Modells können in Texturen abgelegt werden. Dabei werden die Farbkanäle genutzt, um 3D-Vektoren abzuspeichern. Durch die Verwendung der nativen Dekodierungsfunktionen des Browsers für Bilder kann JavaScript größtenteils umgangen und somit die Dekodierungsgeschwindigkeit potentiell erhöht werden. Hoppe et al. [HP03] beschäftigte sich ebenfalls mit diesem Ansatz. Die

dabei verwendete Technik ist jedoch nicht für alle Modelle anwendbar. Insbesondere Modelle, welche aus mehreren kleineren Modellen zusammengesetzt sind, bereiten Probleme. Eine für die meisten 3D-Modelle anwendbare Technik verwendet das Fraunhofer IGD unter dem Namen Image Geometry in dem Framework X3DOM [Beh11]. In deren Ansatz werden die Geometriedaten in die Farbkanäle des Bildes gespeichert. Diese werden im Shader entsprechend ausgelesen und interpretiert. Damit können nahezu beliebige Attribute in den Bildern abgelegt werden.

In der vorliegenden Arbeit wird eine Abwandlung dieser Technik, jedoch ohne die Verwendung des Frameworks, implementiert. Zusätzlich wird untersucht, wie die Besonderheiten von JPEG und PNG ausgenutzt werden können. So kann der Progressive- und Interlace-Modus von JPEG und PNG dafür genutzt werden, um eine Vorschau des Modells während des Downloadvorgangs zu realisieren. Des Weiteren wird untersucht, wie gut die Formate JPEG und PNG geeignet sind, um Geometriedaten in diesen abzuspeichern.

3 Modellformate in WebGL

In OpenGL gibt es mehrere Wege, Geometriedaten an die Grafikkarte zu schicken. In älteren OpenGL Versionen, wie zum Beispiel der Version 1.0, wurden Zeichenbefehle über *glBegin()* und *glEnd()* definiert. Dieser Weg ist für moderne Grafikkarten jedoch nicht länger praktikabel, da so der Datenaustausch zwischen CPU und GPU nicht asynchron erfolgen kann. Aus diesem Grund wird empfohlen, Geometriedaten mittels Vertex Buffer Objects (VBOs) an die Grafikkarte zu übermitteln, um eine hohe Performance zu erreichen. Im WebGL Standard ist dies zugleich der einzige Weg, um Geometriedaten an die Grafikkarte zu übermitteln.

Die Flächen von 3D-Modellen bestehen meist aus drei oder vier Vertices. Darüber hinaus ist es möglich, dass mehr als vier Vertices eine vieleckige Fläche bilden. In der Regel wird dies jedoch nicht von der Grafikkarte unterstützt, da diese meist nur Flächen in Form von Dreiecken effektiv verarbeiten kann. In WebGL können lediglich Dreiecke an die Grafikkarte übermittelt werden. Aufgrund dieser Einschränkung müssen zuvor alle Flächen eines Modells entsprechend trianguliert werden. Die Indexliste hält fest, welche Positionen ein einzelnes Dreieck definieren. Sie besteht aus mehreren Indices, welche die Position des jeweiligen Attributes festhalten. Jeweils drei Indices bilden gemeinsam ein einzelnes Dreieck.

Doch nicht nur die Positionen der Vertices und die mit der Indexliste resultierenden Flächen definieren ein 3D-Modell. Ein Vertex kann darüber hinaus weitere Attribute besitzen. Diese Attribute werden anschließend von dem jeweiligen Shaderprogramm entsprechend ausgewertet und dargestellt. So ist für Lichtberechnungen wichtig, in welche Richtung die Normale eines Vertex zeigt. Diese wird zusammen mit den Normalen zweier weiterer Vertices, welche dann zusammen ein Dreieck bilden, genutzt, um die Orientierung des Dreieckes zu ermitteln. Mit Hilfe dieser Information kann letztendlich ermittelt werden, ob das Dreieck in Richtung der Lichtquelle gerichtet ist. Zudem können beliebige weitere Attribute, wie zum Beispiel die Tangente und Bitangente, an ein Vertex gebunden werden, um weitere Effekte zu erzielen.

Viele 3D-Modelle besitzen zudem eine oder mehrere Texturen, welche dem Modell ein realistisches Aussehen verleihen. Jedem Vertex wird dafür eine Texturkoordinate zugewiesen, welche definiert, an welcher Position sich dieser in der Textur befindet. In dieser Arbeit werden für die Modelle eine Position, Normale und Texturkoordinate für jeden Vertex benötigt. Die verwendeten Modelle wurden vorher entsprechend trianguliert. Im Folgenden wird auf den Aufbau der beiden Modellformate, in welchen diese vorliegen, eingegangen. Die Modelle müssen vor der weiteren Bearbeitung in ein gemeinsames Format überführt werden, damit eine Vergleichbarkeit der Resultate besteht. Letztendlich werden die beiden verwendeten Testmodelle detaillierter vorgestellt.

3.1 Modellformate

Für 3D-Modelle existiert kein standardisiertes Format, welches von allen Herstellern unterstützt wird. Dies macht es schwierig eine gemeinsame, vergleichbare Grundlage zu finden. In dieser Arbeit soll die Performance der evaluierten Kompressionstechniken anhand von Modellen des CryENGINE 3 Free SDK [Cry12a] sowie von einem eingescannten Modell des Goldenen Reiters der TU Dresden verglichen werden. Die Modelle liegen in unterschiedlichen Formaten vor. Die Modelle des CryENGINE 3 Free SDK nutzen ein proprietäres Format, welches Crytek Geometry Format (CGF) genannt wird. Dagegen liegt der 3D-Scan der TU Dresden im öffentlich dokumentierten Wavefront Object Format (OBJ) vor. Das Modellformat ist nicht der einzige Unterschied. So ist das CGF Format primär für den Einsatz in der

CryENGINE ausgelegt. Das Format enthält nicht nur Geometrieinformationen sondern zusätzlich Informationen über das verwendete Physikmodell, die verwendeten Materialien und weitere für die Engine relevante Informationen. Damit die Geometriedaten direkt ohne weitere Bearbeitung an die Grafikkarte weitergeleitet werden können, sind diese im Modell bereits entsprechend aufbereitet.

Zusätzlich sind die Geometrieinformationen des Modells für eine gute Ausnutzung des Vertex-Cache der Grafikkarte optimiert. Dies erlaubt eine schnellere Darstellung der 3D-Modelle und zudem werden dadurch ähnliche Vertices nebeneinander angeordnet. Dies kommt Kompressionsalgorithmen, wie zum Beispiel GZIP, entgegen, da damit die Entropie in der Regel ebenfalls reduziert wird. Bei Modellen, welche nicht entsprechend optimiert wurden, wird meist eine schlechtere Kompressionsrate erzielt.

Falls Modelle zwischen verschiedenen Anwendungen ausgetauscht werden sollen, muss darauf geachtet werden, dass ein möglichst offenes und weit verbreitetes Format genutzt wird. Das OBJ Format eignet sich durch seine frei einsehbare Spezifikation. Somit kann das Format mit relativ wenig Aufwand ausgewertet und in ein für die Grafikkarte darstellbares Format konvertiert werden.

3.1.1 Crytek Geometry Format

Bei dem CGF Format handelt es sich um ein proprietäres binäres Format von Crytek. Damit ist es nicht ohne Weiteres möglich die VBOs aus der Datei zu extrahieren. Das Free SDK bietet allerdings mehrere Möglichkeiten, um an die Geometriedaten der Datei zu gelangen. Mit dem Kommandozeilenwerkzeug *Resource Compiler* [Cry12b] können CGF Dateien in das offene Format Collada überführt werden. Es ist ferner möglich, den visuellen Editor der CryENGINE *Sandbox*, welcher Teil des Free SDK ist, zu nutzen. Über dessen Interface können Objekte direkt in das OBJ Format konvertiert werden. Die Formate OBJ und Collada sind textbasiert und können somit genutzt werden, um die nötigen Geometrieinformationen zu extrahieren. Diese werden jedoch nicht von allen Programmen gleich interpretiert. So unterstützt der OBJ Importer von OpenCTM keine getrennten Indices für Position, Normale und Texturkoordinate. Der visuelle Editor *Sandbox* und einige 3D-Modellierungsprogramme, wie *Blender*, nutzen jedoch getrennte Indices beim Export in das OBJ Format, um die Dateigröße der OBJ Datei klein zu halten, und sind deshalb nicht kompatibel. Um dennoch den Import zu ermöglichen, muss die OBJ Datei transformiert werden, damit ein gemeinsamer Index für alle Attribute bereitgestellt wird.

Die Cache-Optimierungen der Vertexreihenfolge kann beim Import in ein Modellierungsprogramm verloren gehen. Deshalb wurde in dieser Arbeit ein direkter Weg gewählt und die Geometrieinformationen direkt aus der CGF Datei ermittelt. Dies ist möglich, da das CryENGINE 3 Free SDK zusätzlich C-Code zum Auslesen der Daten zur Verfügung stellt. Das CGF Format legt die Geometriedaten in einzelnen Chunks ab. Es beinhaltet jeweils binäre Chunks für die Positionen, Normalen, Texturkoordinaten, Vertexfarbe, Tangente, Binormale und Indices. Der speziell für diese Arbeit entwickelte Exporter liest die einzelnen binären Chunks aus der Datei und schreibt diese in je einen separaten Array. Ähnlich wird die Indexliste aus der Datei ermittelt und gemeinsam mit den Attributen als Arrays in einer JSON Datei abgelegt.

Einige Modelle des CryENGINE 3 Free SDK verfügen über mehrere Materialien. In diesem Fall wird das Modell in mehrere Untermodelle unterteilt. Jedes Untermodell referenziert einen Teil der Indices und bindet eine oder mehrere Texturen. Diese Informationen werden aus der CGF Datei extrahiert und in einer separaten JSON Datei abgelegt. Aus den extrahierten Material- und Geometrieinformationen der CGF Datei kann nun eine OBJ Datei erstellt werden, welche aufgrund der aufgeführten Kompatibilitätsprobleme nur einen Index für alle Attribute verwendet.

Die meisten CGF Dateien beinhalten darüber hinaus mehrere Level of Details für das jeweilige Objekt. Diese Level of Details sind separate Modelle, welche weniger Dreiecke benötigen und das ursprüngliche 3D-Modell entsprechend abstrahieren. Diese Modelle werden vorher vom jeweiligen Modellierer manuell erstellt und verringern die Dreiecksgeometrieanzahl pro Level um jeweils die Hälfte. Das ist nötig, da die

CryENGINE so die Komplexität der Szene dynamisch verringern kann. Dieser Umstand kann ebenfalls ausgenutzt werden, um eine progressive Vorschau des 3D-Modells zu ermöglichen, indem die Levels in ihrer Komplexität aufsteigenden Reihenfolge übertragen werden.

3.1.2 Wavefront Object Format

Das 3D-Scan Modell der TU Dresden ist im OBJ Format abgelegt. Da das Format textbasiert vorliegt und auf einer offenen Spezifikation [Bou12] basiert, kann es relativ leicht ausgewertet und in andere Formate überführt werden. Das OBJ Format nutzt drei getrennte Indices, um Positionen, Normalen und Texturkoordinaten anzusprechen. Der OBJ Importer des OpenCTM Formates unterstützt jedoch nur einen Index für alle Attribute. Die OBJ Datei muss somit entsprechend abgeändert werden.

Beim Exportieren von OBJ Dateien mit dem 3D-Modellierungsprogramm *Blender* muss zudem beachtet werden, dass dieses alle Normalen beim Export neu berechnet und nicht die Normalen des importierten Modells verwendet. Unter Umständen ist ein Modell, welches vorher aus harten und weichen Kanten bestand, wesentlich größer, da es in der Folge aus nur noch harten Kanten besteht. Harte Kanten benötigen mehr Normalen pro Vertex und belegen so mehr Speicherplatz.

Für das 3D-Scan Modell wurde keine Cache-Optimierung vorgenommen, weshalb die Geometrieinformationen direkt aus der OBJ Datei gelesen werden können. In dieser Arbeit wurde ein Exporter für *Blender* auf der Grundlage von Krzysztof Solek [Sol12] programmiert. Dieser Exporter nutzt die interne Repräsentation von *Blender*, in welcher die Geometriedaten bereits in Form von VBOs und einer Indexliste vorliegen, um die Geometrieinformationen direkt in eine JSON Datei zu schreiben.

3.2 JSON

JSON (JavaScript Object Notation) [Cro06] ist ein Format, welches von Douglas Crockford eingeführt wurde, um Daten zwischen Client und Server auszutauschen. Dabei wurde speziell darauf Wert gelegt, dass dieses Format sehr einfach von JavaScript ausgewertet und gut von Menschen interpretiert werden kann. JSON basiert auf einem Subset von JavaScript (Standard ECMA-262 3rd Edition) und kann so direkt über die Funktion *eval()* geparst werden. JavaScript bietet zudem seit dem Standard ECMA-262 5rd Edition native Methoden an, um diese Serialisierung und Deserialisierung durchzuführen. JSON kann so von einem JavaScript Objekt zu einer textbasierten Repräsentation und umgekehrt überführt werden. Diese textbasierte Repräsentation kann anschließend für den Datenaustausch von Client und Server genutzt werden.

Es bietet sich somit an, die Modelldaten direkt in JSON abzulegen. Diese werden direkt als Arrays, aus denen später die VBOs erstellt werden, in der JSON Datei gespeichert. Ein mögliches Format ist dabei im Codelisting 3.1 dargestellt. Es speichert die VBOs für die Position, Normalen und Texturkoordinaten in jeweils einem separaten Array ab. Die Indexliste wird ebenfalls in einem Array abgelegt und daher implizit eine *Structure of Arrays*, welche in Abschnitt A beschrieben wird, verwendet. Dies hat in der JSON Datei den Vorteil, dass alle Attribute getrennt aufgeführt sind und leicht vom Entwickler auseinandergehalten werden können. Für die Verwendung im Renderer können diese Arrays in einem Vorverarbeitungsschritt zu einem *Array of Structures*, welcher ebenfalls in Abschnitt A beschrieben ist, zusammengeführt werden, um ein schnelleres Rendern zu ermöglichen.

Diese JSON Datei kann nun vom Client angefordert und in ein JavaScript Objekt deserialisiert werden. Über dieses Objekt werden anschließend die Arrays angesprochen und aus diesen die VBOs für die Verwendung in WebGL erstellt. Diese Vorgehensweise hat den Vorteil, dass die Modelldaten in der JSON Datei mit einem einfachen Texteditor manipuliert werden können. Sie wird in einigen WebGL Frameworks genutzt, da sie eine für den Entwickler transparente Methode darstellt, um Modelldaten zu

laden und zu manipulieren. Der Entwickler muss dafür lediglich mit WebGL und dem Darstellen von Modellen durch VBOs vertraut sein.

```
{
    "vertices": [1.000000, -1.000000, -1.000000, ...],
    "normals": [0.577349, 0.577349, -0.577349, ...],
    "uvs": [0.333333, 0.998981, ...],
    "faces": [4, 0, 3, 0, 1, 2, 0, 2, 3, ...]
}
```

Listing 3.1: Das JSON Modell Format, welches Vertex-Positionen, -Normalen und Texturkoordinaten unterstützt: Das Array *faces* enthält Triple, welche jeweils die Indices eines Dreieckes angeben.

3.3 Testmodelle

In den folgenden Abschnitten werden zwei verschiedene Testmodelle verwendet. Beide Modelle kommen aus unterschiedlichen Anwendungsgebieten und repräsentieren damit zwei verschiedene Anwendungsfälle. Das erste Modell aus Abbildung 3.1 zeigt eine Hinterhofszene. Das Modell stammt aus dem CryENGINE 3 Free SDK. Es ist also bereits für den Einsatz in dieser Engine optimiert und repräsentiert ein gängiges 3D-Modell, welches produktiv verwendet wird. Es besteht aus 4.615 Dreiecken und besitzt sechs Materialien. Die Materialien wurden für diese Arbeit gegenüber dem Originalmodell vereinfacht und bestehen lediglich aus einer diffusen Textur. Des Weiteren beinhaltet das Modell vier Level of Detail Stufen, welche in Abbildung 3.2 zu sehen sind. Jede nächsthöhere Detailstufe verdoppelt die Anzahl der Vertices. Die Dateigröße der komprimierten JSON Datei beträgt für die einzelnen Levels: 25 kB, 40 kB, 110 kB und 186 kB. Diese können für die progressive Vorschau des Modells verwendet werden.

Bei dem zweiten Modell handelt es sich um einen 3D-Scan des Goldenen Reiters, welcher in Abbildung 3.3 zu sehen ist. Dieses Modell wurde mit einem 3D-Scanner eingescannt und anschließend in dem OBJ Modell Format abgelegt. Das Modell besitzt keine harten Kanten, ist relativ gleichmäßig unterteilt und besitzt lediglich eine Textur. Das ursprüngliche Modell besteht aus 870.069 Dreiecken. In dieser Arbeit wurde es für einige Methoden auf 130.512 Dreiecke reduziert, damit lediglich 65.286 Vertices verwendet werden. Dies resultiert aus den Einschränkungen von WebGL, da über den Index lediglich 65.536 Vertices angesprochen werden können. Die Vereinfachung des Modells wurde mit dem Vereinfachungswerkzeug von *Blender* durchgeführt.

Abbildung 3.1: Das Modell zeigt eine Hinterhofszene aus dem CryENGINE 3 Free SDK, die in Ecken eines Levels eingebaut werden kann. Es besitzt sechs unterschiedliche Materialien und besteht aus insgesamt 4.615 Dreiecken. Es liegt im firmeneigenen CGF Format vor, welches in Kapitel 3.1.1 beschrieben wird, und ist in diesem Format ungefähr 280 kB groß. ©Modell, Crytek GmbH

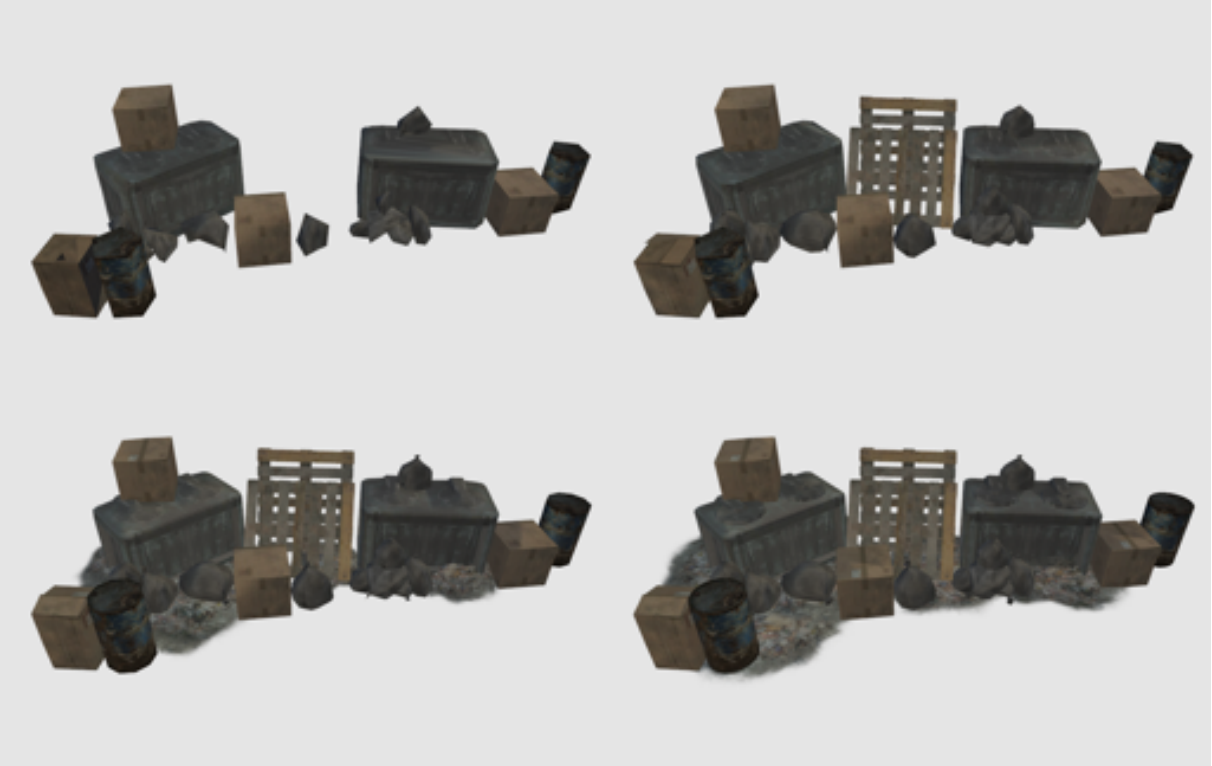

Abbildung 3.2: Das Modell der Hinterhofszene aus dem CryENGINE 3 Free SDK beinhaltet vier verschiedene Detailstufen des Modells, die von einem Artist speziell angefertigt wurden. Die Detailstufen sind von links oben bis nach rechts unten angeordnet. Die unterschiedlichen Detailstufen verdoppeln ungefähr die verwendete Anzahl der Vertices. Die Dateigröße der komprimierten JSON Datei beträgt für die einzelnen Levels: 25 kB, 40 kB, 110 kB und 186 kB. ©Modell, Crytek GmbH

Abbildung 3.3: Zu sehen ist ein vereinfachter 3D-Scan des Goldenen Reiters der TU Dresden. Das Modell ist eine vereinfachte Version des ursprünglichen Modells, welche mit *Blender* erstellt wurde. Es besitzt 130.512 Dreiecke und eine Textur als Material. ©Modell, TU Dresden

4 Kompression

Bei der Übertragung von Dateien im Internet spielt schon sehr lange die Komprimierung dieser eine wichtige Rolle. In diesem Kapitel werden eingehend die Basistechniken beschreiben, welche im Laufe der Arbeit verwendet werden. Anschließend wird die Kompression von HTTP beschrieben, die bei der Übertragung der Daten von Server zu Client verwendet werden kann.

Durch eine Sortierung der Daten kann häufig eine noch bessere Kompressionsrate erreicht werden. Einige allgemein auf Geometriedaten anwendbare Sortierverfahren, wie die Optimierung der Indexliste und das versetzte Abspeichern der Attribute, werden dafür in diesem Kapitel behandelt. Den Abschluss des Kapitels bildet die Quantisierung der Daten, bei welcher diese in geringerer Präzision abgespeichert werden und so weniger Speicherplatz benötigen.

4.1 Basistechniken

Nachfolgend werden die grundlegenden Techniken erläutert, welche für die Kompression der Geometriedaten verwendet werden. Die Dateigröße von Dateien kann lediglich dadurch reduziert werden, indem Duplizierungen und Redundanzen aus einer Zeichenfolge entfernt werden. Es ist nicht möglich zufällige Zeichenfolgen zu komprimieren. Es ist somit von Vorteil die Daten entsprechend aufzuarbeiten und zu sortieren, so dass diese gut von den Kompressionsalgorithmen verarbeitet werden können.

4.1.1 Delta-Kodierung

Delta-Kodierung verringert nicht den benötigten Speicherplatz der Daten, sondern transformiert diese vielmehr, damit diese anschließend besser komprimiert werden können. Durch die Kodierung soll der Bereich der auftretenden Zeichen verringert und so die Entropie der Datei insgesamt verkleinert werden. Besonders bei Daten, welche wenig variieren, erhöht dieser Algorithmus die Effektivität von Kompressionsalgorithmen. Bei zufälligen Daten führt dieser Algorithmus zu keinem messbaren Erfolg. Die Daten sind im Idealfall vorher entsprechend sortiert und ähnliche Werte stehen nebeneinander.

Der Algorithmus ist leicht zu implementieren und in linearer Zeit ($\mathcal{O}$(n)) durchführbar. Als Beispiel soll die folgende Zahlenfolge mittels Delta-Kodierung kodiert werden: $4, 8, 12, 13, 12$. Der Algorithmus notiert zuerst die erste auftretende Zahl. Anschließend ermittelt dieser die Distanz zur nächsten Zahl und vermerkt auch diese. Für die nachfolgenden Zahlen wird wieder jeweils die Distanz zur vorhergehenden Zahl notiert, bis das Ende der Zahlenfolge erreicht ist. Das Ergebnis ist demzufolge: $4, 4, 4, 1, -1$. Diese Zahlenfolge kann nun mit weiteren Kompressionsalgorithmen besser komprimiert werden, da mehr Doppelungen auftreten und somit die Entropie insgesamt verringert wurde.

4.1.2 Huffmann-Kodierung

Durch die Huffman-Kodierung [Huf52] wird jedem Zeichen eine eindeutige Bitfolge zugeordnet. Die Grundidee dabei ist, jedem Zeichen die optimale Bitlänge zuzuteilen, damit $sum(Zeichenanzahl[n] * bit[n], 0, 255)$ minimal ist. Damit die Zeichenkette eindeutig dekodierbar ist, müssen die Bitfolgen präfixfrei sein. Es darf also keine Bitfolge am Anfang einer anderen längeren Bitfolge vorkommen.

Um die Bitfolgen für jedes Zeichen zu ermitteln, müssen vorerst die Wahrscheinlichkeiten dieser bekannt sein. Für jedes Zeichen wird ein Knoten mit der entsprechenden Wahrscheinlichkeit erstellt. Aus diesen Knoten wird anschließend ein Baum konstruiert. Die beiden Knoten mit der kleinsten Wahrscheinlichkeit werden zu einem neuen Wurzelknoten zusammengefasst, welcher in einer höheren Ebene platziert wird. Die Wahrscheinlichkeiten der beiden Knoten werden addiert und dem Wurzelnoten zugewiesen. Anschließend werden wieder die beiden kleinsten Wahrscheinlichkeiten gesucht und miteinander zu einem neuen Wurzelknoten verbunden. Dies wird schrittweise durchgeführt, bis alle Knoten über einen gemeinsamen Wurzelknoten erreichbar sind. Anschließend kann aus dem Baum die Bitfolge für jedes Zeichen abgelesen werden, indem dieser vom Wurzelknoten aus traversiert wird. Dabei wird bei jeder Abzweigung die Bitfolge für das Zeichen erhöht. Für eine Linksabzweigung wird eine 0 und für eine Rechtsabzweigung eine 1 an die Bitfolge konkateniert. Die Zeichen nahe der Wurzel bekommen daher eine kleinere Bitfolge zugewiesen, als Zeichen am Ende des Baumes.

4.1.3 LZ77

Der Algorithmus LZ77 wurde 1977 von Abraham Lempel und Jacob Ziv entworfen [ZL77]. Im Gegensatz zur Huffman-Kodierung, bei welcher einzelne Zeichen nach ihrer Häufigkeit kodiert werden, nutzt dieser Algorithmus die Wiederholung von Zeichenketten, um eine Komprimierung der Daten zu erreichen. Dabei wird ein Wörterbuch fester Größe verwendet, welches schrittweise ergänzt wird. In diesem werden die Daten abgelegt, welche zuvor eingelesen wurden. Die Daten des Wörterbuches bilden so eine Art Fenster, welches schrittweise über den zu komprimierenden Datensatz geschoben wird. Daher spricht man hier von *sliding window compression*. Das Fenster besteht aus einem Vorschaupuffer und einem Suchpuffer. Der Vorschaupuffer enthält die Daten, welche komprimiert werden sollen. Der Suchpuffer enthält dagegen Zeichen, welche bereits eingelesen wurden.

Beim Verschieben des Fensters wird jeweils ein Tripel von Daten notiert. Das Tripel enthält eine Position, Länge und ein Zeichen. Dieses gibt an, ab welcher Position wie viele Zeichen des Wörterbuches kopiert werden sollen. Das Zeichen des Tripels wird an den kodierten String anschließend angehängt. Der Algorithmus ist im Listing 4.1 an dem Wort BANANAS beispielhaft durchgeführt. In der Implementation des LZ77-Algorithmus von GZIP werden Zeichenketten, welche kleiner als drei Zeichen sind, nicht beachtet, da hierbei der Overhead durch die Positions- und Längenangabe größer als die Einsparung ist. Der

```
 01234567
|________|____|
|        |BANA| : (0,0,B)
|       B|ANAN| : (0,0,A)
|      BA|NANA| : (0,0,N)
|     BAN|ANAS| : (6,2,A)
|  BANANA|S   | : (0,0,S)
| BANANAS|    | : (-,-,-)
```

Listing 4.1: Die Kodierung des Wortes BANANAS mit LZ77: Hier wird ein Wörterbuch mit acht Zeichen verwendet. Die Größe des Vorschaufensters beträgt vier Zeichen. Eine Einsparung wird im vierten Schritt bei der Zeichenfolge *AN* erreicht [ZL77]. In diesem wird die Zeichenfolge aus dem Wörterbuch der Länge 2 an der Position 6 wiederholt und anschließend der Buchstabe A angehangen.

Dekodierungsalgorithmus ist im Listing 4.2 aufgezeigt. Dabei werden die vorher abgespeicherten Triple ausgewertet, indem der Algorithmus umgekehrt angewendet wird.

Der LZ77-Algorithmus komprimiert Daten effektiv, sobald diese gleiche Abfolgen besitzen. Für völlig zufällige Zeichenfolgen erzeugt der Algorithmus keine Einsparung. Die Größe des Wörterbuches ist dabei ein wichtiger Faktor. Denn ein großes Wörterbuch bedeutet in der Regel ein gutes Kompressionsverhältnis, erhöht jedoch die Laufzeit und den Speicherverbrauch. Für GZIP wird eine Größe bis zu 32 kB verwendet.

```
          01234567
(0,0,B) :|_________|
(0,0,A) :|________B|
(0,0,N) :|_______BA|
(6,2,A) :|_____BAN|
(0,0,S) :|__BANANA|
(-,-,-) :|_BANANAS|
```

Listing 4.2: Die Dekodierung des Wortes BANANAS mit LZ77: Hier wird ein Fenster der Größe acht verwendet [ZL77]. Im vierten Schritt wird der Tupel (6,2,A) entpackt.

4.1.4 Deflate

Deflate wurde von Phil Katz für die Anwendung im ZIP Format entworfen. Der Algorithmus kombiniert die bereits erwähnten Techniken, LZ77 und Huffman-Kodierung. Dabei wird zuerst LZ77 verwendet, um doppelte Daten aus dem Datenstrom zu entfernen. Anschließend werden die Daten mittels Huffman kodiert [Fel97]. Durch diesen Schritt werden häufig genutzte Zeichen durch kürzere Bitketten ersetzt. Die Daten werden blockweise komprimiert. Jeder Block kann entweder einen fest vordefinierten oder einen statistisch aus den aktuellen Daten erstellten Huffman-Baum verwenden. Die Verwendung des statistisch ermittelten Huffman-Baumes resultiert in einer besseren Kompressionsrate auf Kosten der Kodierungszeit. Durch die Kombination beider Techniken werden beide Vorteile vereint und somit eine hohe Kompressionsrate erreicht. Der Deflate-Algorithmus findet Anwendung in zahlreichen Kompressionsbibliotheken, so zum Beispiel auch bei der HTTP Komprimierung mit GZIP.

4.2 HTTP-Kompression

Das JSON Format erscheint auf den ersten Blick wenig geeignet, um 3D-Modelle effektiv zu übertragen. Sämtliche Daten sind im Textformat abgelegt und vergrößern so die Datei nicht unwesentlich. Jedoch muss hierbei beachtet werden, dass bei der Übertragung der Dateien vom Server zum Client seit dem Standard HTTP 1.1 eine zusätzliche Kompression angefordert werden kann [Net99].

Dabei können unterschiedliche Kompressionsalgorithmen verwendet werden, je nachdem, welche vom Server und Client beidseitig unterstützt werden. Die gängigsten Optionen sind dabei GZIP oder Deflate. Die Bezeichnung der beiden Methoden ist hierbei verwirrend [Hof12]. Der Deflate-Algorithmus komprimiert in der Regel lediglich die Daten, ohne weitere Information in einen Header zu schreiben. Nach dem Standard RFC-2616 wird der Server jedoch angefordert, das Paket mittels ZLIB zu komprimieren, sobald der Client Deflate als Kompression anfordert. ZLIB ist ein weiteres Kompressionsformat, welches die Daten mittels Deflate komprimiert und zusätzlich einen Header und eine Prüfsumme hinzufügt. Ältere Implementationen des Internet Explorers und Internet Information Service (IIS) verarbeiten die Daten jedoch lediglich mit Deflate und senden diese ohne Header und Prüfsumme. Dies sorgt für einige Inkompatibilitäten, weshalb empfohlen wird, GZIP als Kompressionsmethode zu verwenden, da diese von den meisten Servern und Browsern gleich interpretiert wird. GZIP komprimiert die Daten ebenfalls mittels Deflate und fügt, wie bei ZLIB, einen Header und eine Prüfsumme hinzu. Die beiden Methoden unterscheiden sich lediglich im Format des verwendeten Header.

Damit Daten komprimiert gesendet werden können, muss der Client dem Server vorerst mitteilen, welche Kompressionsmethoden unterstützt werden. Dies geschieht im *Accept-Encoding* GET Requests. In diesem kann der Client GZIP als Kompressionsmethode anfordern, wenn dies unterstützt wird. Der Server nutzt diese Information, um die angeforderten Daten, falls unterstützt, mittels GZIP zu komprimieren und zu versenden. Die Kompression und Dekompression kann sehr schnell durchgeführt werden, da auf beiden Seiten nativer Code verwendet wird.

Mittels GZIP kann die Größe der JSON Datei um den Faktor 1 zu 4 verringert werden. Dies hängt natürlich vom konkreten Aufbau der JSON Datei ab. Damit kann bereits durch diesen Schritt ohne jegliche Codeänderungen im Client oder Servercode die Größe der Modelldaten entscheidend verkleinert werden. Im Folgenden wird GZIP als Kompressionsmethode für unkomprimierte Daten bei der Übertragung verwendet.

4.3 Sortierung der Daten

Kompressionsalgorithmen, wie Deflate, können Daten besonders gut komprimieren, wenn viele gleiche Werte nebeneinander liegen. Oft kann dies durch das Sortieren der Daten erreicht werden. Beispielsweise soll die folgende Zeichenfolge mittels einer einfachen Lauflängenkodierung komprimiert werden: $AACCAACC$. Als Ergebnis entsteht hierbei die folgende Zeichenfolge: $2A2C2A2C$. Der kodierte String ist genauso groß wie die ursprüngliche Zeichenfolge. Ist die Reihenfolge der Daten jedoch nicht relevant, so können diese im Vorfeld beispielhaft in die folgende Reihenfolge sortiert werden: $AAAACCCC$. Diese Zeichenfolge kann mittels Lauflängenkodierung folgendermaßen komprimiert werden: $4A4C$. Es macht somit unter bestimmten Umständen einen beträchtlichen Unterschied bei der Komprimierung der Daten aus, in welcher Reihenfolge die Daten letztendlich vorliegen.

Die Geometriedaten können nicht beliebig sortiert werden, da die Reihenfolge teilweise vorgegeben ist. Die Sortierung der einzelnen Index-Tupel muss also erhalten bleiben. Werden die Index-Tupel, welche ein Dreieck definieren, in einer anderen Reihenfolge sortiert, so wird die Orientierung des definierten Dreieckes verändert. Für die Darstellung der Daten ist es nicht relevant, in welcher Reihenfolge die einzelnen Dreiecke gerendert werden, solange die Orientierung dieser beibehalten wird.

4.3.1 Cache-Optimierung

Moderne Grafikkarten besitzen einen Cache, welcher verarbeitete Vertices zwischenspeichert. Dieser Cache kann in der Regel 12 bis 24 Vertices beinhalten. Sobald ein neuer Vertex gerendert werden soll, wird zuerst im Cache überprüft, ob dieser bereits abgearbeitet wurde. Ist dies der Fall, so wird das Ergebnis des Caches verwendet. Vertices werden im Cache nach dem FIFO Prinzip verwaltet. Um eine möglichst hohe Performance zu erreichen, müssen die Geometriedaten entsprechend geordnet vorliegen, damit dieser Cache häufig ausgenutzt werden kann.

Die Effektivität des Vertex-Caches, wird in der Regel als Average Cache Miss Ratio (ACMR) angegeben [Any12]. Dieses ergibt sich aus der Anzahl, wie oft ein Vertex verarbeitet werden muss, geteilt durch die Anzahl der Dreiecke eines Modells. Das ACMR kann für ein typisches Modell minimal 0.5 betragen [Cas09], da ein Vertex durchschnittlich sechs mal wiederverwendet werden kann und ein Dreieck drei Vertices besitzt.

Zur Optimierung des ACMR existieren einige Algorithmen. In dieser Arbeit wurde der Algorithmus von Forsyth verwendet [For06]. Dieser Algorithmus optimiert die Indexliste in linearer Zeit ($\mathcal{O}$(n)) abhängig von der Größe des Modells. Dabei wird ein Least Recently Used (LRU) Cache emuliert. Jedem Vertex werden Punkte nach der Wahrscheinlichkeit der Wiederverwendung gegeben. Dies hängt davon ab, welche Position der Vertex im LRU Cache besitzt. Zusätzlich spielt die Valenz eines Vertex eine Rolle. Vertices mit einer geringen Valenz werden stärker gewichtet. Das verhindert die Entstehung von Dreiecksinseln, welche zum Schluss aufgesammelt werden müssten und so das ACMR wesentlich erhöhen. Jedes Dreieck erhält die summierten Punkte aller drei Vertices. Der Algorithmus fügt das Dreieck hinzu, welches die meisten Punkte besitzt. Anschließend werden die Punkte für jeden Vertex neu berechnet und die Berechnung erneut gestartet bis alle Dreiecke verarbeitet wurden.

Nachdem die Indexliste optimiert wurde, kann diese neu indiziert werden, damit die Indices aufsteigend verteilt sind. Anschließend können die Attribute nach dem neuen Index sortiert werden. Damit stehen

in der Regel ähnliche Attribute nebeneinander, was wiederum zu einer besseren Kompressionsrate führt. Die Abbildung 4.1 zeigt eine Visualisierung der nun sortierten Indexdaten. In dem Beispiel wurde die Cache-Optimierung auf ein reduziertes Modell des Goldenen Reiters angewandt. Das Modell wurde dabei mittels des *Decimate Modifier* von *Blender* auf 20.880 Dreiecke reduziert, damit die Indexe in einer 256×256 Textur Platz finden. Das ACMR betrug vor der Optimierung $2,99$. Nach der Optimierung beträgt dieser $1,12$.

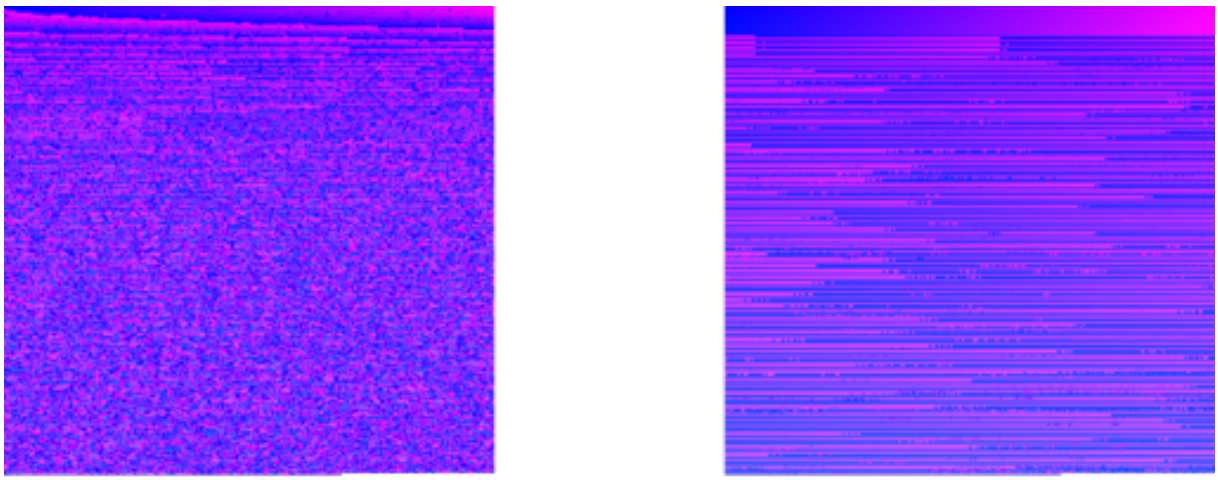

Abbildung 4.1: Cache-Optimierung der Indices des vereinfachten Modells des Goldenen Reiters. Zur Visualisierung wurde in jedem Pixel, im Rot- und Grünkanal der jeweilige Index kodiert. Dies ist näher in Abschnitt 7 beschrieben. Das Bild links zeigt die Anordnung der Indices vor der Optimierung. Rechts ist die Anordnung nach der Optimierung durch Forsyth. Die Abbildung zeigt, dass nach der Optimierung die Daten besser angeordnet sind.

4.3.2 Attribute versetzt abspeichern (Element Interleaving)

Die Attribute der Geometriedaten werden in der Regel als eine Liste von aufeinanderfolgenden 3D-Vektoren abgespeichert. Um diese Liste für eine nachfolgende Komprimierung zu optimieren, sollten die Attribute nach ihrer Position im Raum sortiert werden. Somit unterscheiden sich im besten Fall die einzelnen Elemente der 3D-Vektoren nur in einigen Stellen voneinander. Im Idealfall wird dieser Kontext von dem verwendeten Kompressionsalgorithmus erkannt und ausgenutzt. Da GZIP jedoch keine Mustererkennung besitzt, ist es für GZIP nicht ohne Weiteres möglich, dieses Muster aus aufeinanderfolgenden ähnlichen 3D-Vektoren zu erkennen. Wie in Abschnitt 4.1.3 beschrieben, verwendet GZIP ein gleitendes Fenster, welches lediglich nacheinander die Werte betrachtet und nicht erkennt, dass 3D-Vektoren verwendet werden. Da sich die einzelnen Komponenten des 3D-Vektors meist stark unterscheiden, kann so keine gute Kompressionsrate erzielt werden. Eine Komprimierung mit Mustererkennung, wie zum Beispiel Prediction by Partial Matching (PPM), ist in der Lage, diese Muster wesentlich besser zu komprimieren. Dies würde jedoch voraussetzen, dass diese Methode über HTML zur Verfügung steht, was zum Stand der Arbeit jedoch nicht der Fall ist.

Um dennoch eine gute Komprimierungsrate mit GZIP zu erreichen, können die Komponenten der einzelnen Vektoren versetzt abgespeichert werden. Dies bedeutet, dass die einzelnen Komponenten der Tupel nun hintereinander angeordnet werden. Das folgende Beispiel illustriert die Technik für einen Beispiel Array:

$x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2, ..., x_n, y_n, z_n \Rightarrow$
$x_0, x_1, x_2, ..., x_n, y_0, y_1, y_2, ..., y_n, z_0, z_1, z_2, ..., z_n$

Falls es sich bei den Komponenten um Integer-Variablen handelt, kann zusätzlich eine Delta-Kodierung angewandt werden, um den Wertebereich insgesamt weiter zu verkleinern. Damit stehen im Idealfall viele gleiche Werte nebeneinander, was wiederum zu einer guten Kompressionsrate mit GZIP führt.

4.3.3 Bytes versetzt abspeichern (Byte Interleaving)

Im Abschnitt 4.3.2 wurden die Elemente des Vektors versetzt abgespeichert, damit ähnliche Daten näher beieinanderstehen. Dies kann ebenfalls für die einzelnen Elemente selbst durchgeführt werden. Dabei wird das Element in seine einzelnen Byte-Komponenten zerlegt und versetzt abgespeichert. Die Attribute der Geometriedaten liegen in den Ursprungsdaten meist als Float-Variablen vor. Damit der gewünschte Effekt eintritt, müssen diese Float-Variablen in entsprechende Integer-Variablen überführt werden. Kleine Float-Werte besitzen in der Regel keine kleinere Entropie als große Float-Werte. Dies liegt an der Zerlegung der Float-Variable in Exponent und Mantisse. Eine 32 bit Float-Variable enthält im vorderen Byte nach IEEE-754 das Vorzeichen und 7 bit des Exponenten. Das nachfolgende Byte enthält das restliche Bit des Exponenten und 7 bit der Mantisse. Die restlichen beiden Byte enthalten je 8 bit der Mantisse. Daher ist es besser, Float-Variablen vorher in Integer-Variablen zu transformieren, da hier kleine Integer-Werte in der Byte-Repräsentation ebenfalls viele führende Nullen besitzen und somit die Entropie insgesamt verringern. Das Beispiel in Listing 4.3 illustriert dies an einem konkreten Beispiel. Durch die Delta-Kodierung soll der Wertebereich und dementsprechend die Entropie verkleinert werden. Dies ist für die Byte-Repräsentation der Float-Variablen jedoch nicht der Fall. Die Integer-Variablen besitzen nach der Delta-Kodierung in ihrer Byte-Repräsentation dagegen viele führende Null-Bytes.

```
1  Float
2  1.2345 -> 00111111 10011110 00000100 00011001
3  1.2346 -> 00111111 10011110 00000111 01011111
4  1.2347 -> 00111111 10011110 00001010 10100110
5
6  Float delta-kodiert
7  1.2345 -> 00111111 10011110 00000100 00011001
8  0.0001 -> 00111000 11010001 10110111 00010111
9  0.0001 -> 00111000 11010001 10110111 00010111
10
11
12 Integer
13 12345  -> 00000000 00000000 00110000 00111001
14 12346  -> 00000000 00000000 00110000 00111010
15 12347  -> 00000000 00000000 00110000 00111011
16
17 Integer delta-kodiert
18 12345  -> 00000000 00000000 00110000 00111001
19 1      -> 00000000 00000000 00000000 00000001
20 1      -> 00000000 00000000 00000000 00000001
```

Listing 4.3: Die Konvertierung einer Float-Variable zu einer Integer-Variable bringt Vorteile, sobald die Daten delta-kodiert werden. Kleine Float-Werte besitzen in der Regel gegenüber großen Float-Werten keine kleinere Entropie. Daher ist es besser, diese vorher in Integer-Variablen zu transformieren, da hier kleinere Werte in der Byte-Repräsentation viele führende Nullen besitzen und so die Entropie insgesamt verringern.

Um nun möglichst viele dieser Null-Bytes hintereinander anzuordnen, werden die Bytes versetzt abgespeichert. Das folgende Beispiel illustriert die Technik für ein Beispiel mit einem Array aus 32 bit Variablen:

$a_0, b_0, c_0, d_0, a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, ..., a_n, b_n, c_n, d_n \Rightarrow$
$a_0, a_1, a_2, ...a_n, b_0, b_1, b_2, ..., b_n, c_0, c_1, c_2, ..., c_n, d_1, d_2, ..., d_n$

Besonders gut geeignet ist diese Technik, falls viele kleine 32 bit Integer-Variablen im Bereich von $[0, 255]$ abgespeichert werden sollen, wie es nach einer Delta-Kodierung oft der Fall ist. Die vorderen drei Bytes

der 32 bit Zahl bestehen in diesem Fall lediglich aus Nullen, welche durch das versetzte Speichern nun eine lange Kette bilden. GZIP verarbeitet die Daten in 32 bit Blöcken und kann deswegen bei vielen Null-Bytes sehr effektive Huffman-Bäume erstellen. Wie beim Element Interleaving wird diese Sortierung durch einen besseren Kompressionsalgorithmus, welcher Mustererkennung unterstützt, unnötig.

4.4 Quantisierung

Die Attribute der Ausgangsdaten werden in der Regel als 32 bit Float-Variablen für jede Komponente abgelegt. Diese hohe Präzision wird oft nicht benötigt und kann darum durch entsprechende Quantisierung verringert werden. Je niedriger die Präzision ist, desto kleiner ist gleichzeitig der benötigte Speicherplatz. Wie weit diese konkret verringert werden kann, hängt stark vom jeweiligen Modell beziehungsweise dem Anwendungsfall ab. Für die gängigsten Anwendungsfälle wird eine Quantisierung auf 16 bit weiterhin gute Ergebnisse liefern. In der Hinterhofszene aus Abbildung 3.1 kann die Präzision auf 8 bit pro Komponente für Position, Normale und Texturkoordinate verringert werden. Es entstehen zwar kleinere Artefakte gegenüber der Szene aus den Ursprungsdaten, diese fallen visuell allerdings nicht besonders ins Gewicht. Jedoch sorgt die starke Quantisierung dafür, dass zwei planare Flächen des Modells in der selben Ebene liegen. Dadurch entstehen Artefakte, welche Z-Fighting genannt werden. In den Ursprungsdaten liegen diese Ebenen zwar sehr nah beieinander, allerdings nicht übereinander. Dieses Artefakt verschwindet erst bei einer Präzision von 12 bit. Ab dieser Präzisionsstufe sind darüber hinaus visuell nur noch minimale Unterschiede gegenüber der 32 bit Präzision zu erkennen.

Dies gilt ebenfalls für das Modell des Goldenen Reiters aus Abbildung 3.3. Hier wäre je nach Anwendungsfall bereits eine Präzision von 10 bit ausreichend. Ab 12 bit sind bei diesem Modell ebenfalls visuell nur noch minimale Unterschiede gegenüber den Ausgangsdaten zu erkennen. An beiden Modellen ist zu erkennen, dass je nach Anwendungsfall eine wesentlich geringere Präzision als 32 bit nötig ist. Bei einer Quantisierung auf 16 bit kann somit bereits die Hälfte des benötigten Speicherplatzes eingespart werden. Gleichzeitig steigt die Kompressionsrate, da nun häufiger gleiche Zeichenketten vorkommen. Die Normalen und Texturkoordinaten können darüber hinaus mit einer geringeren Präzision als die Positionen abgespeichert werden. Eine Präzision von 10 bit ist dabei oft schon ausreichend.

Die Höhe der Quantisierung muss je nach Anwendungsfall und Modell entschieden werden. Idealerweise wird manuell, mit Hilfe einer Vorschau, die niedrigste Quantisierungsstufe bestimmt, bis zu welcher die visuellen Artefakte vertretbar sind. Denkbar wäre ebenfalls eine Metrik, ähnlich wie im ROAM-Algorithmus [Boe00], welche während des Streaming, abhängig vom aktuellen Bildausschnitt, den jeweiligen Fehler feststellt und die geeignete Quantisierungsstufe anfordert.

Bei der Quantisierung der Daten handelt es sich zwar um eine verlustbehaftete Komprimierung, jedoch fällt diese bei gut gewählten Präzisionstufen visuell kaum auf. Die Abbildungen 4.2 und 4.3 zeigen die einzelnen Quantisierungsstufen für eine Präzision von 4 bit, 6 bit, 8 bit, 10 bit, 12 bit und 16 bit. Bereits ab 8 bit ist nur bei näherem Betrachten ein visueller Unterschied zu erkennen. Weit entfernte Objekte, welche kleine Level of Details verwenden, können so durchaus mit einer geringeren Präzision abgespeichert werden.

Die Daten werden quantisiert, indem die Bounding Box für das Modell ermittelt wird. Mithilfe dieser können anschließend die Float-Attribute in Integer-Variablen umgewandelt werden, indem diese von dem Float-Bereich der Bounding Box in den jeweiligen quantisierten Integer-Bereich, zum Beispiel $[0, 255]$ für 8 bit, transformiert werden. Die Größe des Integer-Bereiches gibt dabei die Quantisierung vor. Das Codebeispiel 4.4 zeigt, wie die Float-Werte der Attribute in den 8 bit Integer-Bereich transformiert werden. Nach der Quantisierung können die Integer-Daten byteweise im Binärformat abgelegt werden. Durch die anschließende Komprimierung mit GZIP ist es zudem möglich, die reduzierten Integer-Werte weiterhin als 32 bit Integer-Variablen abzulegen. Die Kompression sorgt dafür, dass die führenden Nullen innerhalb der größeren 32 bit Integer-Variablen weitestgehend komprimiert werden. In der Tabelle 4.1 ist ein

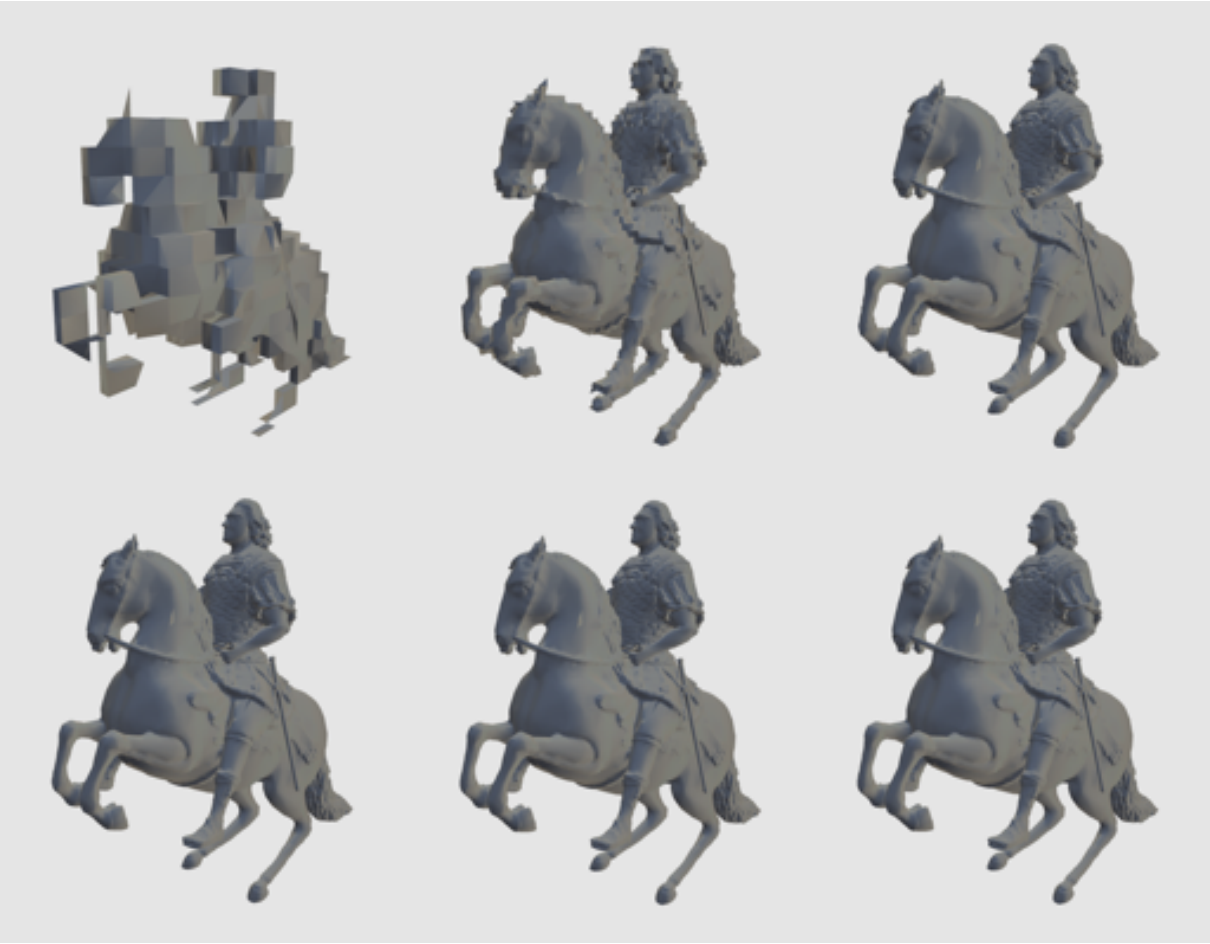

Abbildung 4.2: Quantisierung des 3D-Scan des Goldenen Reiters. Von links oben nach rechts unten nimmt die Präzision von 4 bit, 6 bit, 8 bit, 10 bit, 12 bit und 16 bit stückweise zu. ©Modell, TU Dresden

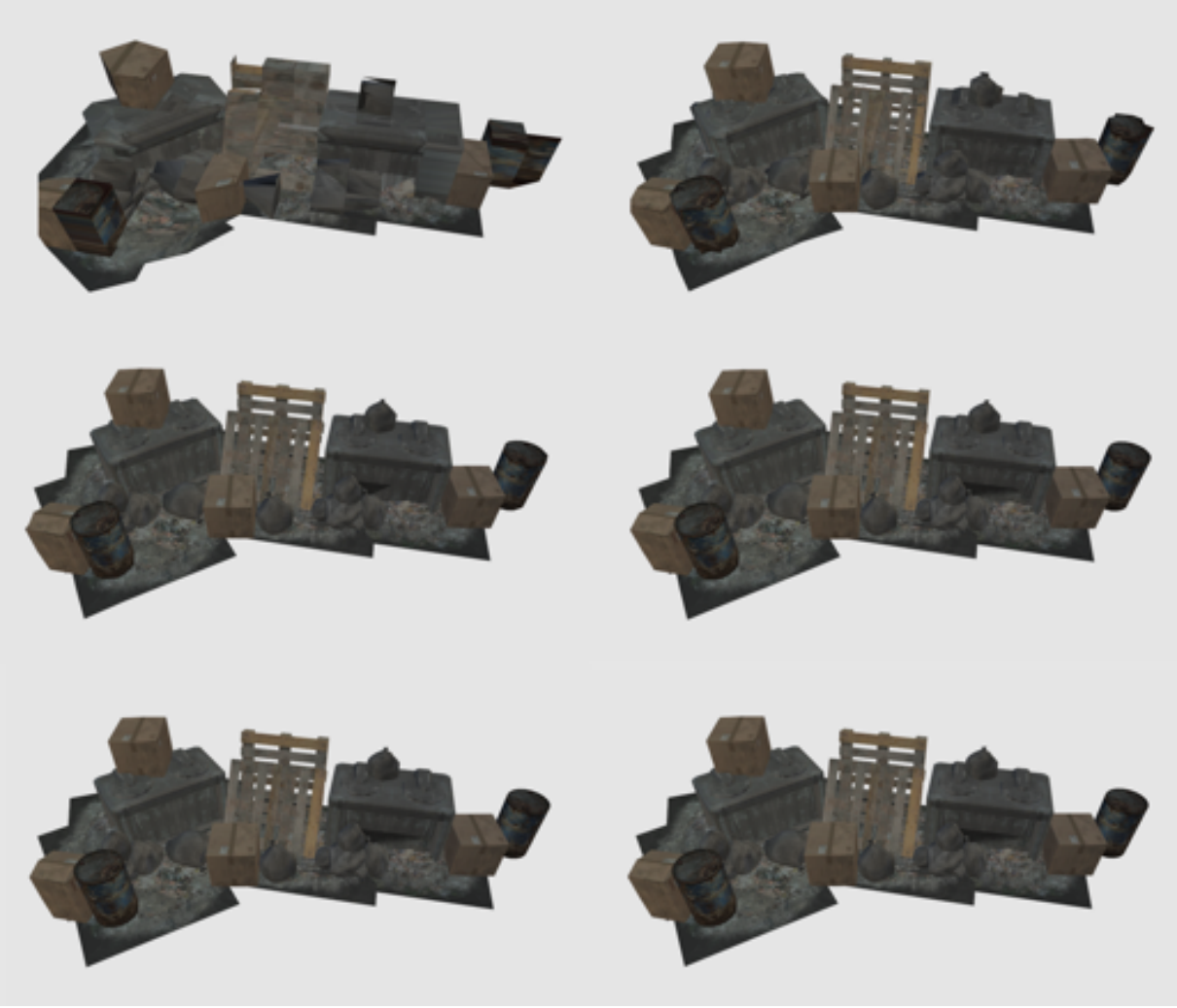

Abbildung 4.3: Quantisierung der Hinterhofszene aus dem CryENGINE 3 Free SDK. Von links oben nach rechts unten nimmt die Präzision von 4 bit, 6 bit, 8 bit, 10 bit, 12 bit und 16 bit stückweise zu. ©Modell, Crytek GmbH

Vergleich zwischen der Speicherung als Short-Variable und Integer-Variablen mit jeweils anschließender Kompression durch GZIP aufgeführt. Hierbei ist zu erkennen, dass GZIP nur zum Teil die redundanten Informationen, welche durch die führenden Nullen entstehen, komprimiert. Ein besserer Kompressionsalgorithmus mit Mustererkennung führt hierbei zu besseren Resultaten.

```
public static Short map(float value, float aabbMin, float aabbMax) {
    return  (short) (255 * ((value - aabbMin) / (aabbMax- aabbMin)));
}
```

Listing 4.4: Quantisierung der Float-Attribute in den Integer-Bereich: Dabei werden alle Werte nach 8 bit (Bereich $[0, 255]$) quantisiert. Im Shader werden, mit Hilfe der Bounding Box, die Attribute anschließend wieder zurück transformiert.

	16 bit Short GZIP	32 bit Integer GZIP
2 bit	13.823 byte	17.106 byte
4 bit	20.092 byte	23.629 byte
8 bit	35.240 byte	40.842 byte
16 bit	49.298 byte	61.476 byte

Tabelle 4.1: Vergleich zwischen der Speicherung als Short-Variable und Integer-Variable mit anschließender GZIP Komprimierung. Die Komprimierung sorgt im optimalen Fall dafür, dass die redundanten führenden Nullen im 32 bit Integer-Format komprimiert werden. Jedoch ist in diesem Vergleich für das Modell der Hinterhofszene zu erkennen, dass dies nur teilweise zum Erfolg führt. Im 16 bit Short-Format beträgt die unkomprimierte Größe 102.202 byte. Das 32 bit Integer-Format belegt unkomprimiert den doppelten Speicherplatz (204.404 byte).

Die Quantisierung kann des Weiteren dazu genutzt werden, um einfache Level of Detail Stufen zu erzeugen. Besonders bei komplexen und geschlossenen Modellen, wie dem Goldenen Reiter, kann dies gut für den progressiven Transfer verwendet werden. Denkbar wäre zu Beginn eines Transfers lediglich die Indices, welche sehr gut komprimiert werden können, zu transferieren und anschließend die Attribute mit lediglich 4 bit zu übertragen. Die Präzision der Attribute wird danach erhöht, indem diese schrittweise verbessert wird. Damit wird eine progressive Vorschau wie in Abbildung 4.2 erzielt. Bereits übertragene Informationen werden so für die höheren Level of Details wiederverwendet. Ein ähnlicher Ansatz wurde bereits experimentell vom Fraunhofer IGD erprobt [Fra12].

5 Progressive Meshes

Im Idealfall soll der Nutzer während des Streaming eine Vorschau des 3D-Modells erhalten. Für diese können die einzelnen Level of Details des Modells hintereinander übertragen werden. Da die Level of Detail Modelle jedoch gegenseitig nicht aufeinander aufbauen, können bereits übertragene Levels nicht wiederverwendet werden. Besser ist es, aus dem höchsten Level of Detail während der Übertragung eine progressive Vorschau abzuleiten. So muss nur jeweils das höchste Level übertragen werden und die benötigte Bandbreite für die restlichen Level of Details kann eingespart werden.

Progressive Meshes [Hop96] erlauben es, ein 3D-Modell automatisch in mehrere Level of Detail Stufen zu zerlegen. Dem Modell werden dabei schrittweise Kanten entfernt oder hinzugefügt. Dabei entsteht bei jedem Schritt ein weniger detailliertes Modell. Der Einsatzzweck für diese Technik ist meist die dynamische Erstellung von Level of Details eines 3D-Modells für die Verwendung in einer komplexen Szene. Die Technik kann allerdings auch dafür genutzt werden, ein 3D-Modell schrittweise über eine Datenverbindung zu transferieren und beim Client einzeln wieder zusammenzusetzen. Da im optimalen Fall bereits sehr kleine Level of Detail Stufen die ungefähre Form des 3D-Modells erahnen lassen, hat dies den Vorteil, dass der Client bereits sehr früh einen Eindruck von dem 3D-Modell erfährt, bevor das ganze Modell heruntergeladen wurde.

5.1 Zerlegung des Meshes

Eine Schwierigkeit bei dieser Technik ist die Zerlegung des Meshes. Hierbei soll die Form des 3D-Modells auch in kleineren Level of Detail Stufen beibehalten werden, damit dieses weiterhin erkennbar bleibt. Dafür existieren in der Literatur verschiedene Ansätze wie *quadric error metric* [GH97] und *memoryless scheme* [LT98]. Da in dieser Arbeit der Fokus auf die Übertragung des 3D-Modells beim Client liegt, wird nur ein einfacher Algorithmus für die Zerlegung des Modells verwendet [Mel98], welcher lediglich Dreiecke verarbeitet. Er ist somit nur für bereits triangulierte Modelle anwendbar. Das Modell wird zerlegt, indem nacheinander einzelne Kanten entfernt werden. Die Auswahl einer Kante erfolgt nach fest definierten Kriterien. Ziel dabei ist es, die Kante zu entfernen, welche den kleinsten Einfluss auf das Aussehen des 3D-Modells besitzt.

Ein möglicher Algorithmus verfährt wie folgt: Für jede Kante wird im Vorfeld berechnet, wie sehr sich das Modell visuell verändert, sobald diese Kante entfernt wird. Dieser Wert wird als Kosten bezeichnet. Die Kosten einer Kantenentfernung setzen sich dabei aus dem jeweiligen gewählten Kriterium zusammen. Sobald die Kosten der einzelnen Kanten berechnet wurden, wird die Kante ausgewählt, welche die geringsten Kosten verursacht. Diese wird anschließend durch einen *Edge Collapse* entfernt und die Änderung notiert. Anschließend wird für die verbleibenden Kanten erneut eine Kostenberechnung durchgeführt. Dieser Kreislauf wird solange ausgeführt, bis alle Kanten des Modells verarbeitet wurden. Das Resultat des Algorithmus ist eine Liste von *Vertex Splits*, die verwendet werden können, um das Modell wieder zusammenzusetzen. Diese sind in Abschnitt 5.4 näher beschrieben. Für die Berechnung der Kosten existieren unterschiedliche Algorithmen, welche unterschiedliche Laufzeiten und Qualitäten aufweisen. In dieser Arbeit soll jedoch nur ein einfaches Kriterium umgesetzt werden, um das Verfahren hinreichend zu demonstrieren.

Da die Details des Meshes in der Regel in den jeweils kleinsten Dreiecken liegen, können die Kosten nach der Kantenlänge festgelegt werden. Damit wird bereits ein akzeptables Ergebnis erzielt. Weil die Kantenlänge sehr schnell ermittelt werden kann, ist dieser Algorithmus für eine schnelle Berechnung der

Kosten der Kanten geeignet. In dieser Arbeit soll dieses Kriterium nach Melax [Mel98] um eine weitere Bedingung erweitert werden. Dabei soll die Krümmung der Oberfläche in die Ermittlung der Kosten für eine Kante mit einfließen. Dies folgt aus der Erkenntnis, dass gekrümmte Flächen mehr Dreiecke und damit Kanten, benötigen als planare Flächen und somit möglichst spät entfernt werden sollten. Durch diese Erweiterung wird die Form des 3D-Modells bei der Zerlegung besser beibehalten.

Die konkrete Berechnung der Kosten einer Kante zwischen dem Vertex u und v, wobei u auf den Vertex v gezogen werden soll, erfolgt dabei nach folgender Formel:

$$cost(u,v) = \|u - v\| * max_{f \epsilon Tu} \{min_{n \epsilon Tuv} \{(1 - f.normal \cdot n.normal)\,/2\}\} \tag{5.1}$$

Wobei Tu eine Menge von Dreiecken ist, welche u enthält, und Tuv eine Menge von Dreiecken ist, welche u und v enthält.

Die Krümmung wird bestimmt, indem zwischen den anliegenden Flächen des Vertex u und den an der Kante anliegenden Flächen uv das Skalarprodukt gebildet wird. Damit werden Eckkanten teurer bewertet als Kanten, die in einer planaren Umgebung liegen. Zusätzlich fließt die Länge der jeweiligen Kante in die Berechnung der Kosten ein.

5.2 Datenstruktur

Die Geometriedaten von 3D-Modellen bestehen in der Regel aus einer Liste von Attributen und einer Indexliste, welche die einzelnen Dreiecke definiert. Dieses Format ist geeignet, um Modelldaten zur Grafikkarte zu übertragen. Allerdings kann nur schwer eine Zerlegung des Modells vorgenommen werden. Es liegen in dieser Repräsentation keine Informationen zu den Verbindungen der einzelnen Dreiecke und Vertices untereinander vor. Diese sind allerdings notwendig um beispielsweise alle anliegenden Dreiecke um einen einzelnen Vertex in kurzer Zeit zu finden. Diese Dreiecke können zwar gefunden werden, indem die Liste aller Dreiecke traversiert wird, jedoch hätte dies eine unzureichende Performance bei größeren Modellen zur Folge.

Aus diesem Grund wird in dieser Arbeit die Datenstruktur nach Melax [Mel98] implementiert. Die Datenstruktur ist beispielhaft im Codelisting 5.1 aufgeführt. Diese Struktur zerlegt das Modell in einzelne Vertices und Dreiecke. Die Vertices besitzen als Attribute die Position und eine Referenz auf die anliegenden Vertices und Dreiecke. Damit können in relativ kurzer Zeit die anliegenden Dreiecke eines Vertex ermittelt werden. Des Weiteren enthält der Vertex ein Attribut für die Kosten und eine Referenz auf den Vertex, mit welchen dieser zusammengeführt werden soll. Diese beiden Werte werden von dem, nachfolgend beschriebenen, *Half Edge Collapse*-Algorithmus verwendet. Die Dreiecke halten eine Referenz auf die drei Vertices, welche dieses definieren. Sie beinhalten die Normale und Texturkoordinate des Vertex innerhalb des Dreieckes.

5.3 Half Edge Collapse

Durch einen Edge Collapse [LRC+02] wird eine Kante des Meshes entfernt, indem beide Vertices der Kante zu einem neuen Vertex in der Mitte der Kante zusammengeführt werden. Dieser Vertex wird anschließend mit den Vertices verbunden, mit welchen die Kante vorher verbunden war.

Der Half Edge Collapse ist ein Spezialfall des Edge Collapse. Durch ihn wird die Kante uv, welche zwischen den beiden Vertices u und v liegt, entfernt, indem der Vertex u auf den Vertex v verschoben wird. Alle dabei resultierenden Änderungen an der Struktur des Meshes werden aufgezeichnet. Diese Information kann anschließend genutzt werden, um den Half Edge Collapse umzukehren.

Die Kante wird entfernt, indem im ersten Schritt die Attribute der um den Vertex u anliegenden Dreiecke aktualisiert werden. Die an den Vertex u angrenzenden Dreiecke werden über die Referenz ermittelt und in

```
function Triangle(v0, v1, v2, uv0, uv1, uv2, _id) {
  this.vertex = new Array;
  this.vertex[0] = v0; // Referenz auf die drei Vertices des Dreiecks
  this.vertex[1] = v1;
  this.vertex[2] = v2;
  this.uv = new Array;
  this.uv[0] = uv0; // Referenz auf die drei Texturkoordinaten des Dreiecks
  this.uv[1] = uv1;
  this.uv[2] = uv2;
  this.id = _id; // ID

  this.normal = vec3.create(); // orthogonaler Einheitsvektor
}

function Vertex(vec, _id) {
  this.position = vec;  // Position des Vertex [x,y,z]
  this.id = _id; // ID
  this.neighbor = new Array; // angrenzende Vertices
  this.face = new Array; // angrenzende Dreiecke
  this.cost = 0.0; // Kosten für Entfernung
  this.collapse = null; // Vertex zu dem der Vertex zusammengeführt werden soll
}
```

Listing 5.1: Datenstruktur: Das Modell wird in einzelne Vertices und Dreiecke zerlegt. Die Vertices beinhalten die Position sowie eine Referenz auf die anliegenden Vertices und die anliegenden Dreiecke. Die Dreiecke beinhalten die jeweiligen Attribute der verbundenen Vertices. Hier sind dies die Texturkoordinaten.

diesen die Attribute des Vertex u mit den Attributen des Vertex v aus einem der an der Kante anliegenden Dreiecks ersetzt. Bei Texturkoordinaten muss jedoch beachtet werden, das gegenüberliegende Dreiecke unter Umständen völlig andere Bereiche der Textur abrufen. Diese Ersetzung kann daher nur stattfinden, wenn der Vertex u des an der Kante anliegenden Dreieckes die gleichen Texturkoordinaten wie der Vertex v des anliegenden Dreieckes besitzt.

Im nächsten Schritt können die an der Kante anliegenden Dreiecke entfernt werden. Dabei werden die Flächen, welche als Vertex sowohl u als auch v besitzen, entfernt. Anschließend wird in allen restlichen umliegenden Flächen von u, der Vertex u durch v ersetzt. Abschließend wird u entfernt. Der Algorithmus ist im Listing 5.2 aufgeführt und in der Abbildung 5.1 visualisiert.

```
1. Aktualisiere die Attribute der um v und u anliegenden Dreiecke, welche nicht entfernt
    werden.
2. Entferne alle Dreiecke, welche an der Kante uv anliegen. Dies sind die Dreiecke, welche u
    und v als Vertices beinhalten.
3. Ersetze in den restlichen Dreiecken, welche u beinhalten, u durch v.
4. Entferne u.
```

Listing 5.2: Edge Collapse: Algorithmus zur Entfernung einer Kante, indem der Vertex u auf den Vertex v gezogen wird.

5.4 Vertex Split

Der Vertex Split ist der inverse Vorgang zum Edge Collapse. Durch diesen werden alle Änderungen des Edge Collapse umgekehrt. Das Modell kann also mittels einer Menge von Vertex Splits schrittweise wiederaufgebaut werden. Dieser Vorgang kann für den progressiven Zusammenbau des Meshes beim Client genutzt werden.

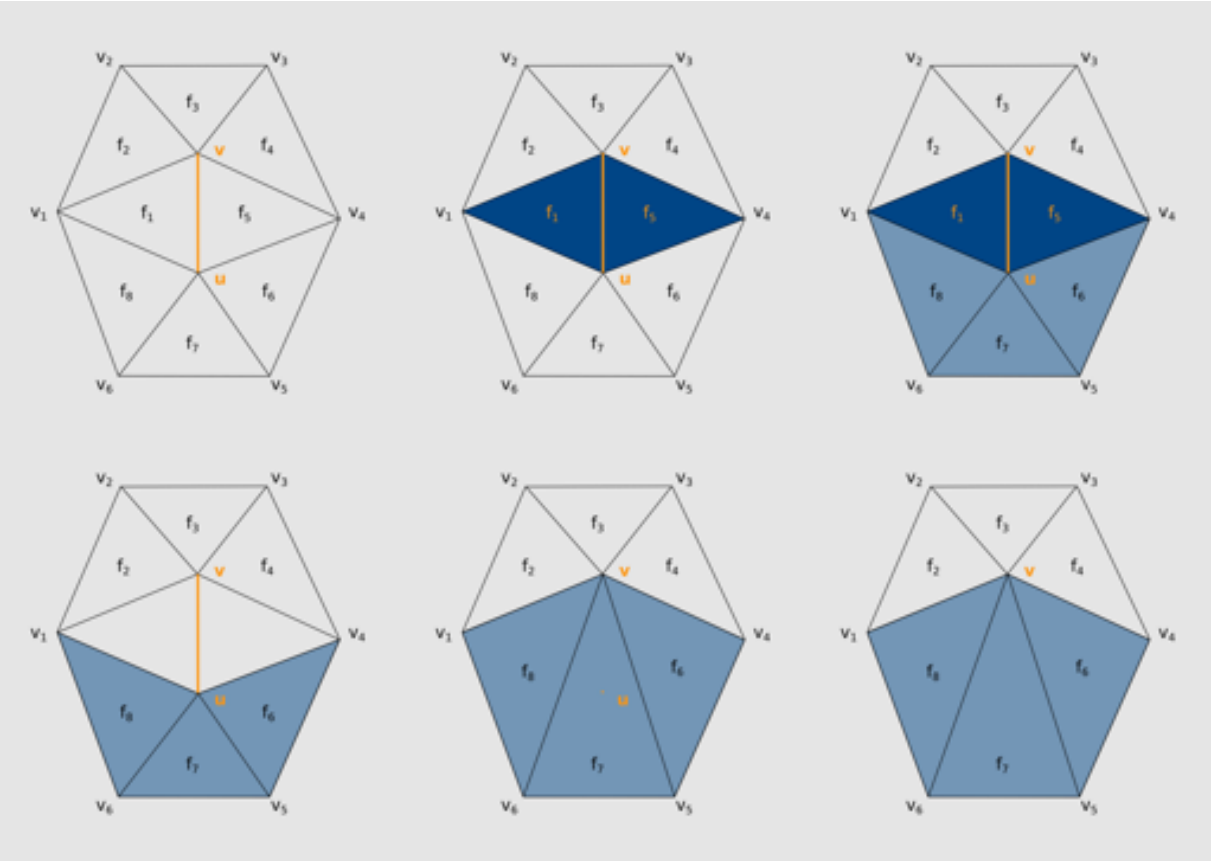

Abbildung 5.1: Edge Collapse: Es wird die Kante uv entfernt, indem der Vertex u auf die Kante v gezogen wird. Dabei werden die Flächen, welche als Vertex sowohl u als auch v besitzen, entfernt. Anschließend wird in allen restlichen umliegenden Flächen von u der Vertex u durch v ersetzt. Abschließend wird u entfernt.

Der Vertex Split muss alle Änderungen enthalten, welche beim Half Edge Collapse durchgeführt wurden. Alle Änderungen werden in der umgekehrten Reihenfolge ausgeführt. Im ersten Schritt wird der Vertex u wiederhergestellt. Dafür wird die Position und die ID des Vertex u benötigt. Anschließend werden die Dreiecke, welche beim Half Edge Collapse auf den Vertex v gezogen wurden, wieder dem Vertex u zugewiesen. Dafür werden alle IDs der Dreiecke notiert, für welche diese Änderung vorgenommen wird.

Im nächsten Schritt werden die gelöschten Dreiecke wiederhergestellt. Dafür werden die ID des Dreieckes, die drei IDs der Vertices und die drei Texturkoordinaten benötigt. Die IDs der Vertices werden benötigt, da die Normalen der Dreiecke nach jedem Split neu berechnet werden. Für diese Normalenberechnung ist die richtige Reihenfolge der Vertices des Dreieckes nötig, da andernfalls die Normale geflippt wird und somit in die falsche Richtung zeigen würde.

Letztendlich werden die alten Texturkoordinaten der mit u verbundenen Dreiecke wiederhergestellt. Dafür sind wiederum die IDs der mit u verbundenen Dreiecke und deren Texturkoordinaten notwendig. Alle nötigen Daten für den Vertex Split sind im Listing 5.3 aufgeführt.

```
VertexSplit(
u.xyz, u.id, // Position und ID des neuen Vertex
v.id, // ID von v
t1.id, t1.v1.id, t1.v2.id, t1.v3.id, t1.v1.uv, t1.v2.uv, t1.v3.uv, // Daten des Dreiecks t1
t2.id, t2.v1.id, t2.v2.id, t2.v3.id, t2.v1.uv, t2.v2.uv, t2.v3.uv, // Daten des Dreiecks t2
f1.id, f1.uv, f2.id, f2.uv, ... // Texturkoordinaten
f1.id, f2.id, f3.id, ... // Dreiecke, welche mit u verbunden waren
)
```

Listing 5.3: Aufbau des Vertex Splits mit allen nötigen Informationen, welche beim Edge Collapse ermittelt wurden.

5.5 Bewertung

Die Abbildung 5.2 zeigt diesen Algorithmus angewendet auf ein Modell von Lee Perry Smith. Bei diesem kann die ungefähre Form bereits bei einer kleinen Anzahl von Dreiecken erahnt werden. Ab ungefähr einem Zehntel der Übertragung ist das Modell bereits gut abstrahiert und ab der Hälfte werden lediglich Details hinzugefügt. Im Beispiel wurde Flat Shading verwendet, damit die übertragenen Details besser erkannt werden können. Durch die Verwendung von Smooth Shading kann noch früher ein guter visueller Eindruck erreicht werden.

Die prototypische Implementationen in dieser Arbeit zeigten, dass der Algorithmus jedoch lediglich für geschlossene Modelle ein gutes Ergebnis erzielt. Dies ist bei vielen Modellen jedoch nicht der Fall. So besteht zum Beispiel die Hinterhofszene aus mehreren zusammengesetzten Einzelmodellen, welche untereinander Materialien teilen. Die Abbildung 5.3 zeigt die progressiven Zwischenschritte des Algorithmus angewendet auf das Modell der Hinterhofszene. Da der Algorithmus nicht erkennt, welche Vertices zu einem geschlossenen Modell gehören, verliert die Szene in den kleinsten Zwischenstufen die ursprüngliche Form. Der Algorithmus muss für dieses Modell entsprechend abgeändert oder das Modell in alle einzelnen Untermodelle zerlegt werden. Des Weiteren lässt der in dieser Arbeit verwendete Algorithmus nur jeweils ein Material für das gesamte Modell zu, weshalb in der Abbildung 5.3 eine Platzhalter-Textur verwendet wird. Um mehrere Materialien zu unterstützen, muss ein Textur-Atlas verwendet oder abermals das Modell in die einzelnen geschlossenen Untermodelle zerlegt werden.

Die Dekodierungszeit beträgt, angewandt auf das Modell von Lee Perry Smith, ungefähr 180 ms für jeden progressiven Schritt. Bei einer durchschnittlichen Breitbandverbindung [OEC11] von 4,1 MBit/s muss das Modell daher größer als 92 kB sein, um einen progressiven Schritt anzuzeigen. Durch die relativ lange Dekodierungszeit muss die Dekodierung über einem Web Worker in einem separaten Thread ablaufen, da sonst keine 60 FPS aufrecht erhalten werden können.

Ziel der Technik war es, Bandbreite bei der Übertragung einzusparen, da lediglich das Modell mit dem höchsten Level of Detail übertragen wird. In der hier verwendeten Implementation werden jedoch viele redundante Daten innerhalb der einzelnen Vertex Splits abgespeichert, um den Zusammenbau beim Client zu beschleunigen. Dies führt allerdings dazu, dass die Dateigröße aller Vertex Splits größer als die Dateigröße aller komprimierten binären Level of Detail Modelle ist. Damit ist absehbar, dass die erhoffte Einsparung mit dieser Implementation nicht erreicht werden kann. Hierfür wäre ein verbesserter Algorithmus nötig, welcher redundante Informationen in den Vertex Splits vermeidet. In der Arbeit von Hoppe [Hop98] wurde eine effizientere Darstellung der Vertex Splits aufgezeigt, welche ebenfalls in DirectX 8.0 implementiert wurde. Diese Darstellung benötigt jedoch eine andere Repräsentation der Geometriedaten, als in dieser Arbeit verwendet wurde. Im Folgenden wird deswegen ein anderer Ansatz verfolgt, um die Geometriedaten effektiv zu übertragen.

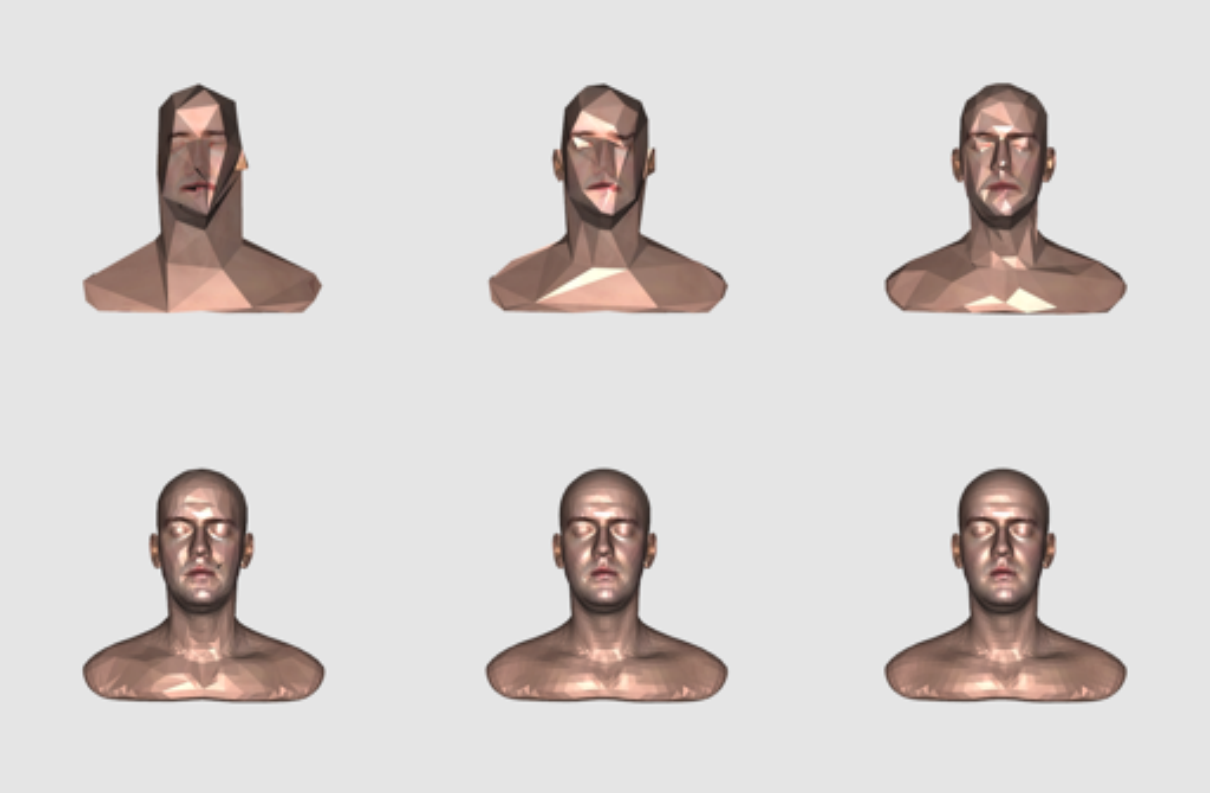

Abbildung 5.2: Die Abbildung zeigt ein Beispiel für einen Progressive Mesh für das 3D-Modell von Lee Perry Smith. Von links oben nach rechts werden aufsteigend die Zwischenstufen mit der Anzahl von Dreiecken (172, 348, 1.056, 4.944, 10.070 und 17.672 Dreiecke) gezeigt. ©Modell, Lee Perry Smith

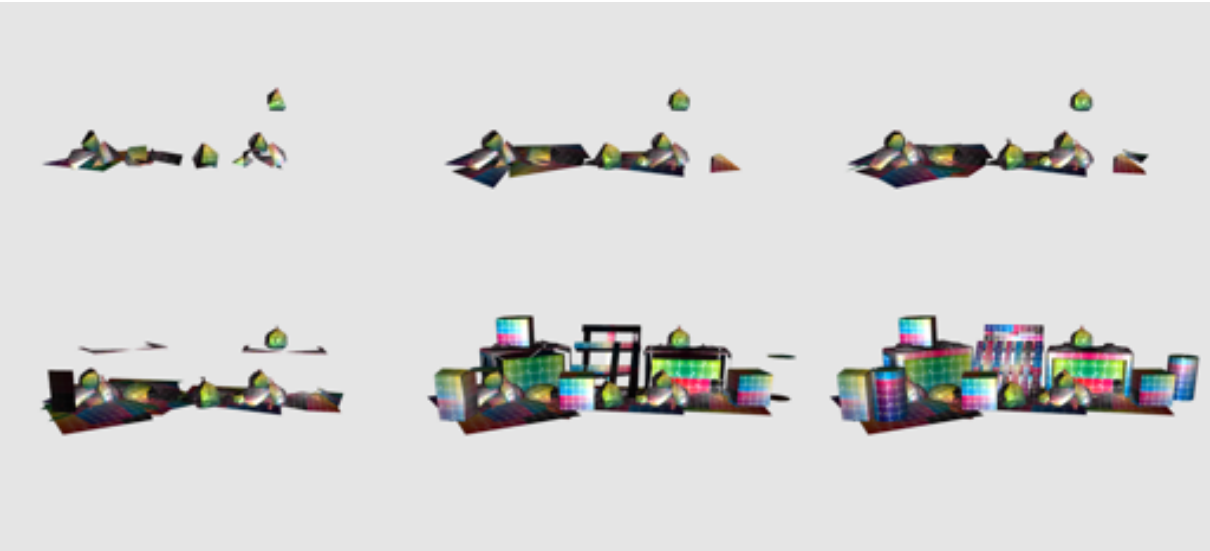

Abbildung 5.3: Die Abbildung zeigt den Progressive Mesh für die Hinterhofszene. Von links oben nach rechts werden aufsteigend die Zwischenstufen mit der Anzahl von Dreiecken (139, 279, 605, 1.117, 2.375 und 4.657 Dreiecke). ©Modell, Crytek GmbH

6 Komprimierung von 3D-Modellen

Für Anwendungen, welche besonderen Wert auf die Qualität der Darstellung eines Modells legen, ist die visuelle Qualität der einzelnen Level of Details eines Modells sehr wichtig. Aus diesem Grund werden die einzelnen Levels in der Regel nicht automatisiert erstellt, sondern speziell von einem Modellierer manuell aufbereitet. Dieser erstellt unter Umständen für ein grobes Level of Detail ein Modell, welches nur wenige Gemeinsamkeiten mit dem ursprünglichen Modell besitzt, dieses dafür allerdings sehr gut abstrahiert.

Diese Level of Details können genutzt werden, indem diese schrittweise zum Client übertragen werden. Dieser erhält zuerst das kleinste Level of Detail und kann dieses sofort anzeigen. Dieses ist in der Regel sehr klein und kann demnach in kurzer Zeit übertragen werden. Anschließend können schrittweise die nächsten Level of Details übertragen werden und das vorhergehende in der Darstellung ersetzen. Bei den in dieser Arbeit verwendeten Modellen ist das nächstgrößere Level of Detail doppelt so groß und benötigt also ungefähr die doppelte Zeit bis dieses übertragen wurde.

Damit die einzelnen Stufen möglichst kompakt übertragen werden, müssen diese vorher entsprechend komprimiert werden. Dazu sollen im Folgenden die Komprimierungsalgorithmen aus dem Einführungskapitel genutzt werden, um schrittweise die Daten zu komprimieren. Anschließend werden die beiden Kompressionsformate OpenCTM und WebGL Loader evaluiert. Abschließend wird ein Ansatz vorgestellt, welcher die Geometriedaten in Bilder abspeichert, um die Komprimierung von JPEG und PNG auszunutzen. Damit wird nur ein geringer Teil der Dekodierung in JavaScript ausgeführt und kann somit unabhängig von der jeweiligen Browserengine schnell ausgeführt werden.

6.1 Schrittweise Komprimierung

Dieses Kapitel beschreibt, wie das JSON Format mit den eingangs erwähnten Techniken schrittweise komprimiert werden kann. Als Testmodell dient dabei zum einen die Hinterhofszene aus der Abbildung 3.1, und zum anderen das Modell des Goldenen Reiters, welcher in der Abbildung 3.3 zu sehen ist. Durch die fehlende Cache-Optimierung bestehen zwischen beiden Modellen grundlegende Unterschiede bei der Effektivität der einzelnen Schritte. Die Ergebnisse jedes Schrittes für die beiden Modelle sind in Tabelle 6.1 und 6.2 aufgeführt.

Um eine bessere Vergleichbarkeit der jeweiligen Schritte zu erreichen, werden im Folgenden die Attribut-Daten vom Index getrennt betrachtet. Des Weiteren werden die Daten jeweils im Binär- und Textformat verglichen. Das Textformat kann, gegenüber dem Binärformat, für sehr kleine Integer-Variablen im Bereich von $0 - 9$ eine bessere Komprimierung bieten, da für ein ASCII Zeichen nur jeweils ein Byte, gegenüber vier Byte für eine 32 bit Integer-Variable, benötigt werden. Kleine 32 bit Integer-Werte besitzen in ihrer binären Repräsentation zwar viele führende Nullen, welche im Idealfall vom Kompressionsalgorithmus erkannt und effektiv komprimiert werden. Jedoch ist GZIP nicht geeignet, um diese sehr effektiv zu komprimieren. Eine Komprimierung mit Mustererkennung wäre hier wesentlich besser geeignet.

Im ersten Schritt werden die Attribute und der Index getrennt voneinander in zwei separaten JSON Dateien im Textformat abgelegt. Um den Overhead der Textrepräsentation zu umgehen, werden die Daten ebenfalls im Binärformat abgelegt. Die Attribute werden als 32 bit Float-Variablen und die Indices als 32 bit Unsigned-Integer-Variablen abgelegt. Die JSON und binären Daten werden anschließend mittels GZIP komprimiert.

6.1.1 Schritt 1: Optimierung des Index nach Forsyth

Die Ausgangsdaten des Goldenen Reiters sind nicht für eine gute Cache-Ausnutzung optimiert. Die Daten sollen in diesem Schritt entsprechend für den Cache optimiert werden. Da die Cache-Optimierung, wie bereits im Kapitel 4.3.1 beschrieben, die Daten so sortiert, dass ähnliche Attribute und Indices nebeneinander stehen, führt dies zu einer guten Komprimierung mit GZIP. Da die Daten lediglich sortiert werden, bleibt die Dateigröße für die unkomprimierten Daten gleich. Jedoch ändert sich die Dateigröße, sobald diese mit GZIP komprimiert werden, da nun Redundanzen zwischen den Daten besser ausgenutzt werden können. Für das Modell des Goldenen Reiters resultiert dies in einer Einsparung von ungefähr 50% des Speicherplatzes (Tabelle 6.1) für die Indices. Die Attribute profitieren ebenfalls im geringen Ausmaß von dieser Optimierung.

Für das Modell der Hinterhofszene kommt es nur teilweise zu einem Gewinn an Speicherplatz. Dies liegt an den bereits optimierten Geometriedaten. Zwar können die Index Daten zu einem kleinen Teil besser komprimiert werden, jedoch steigt dafür der Speicherverbrauch für die Attribute an. In der Summe kommt es im Binärformat zwar zu einem Gewinn, dennoch muss dieser Schritt vorab evaluiert werden. Die Cache-Optimierung sollte somit nur vorgenommen werden, sobald es zu einem messbaren Gewinn führt. Dazu muss die Optimierung durchgeführt und der Speicherplatzverbrauch mit dem vorhergehenden Daten verglichen werden.

Um zusätzlich die Attribute zu optimieren, kann das Modell in einzelne Segmente zerlegt werden. Für jedes Segment kann anschließend separat die Indexoptimierung nach Forsyth vorgenommen werden. Damit würde die Lokalität der einzelnen Vertices untereinander besser beibehalten werden. Die einzelnen Zellen können anschließend durch eine Z- oder Hilbert Kurve durchlaufen werden und so die komplette Indexliste bilden. Damit entsteht ein Kompromiss zwischen der Optimierung des Caches und der Lokalität der Attribute.

6.1.2 Schritt 2: Delta-Kodierung des Index

Da die Indexdaten für die Ausnutzung des Vertex-Caches optimiert wurden, liegt es nahe, dass diese gut Delta-Kodiert werden können. Durch die neue Indizierung der Indexliste besteht diese aus einer mehrheitlich aufsteigenden Zahlenfolge. Diese kann also gut durch eine Delta-Kodierung in einen kleineren Zahlenbereich überführt werden. Im Idealfall besteht die daraus entstandene Zahlenfolge aus vielen aufeinanderfolgenden Einsen, welche gut durch GZIP komprimiert werden können. Im Textformat hat dies den weiteren Vorteil, dass für Zahlen im Bereich von $[0, 9]$ nur jeweils zwei Zeichen, inklusive dem Trennzeichen, benötigt werden. Dementsprechend müssen nur zwei Byte statt vier Byte für eine 32 bit Integer-Variable verwendet werden. Unter Umständen kann das komprimierte Textformat nach diesem Schritt kleiner als das komprimierte Binärformat sein.

Dies ist auch tatsächlich der Fall für die Delta-Kodierten Indexdaten des Goldenen Reiters. Für diesen sind die komprimierten Indices im Textformat kleiner als die komprimierten Indices im Binärformat. Gleiches gilt ebenfalls für das Modell der Hinterhofszene. Das binäre Format nutzt für die Speicherung der Integer-Variablen jeweils 32 bit. Da nach einer durchgeführten Delta-Kodierung diese 32 bit für die meisten Indices wesentlich zu groß sind, entstehen viele redundante führende Nullen in der Byte-Repräsentation. Die Kompression durch GZIP entfernt diese führenden Nullen nicht effektiv. Ein Kompressionsalgorithmus welcher dies erkennt, erzielt hierbei ein besseres Ergebnis.

Um die Daten besser für GZIP aufzubereiten, können die Integer-Variablen, wie im Abschnitt 4.3.3 beschrieben, byteweise versetzt abgespeichert werden. Damit bilden die führenden Nullen längere Ketten, welche wiederum gut von GZIP komprimiert werden können. Alternativ kann die maximale Größe des Integer-Bereiches vorab bestimmt werden. Damit wird vermieden das unnötige führende Nullen in der Byte-Repräsentation erst abgelegt werden, was wiederum zu einer kompakteren Dateigröße führt. Im besten Fall benötigt die binäre Repräsentation immer weniger Speicherplatz als die Textrepräsentation.

6.1.3 Schritt 3: Attribute quantisieren und als 16 bit Short-Variable abspeichern

In der Regel wird die Präzision von 32 bit für Geometriedaten nicht benötigt. Im Kapitel 4.4 wurde bereits aufgezeigt, dass eine Präzision von 12 bit für beide Testmodelle ausreichend ist. Um eine allgemeine Aussage für die meisten Modelle treffen zu können, sollen im Nachfolgenden die Attribute der Testmodelle auf jeweils 16 bit komprimiert werden. Dies ist für die meisten Anwendungsfälle ausreichend. Im Idealfall bedeutet dies, dass ungefähr 50% des Speicherplatzes der Attribute eingespart werden können. Da im Folgenden alle 32 bit Float-Variablen zu 16 bit Short-Variablen transferiert wurden, ist dies für die binären Daten auch wie erwartet der Fall. Für die mit GZIP komprimierten Daten werden ungefähr 40% des Speicherplatzes nach der Quantisierung eingespart. Die für die Testmodelle ermittelten Werte sind in Tabelle 6.1 und 6.2 aufgelistet.

6.1.4 Schritt 4: Versetztes Abspeichern der Attribute

Durch die Cache-Optimierung kann unter Umständen ebenfalls die Lokalität der Attribute verbessert werden. Die Attribute werden dafür nach dem durch Forsyth optimierten Index angeordnet. Da die Indices auf eine möglichst häufige Wiederverwendung optimiert wurden, stehen ähnliche Attribute letztendlich ebenfalls nebeneinander. Damit kann versucht werden, diese Lokalität der Attribute auszunutzen und diese versetzt, wie in Abschnitt 4.3.2 beschrieben, abzuspeichern. Nach diesem Schritt stehen alle Komponenten der einzelnen Attribute hintereinander, was in einer besseren Komprimierung durch GZIP resultiert. Entgegen dieser Erwartung erhöht dieser Schritt allerdings die Dateigröße, anstatt diese zu verringern. Die Cache-Optimierung betrachtet lediglich die Indices und optimiert die Attribute nicht gut genug, um eine optimale Lokalität dieser zu erreichen. Dafür wäre ein anderer Optimierungsalgorithmus nötig, welcher einen Kompromiss zwischen der Optimierung der Lokalität der Attribute und der Cache-Optimierung der Indices vornimmt.

6.1.5 Schritt 5: Transformation der Attribute mittels Delta-Kodierung

Die Sortierung der Attribute kann weiter optimiert werden, indem zwischen den Attributen eine Delta-Kodierung vorgenommen wird. Zwar ist die Lokalität der Attribute nach der Cache-Optimierung nicht optimal, aber dennoch weisen die Komponenten untereinander einen Zusammenhang auf. Daher können die Attribute durch die Delta-Kodierung in einen kleineren Zahlenbereich überführt werden. Dies ist allerdings wiederum nur für das Modell des Goldenen Reiters der Fall. Die Hinterhofszene besitzt wenig Vertices, welche im Modell somit auch relativ weit entfernt liegen. Dadurch kann keine gute Delta-Kodierung, wie mit dem 3D-Scan in welchem viele Vertices nahe beieinander platziert sind, erreicht werden.

6.1.6 Bewertung

Die Abbildung 6.1 und 6.2 visualisieren den Effekt der einzelnen Schritte. Dabei ist zu erkennen, dass nicht alle durchgeführten Schritte die Datei verkleinern. Die Optimierung der Indexliste mit Forsyth und die anschließende Delta-Kodierung verkleinert diese um den Faktor 1 zu 2,5 (Hinterhofszene) beziehungsweise 1 zu 4 (Goldener Reiter). Bei Modellen, welche bereits Cache-Optimiert sind, führt die erneute Optimierung der Indices zu keinen nennenswerten Vorteil und kann bei einigen Modellen gar zu einem Anstieg der Dateigröße führen, da durch Forsyth nicht die optimale Cache-Optimierung erreicht wird. Bei diesen Modellen kann die Optimierung entfallen und stattdessen direkt die Delta-Kodierung durchgeführt werden. Durch die Quantisierung der Daten von 32 bit auf 16 bit können die Attribute in der Regel um den Faktor 1 zu 2 verkleinert werden. Die Quantisierung könnte weiter erhöht werden, da der Abschnitt 4.4 zeigte, dass für viele Modelle eine Präzision von 12 bit bereits ausreichend ist.

Das versetzte Abspeichern der Attribute reduziert das Modell nicht unbedingt. Dazu müssen die Attribute eine gute Lokalität untereinander aufweisen, also im Raum nahe beieinander angeordnet sein. Der hier verwendete Optimierungsalgorithmus beachtet allerdings lediglich die Indices und nicht die Position der Attribute im Raum. Deswegen reduziert die anschließende Delta-Kodierung der Attribute im Allgemeinen lediglich Modelle, welche viele kleine, nahe beieinanderliegende Dreiecke besitzen. Insbesondere bei niedrig aufgelösten Modellen, wie der Hinterhofszene, in denen die Vertices im Modell in der Regel weit verteilt sind, kommt es nicht zum gewünschten Effekt. Es muss also vorher evaluiert werden, auf welche Modelle die Kompression konkret angewendet werden soll.

Schrittweise Kompression des Modells Goldener Reiter (Werte in Byte)					
		JSON	JSON GZIP	BIN	BIN GZIP
Ausgangsdaten	Attribute	9.228.318	3.852.106	3.927.712	3.456.491
	Index	2.335.541	1.009.582	1.566.108	1.023.478
Optimiert nach Forsyth	Attribute	9.228.318	3.791.009	3.927.712	3.444.846
	Index	2.426.970	472.700	1.566.108	470.232
Delta-Kodierung der Indices	Attribute	9.228.318	3.791.009	3.927.712	3.444.846
	Index	1.029.690	265.810	1.566.108	329.512
Quantisierung zu 16 bit	Attribute	6.153.574	2.562.098	1.963.856	1.898.791
	Index	1.029.690	265.810	783.054	261.964
Versetzte Attribute	Attribute	6.153.570	2.545.149	1.963.856	1.931.733
	Index	1.029.690	265.810	783.054	261.964
Versetzen und Delta-Kodierung der Attribute	Attribute	4.669.526	1.978.757	1.963.856	**1.693.731**
	Index	1.029.690	265.810	783.054	**261.964**

Tabelle 6.1: Überblick der einzelnen Komprimierungsschritte und deren Einsparungen für das 3D-Scan Modell des Goldenen Reiters aus Abbildung 3.3.

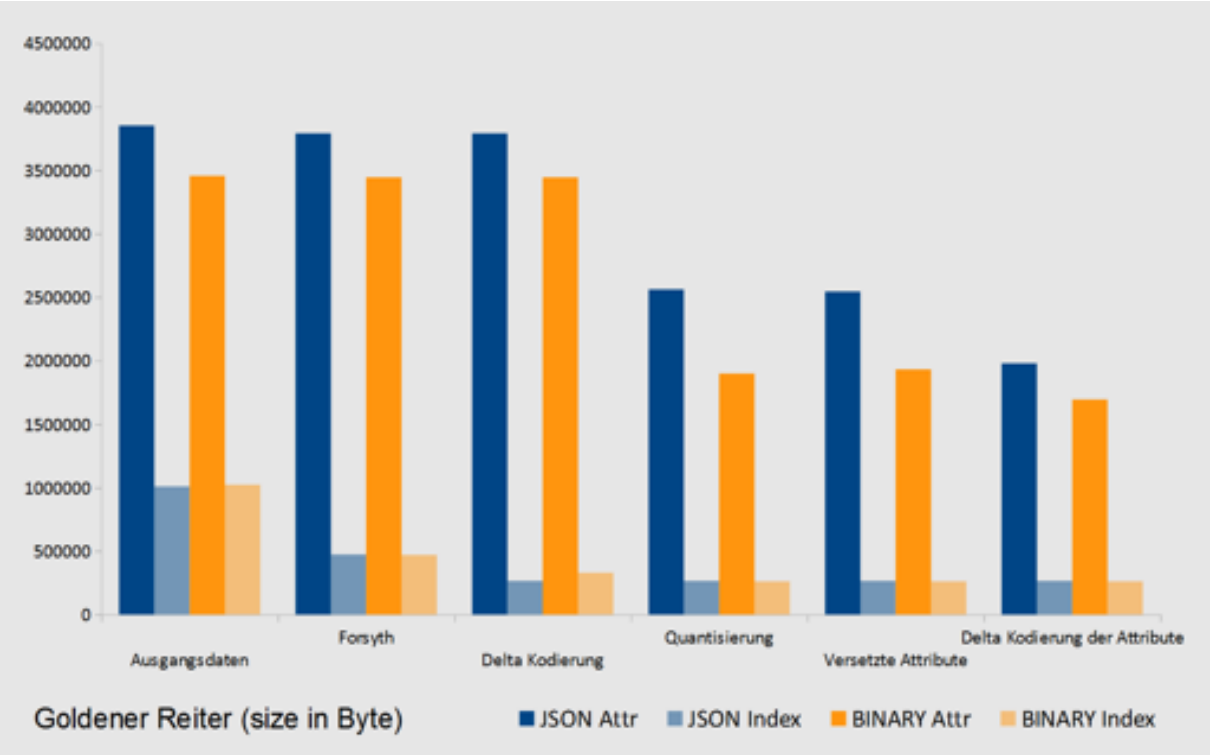

Abbildung 6.1: Das Diagramm visualisiert die einzelnen durchgeführten Schritte, um das Modell des Goldenen Reiters aus Abbildung 3.3 zu komprimieren. Sämtliche Daten wurden dabei bereits mit GZIP komprimiert.

Schrittweise Kompression des Modells Hinterhofszene (Werte in Byte)					
		JSON	JSON GZIP	BIN	BIN GZIP
Ausgangsdaten	Attribute	393.453	94.665	149.024	73.110
	Index	65.932	17.112	55.380	17.337
Optimiert nach Forsyth	Attribute	393.453	98.503	149.024	73.489
	Index	65.828	16.228	55.380	16.711
Delta-Kodierung der Indices	Attribute	393.453	98.503	149.024	73.489
	Index	34.257	6.969	55.380	8.666
Quantisierung zu 16 bit	Attribute	228.524	57.380	74.512	**42.271**
	Index	34.257	6.969	27.690	**7.060**
Versetzte Attribute	Attribute	229.948	62.979	74.512	50.233
	Index	34.257	6.969	27.690	7.060
Versetzen und Delta-Kodierung der Attribute	Attribute	175.290	62.458	74.512	51.512
	Index	34.257	6.969	27.690	7.060

Tabelle 6.2: Überblick der einzelnen Komprimierungsschritte und deren Einsparungen für die Hinterhofszene aus Abbildung 3.1.

Abbildung 6.2: Das Diagramm visualisiert die einzelnen durchgeführten Schritte, um das Modell der Hinterhofszene aus Abbildung 3.1 zu komprimieren. Sämtliche Daten wurden dabei bereits mit GZIP komprimiert.

6.2 OpenCTM

Das OpenCTM [Gee10a] Format wurde entwickelt, um Geometriedaten in einem offenen Format komprimiert abzulegen. Es unterstützt mehrere Modi, welche unterschiedliche Anwendungsfälle besitzen. Der RAW Modus dient dazu, die Daten unbearbeitet in einem Standard-Format auszugeben. Der MG1 Modus komprimiert die Geometriedaten verlustfrei. Der MG2 Modus bietet zwar ein gutes Kompressionsverhältnis, jedoch werden die Daten dazu quantisiert und so verlustbehaftet abgespeichert. Die Modi MG1 und MG2 verwenden den Lempel-Ziv-Markov-Algorithmus (LZMA) [Pav12], um die sortierten Geometriedaten abschließend zu komprimieren. Dieser Algorithmus erreicht höhere Kompressionsraten als GZIP benötigt allerdings auch eine längere Dekodierungszeit.

Die Bibliothek selbst ist in C++ geschrieben, kann jedoch über entsprechende Bindings auch mittels Python angesprochen werden. Es existiert des Weiteren eine JavaScript Bibliothek [Mel12], mit welcher das Format ausgelesen werden kann.

Für die Konvertierung von 3D-Modellen in das OpenCTM Format stellt die Bibliothek einen Konverter mit einer 3D-Vorschau des Modells zur Verfügung. Dieser ist in der Lage mehrere Formate, darunter auch OBJ, zu verarbeiten. Da der Konverter jedoch das OBJ Format falsch verarbeitet, sobald dieses getrennte Indices für die Attribute verwendet, wurden die Modelle in dieser Arbeit für die Konvertierung in das Stanford Triangle Format (PLY) konvertiert.

6.2.1 Aufbau des OpenCTM Formates

Ein OpenCTM Modell besteht aus einem Array von Positionen, Normalen und Texturkoordinaten. Das Format erlaubt zusätzliche selbst definierte Attribute als weitere Arrays hinzuzufügen. Die Arrays werden über eine Indexliste angesprochen, welche die einzelnen Dreiecke definiert. Somit bilden jeweils drei Indices ein Dreieck mit den entsprechenden Werten aus den jeweiligen Attribut-Arrays. Der Aufbau ist in Tabelle 6.3 und 6.4 aufgezeigt.

Jede OpenCTM Datei besitzt einen Header, in welchem der verwendete Modus und einige grundlegende Informationen, wie die Anzahl der Dreiecke und Vertices, angegeben sind. Danach folgen die Datenblöcke, in welchen sich die jeweiligen Attribute und die Indexliste befindet. Jeder Datenblock wird dabei durch ein entsprechendes Schlüsselwort eingeleitet.

Index	0	1	2	3	4	...	n
Vertex	v_0	v_1	v_2	v_3	v_4	...	v_n
Normal	n_0	n_1	n_2	n_3	n_4	...	n_n
UV	uv_0	uv_1	uv_2	uv_3	uv_4	...	uv_n

Tabelle 6.3: Alle Attribute der Vertices werden in entsprechenden Arrays abgelegt. Diese Daten werden später über den Index verwendet, um die Dreiecke zu bilden.

Triangle	t_0	t_1	t_2	t_3	t_4	...	t_n

Tabelle 6.4: Die Indexliste verwaltet, welche Vertices jeweils zusammen ein Dreieck bilden. Ein Dreieck besteht dabei aus einem Tripel, zum Beispiel $t_0 = (0, 1, 2)$, $t_1 = (0, 2, 3)$,...

6.2.2 RAW Modus

Der RAW Modus von OpenCTM speichert sämtliche Daten in einem Binärformat ab, ohne eine spezielle Kompression vorzunehmen. Sie dient dazu, diese in ein einfaches Format zu überführen. Damit kann diese Methode für einen Debug-Modus verwendet werden, um die Datenstruktur zu untersuchen. Die Dekodierungszeit für die Hinterhofszene beträgt circa 2 ms. Der Goldene Reiter benötigt ungefähr 410 ms zum Dekodieren. Die relativ hohe Dekodierungszeit ist dabei etwas überraschend, da die Daten bereits im Binärformat vorliegen und daher ohne jegliche weitere Dekodierungsschritte auskommen sollten. Jedoch wird in dieser Implementation eine externe Bibliothek für die Dekodierung der CTM Daten verwendet, welche für die Darstellung der Daten in ein anderes Format überführt werden muss. Deshalb sind beide Werte nur bedingt vergleichbar.

6.2.3 MG1 Kompression

Mit MG1 kann eine Komprimierung des Modells erreicht werden, ohne dabei die Qualität zu verringern. Die Methode versucht die Entropie der Indexdaten durch eine Neuanordnung dieser zu verringern. Damit können diese besser mit einem anschließenden Komprimierungsalgorithmus komprimiert werden. Durchschnittlich soll nach Aussagen des Autors von OpenCTM [Gee10a] das 3D-Modell auf 15% [Gee10b] der unkomprimierten Ausgangsgröße verkleinert werden.

Zuerst werden die einzelnen Tripel für jedes Dreieck so sortiert, dass jeweils der kleinste Index am Anfang des Tupels steht. Die Orientierung der Dreiecke wird dabei nicht geändert. Im nächstem Schritt werden diese Tupel nach ihrem ersten Index aufsteigend sortiert. Dies sorgt für eine bessere Kompression, da ähnliche Indices nun nebeneinander und nicht verteilt liegen. Das Listing 6.1 zeigt beispielhaft den Ablauf für vier Dreiecke.

Anschließend werden die Beziehungen der Tupel untereinander genutzt, um diese über eine Art Delta-Kodierung zu kodieren. Damit wird die Entropie weiter verringert. Das Kodieren der Indices erfolgt dabei folgendermaßen:

$$i'_{k,1} = \begin{cases} \mathrm{i}_{k,1} - i_{k-1,1} & k \geq 2 \\ \mathrm{i}_{k,1} & \text{sonst} \end{cases} \tag{6.1}$$

$$i'_{k,2} = \begin{cases} \mathrm{i}_{k,2} - i_{k-1,2} & k \geq 2, i_{k,1} = i_{k-1,1} \\ \mathrm{i}_{k,2} - i_{k,1} & \text{sonst} \end{cases} \tag{6.2}$$

$$i'_{k,3} = i_{k,3} - i_{k,1} \tag{6.3}$$

Die Daten können über den inversen Schritt wieder dekodiert werden:

$$i_{k,1} = \begin{cases} \mathrm{i}'_{k,1} + i_{k-1,1} & k \geq 2 \\ \mathrm{i}'_{k,1} & k = 1 \end{cases} \tag{6.4}$$

$$i_{k,2} = \begin{cases} \mathrm{i}'_{k,2} + i_{k-1,2} & k \geq 2, i_{k,1} = i_{k-1,1} \\ \mathrm{i}'_{k,2} + i_{k,1} & \text{sonst} \end{cases} \tag{6.5}$$

$$i_{k,3} = i'_{k,3} + i_{k,1} \tag{6.6}$$

Abschließend wird der modifizierte Index Array versetzt durch Element und Byte Interleaving abgespeichert und anschließend mit LZMA komprimiert. Die Attribute werden nicht weiter modifiziert, sondern direkt Element und Byte Interleaved abgespeichert und ebenfalls mittels LZMA komprimiert.

```
[[4, 0, 3], [4, 3, 7], [2, 6, 7], [2, 7, 3], ... ]

//Tupel sortieren, damit diese immer mit dem kleinsten Index beginnen.
//Die Orientierung wird dabei nicht geändert.
[[0, 3, 4], [3, 7, 4], [2, 6, 7], [2, 7, 3], ... ]

//Tupel aufsteigend nach ihrem ersten Index sortieren
[[0, 3, 4], [2, 6, 7], [2, 7, 3], [3, 7, 4], ... ]

//Delta nach Rechenvorschrift erstellen
[[0, 3, 4], [2, 4, 5], [0, 1, 1], [1, 4, 1], ... ]
```

Listing 6.1: Sortierung der Indexliste von OpenCTM MG1 am Beispiel von vier Dreiecken: Im ersten Schritt werden die Tupel sortiert, damit der jeweils kleinste Index dieses beginnt. Anschließend werden die Tupel aufsteigend sortiert und der Delta-Wert nach der Rechenvorschrift ermittelt.

6.2.4 MG2 Kompression

Das OpenCTM Format bietet einen weiteren Modus, welcher zusätzlich die Attribute der Geometriedaten komprimiert. Dieser Vorgang verringert die Präzision der Attribute und ist also verlustbehaftet. Dabei kann manuell bestimmt werden, wie hoch die Präzision eingestellt werden soll, oder diese Auswahl dem Konvertierungswerkzeug überlassen werden. Im manuellen Modus kann die Präzision in Meter angegeben werden. Dies eignet sich gut für 3D-CAD Modelle, von welchen im Vorfeld bekannt ist, dass diese von den Maschinen nur bis zu einer bestimmten Größe verarbeitet werden können. Im automatischen Modus wird abhängig von der durchschnittlichen Eckenlänge die nötige Präzision ermittelt.

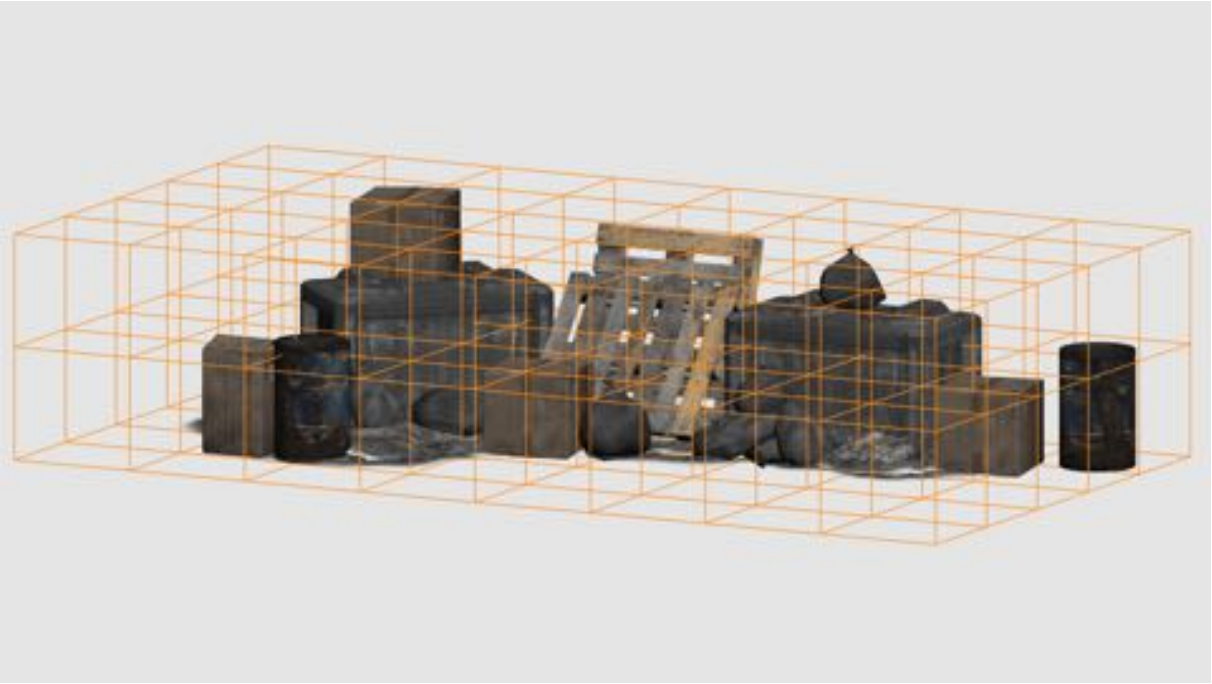

Abbildung 6.3: Das Modell wird in einzelne Segmente zerlegt. Die Vertices werden abhängig von der Position einer Zelle zugeordnet, welche einen eindeutigen Index besitzen. ©Modell, Crytek GmbH

Für die Kompression der Attribute wird das gesamte Modell in einzelne 3D-Segmente zerlegt. Die Abbildung 6.3 zeigt dabei die Unterteilung für das Modell der Hinterhofszene. Jede Zelle erhält einen eindeutigen Index. Die Koordinaten der Attribute innerhalb einer Zelle werden relativ zu dem Ursprung

dieser kodiert. Anschließend werden die Attribute innerhalb der Zelle entsprechend ihrer X-Koordinate sortiert. Nach dieser Sortierung können die X-Koordinaten effektiver Delta-Kodiert abgespeichert werden. Ebenfalls Delta-Kodiert werden die Zellen-Indices, die Normalen und die Texturkoordinaten.

6.2.5 Bewertung

Die Konvertierung des Modells kann entweder über ein Kommandozeilenprogramm durchgeführt werden oder eine Python beziehungsweise C API verwendet werden. Von dem Programm werden bereits zahlreiche Modellformate unterstützt, womit es möglich ist vorhandene Modelle in das Format zu konvertieren. Durch die Schnittstellen kann das Format darüber hinaus in eine vorhandene Produktionspipeline eingebaut werden.

Die Geometriedaten werden im MG1 Modus im Schnitt um den Faktor 1 zu 9 gegenüber den unkomprimierten JSON Daten verkleinert (ermittelt aus Tabelle 8.1 und Tabelle B.2). Sobald die JSON Dateien jedoch mittels GZIP komprimiert werden, reduziert sich der Kompressionsfaktor auf 1 zu 2. Gegenüber den mit GZIP komprimierten Binärformat ist die Kompressionsrate sogar teilweise kleiner. Lediglich bei sehr hoch tesselierten Modellen kann eine erkennbare Einsparung erzielt werden. Diese Einsparung ist gegenüber dem benötigten Implementierungsaufwand und der Dekodierungszeit jedoch zu vernachlässigen. Interessant wäre ein Vergleich des MG1 Modus mit anschließender GZIP Kompression statt der angewendeten LZMA Kompression. Damit könnte festgestellt werden, wie viel Speicherplatz durch die Sortierung eingespart werden konnte. Die bisherigen Ergebnisse legen nahe, dass diese Einsparung nur minimal ausfällt, da in der Regel die LZMA Kompression der GZIP Kompression überlegen ist. Trotzdem ist die erreichte Kompressionsrate nicht höher als des mit GZIP komprimierten Binärformates, womit die Sortierung theoretisch zu einem Anstieg der Dateigröße führte. Dieser Vergleich konnte jedoch nicht aufgestellt werden, da dafür das OpenCTM Format intern abgeändert werden müsste.

Obwohl der MG2-Algorithmus stärker komprimiert als der MG1-Algorithmus, kann die Dekodierung schneller durchgeführt werden als beim MG1-Algorithmus. Durch die manuell einstellbare Präzision kann das Modell zusätzlich entsprechend dem Anwendungsfall aufbereitet werden. Jedoch können durch die Sortierung der Attribute keine Modelle mit mehreren Materialien in diesem Modus verwendet werden. Die Sortierung führt dazu, dass die alten Materialinformationen nicht länger genutzt werden können. Zu erkennen ist dies am Modell der Hinterhofszene, welches nach der Konvertierung falsche Materialien zugewiesen bekommt. Dieser Effekt ist in Abbildung 6.4 zu sehen. Modelle mit mehreren Materialien müssen entweder in die einzelnen Materialien zerlegt werden oder für alle Texturen ein einziger Texturatlas verwendet werden.

Das OpenCTM Format wurde nicht für die Anwendung mit WebGL entwickelt. Die in dem Modi MG1 und MG2 verwendete LZMA Komprimierung kann zwar relativ schnell in nativen Code durchgeführt werden, in JavaScript benötigt diese Dekodierung, wie im Abschnitt 8.2 noch beschrieben wird, jedoch viel Zeit. In einigen Fällen führt dies dazu, dass die Gesamtperformance schlechter als bei einem unkomprimierten Modell ist. Die Dateigröße ist durch die LZMA Komprimierung zwar relativ klein und das Modell kann deswegen schneller heruntergeladen werden. Durch die hohe Dekodierungszeit muss der Nutzer, bei einer schnellen Verbindung, dennoch auf die Anzeige des Modells warten. Somit sollte diese Komprimierung nur verwendet werden, falls der Datentransfer sehr teuer ist und jeder gespeicherte Byte zählt. Eine mögliche Verbesserung kann erzielt werden, indem die native Komprimierung des Browsers mittels GZIP an Stelle der LZMA Komprimierung verwendet wird. Damit würde zwar die Komprimierung schlechter, dafür allerdings die Dekodierungszeit wesentlich schneller durchgeführt werden können.

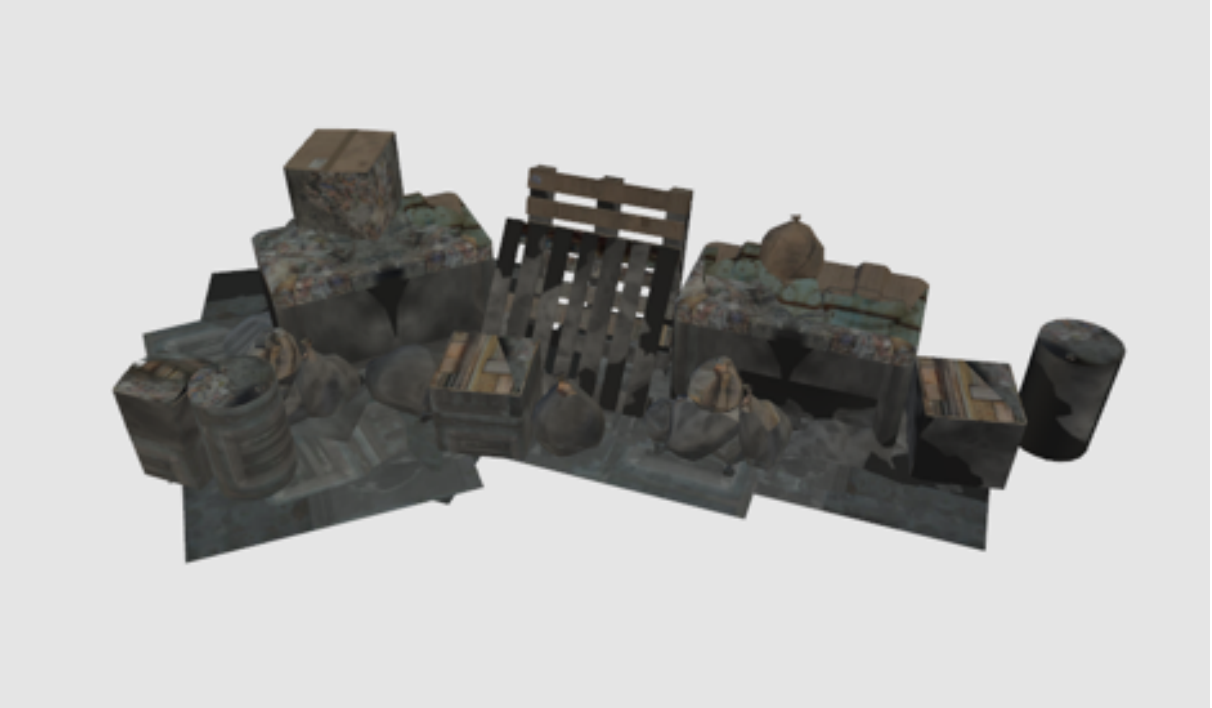

Abbildung 6.4: Im MG2 Modus des OpenCTM Formates werden die Indices und Attribute so sortiert, dass die ursprüngliche Materialinformationen nicht länger verwendet werden können. Werden dennoch die alten Materialinformationen verwendet, führt dies zu einer falschen Darstellung des Modells. ©Modell, Crytek GmbH

6.3 WebGL Loader

Die erste Version des WebGL Loader Formates wurde von Won Chun et al. [Chu12] speziell für den Anwendungsfall der Webanwendung Google Body, mittlerweile Zygote Body [Zyg12], konzipiert. Das Format befindet sich zur Zeit in der Entwicklungsphase, mit dem Ziel die Kompression und Stabilität weiter zu verbessern. In dieser Arbeit wird auf die Kompressionsmethode eingegangen, welche in Revision 50 des Formates verwendet wurde.

Im Gegensatz zu OpenCTM wurde das Format speziell für die Übertragung von 3D-Modellen mit den Einschränkungen von WebGL und JavaScript konzipiert. Es versucht potentielle Einschränkungen bei der Verwendung von JavaScript zu umgehen. So wird unter anderem die abschließende Komprimierung nicht über eine eigene Implementation durchgeführt, sondern die native GZIP Komprimierung des Browsers verwendet. Zwar liefert diese nicht die beste Kompressionsrate, kann dafür aber sehr schnell durchgeführt werden, da native Funktionen des Browsers verwendet werden.

6.3.1 UTF-8 zur Speicherung binärer Daten

In der Webanwendung Google Body wurde aus Kompatibilitätsgründen *XmlHTTPRequests* in der Version 1, statt Version 2, verwendet. Deshalb konnten zur Übertragung der Geometriedaten keine Typed Arrays verwendet werden. Aus diesem Grund musste für das Format eine Lösung gefunden werden, um binäre Daten effektiv an den Client zu übertragen. Das Format sieht vor, Daten in das UTF-8 Format (Universal Character Set Transformation Format - 8 bit) zu kodieren und diese anschließend als UTF-8 String an den Client zu senden. Das UTF-8 Format wurde von Ken Thompson entwickelt. Es ist abwärtskompatibel zu ASCII und kodiert alle 1.114.112 Unicode Code Points.

Diese Code Points sind Integer-Zahlen, welche einem bestimmten Zeichen zugewiesen sind. Dabei werden abhängig vom zu speichernden Zeichen unterschiedlich viele Bytes verwendet. Zeichen am Anfang des Unicode Sets verbrauchen dabei weniger Bytes als nachfolgende. Am Anfang des Unicode Sets befinden sich die ASCII Zeichen. Für diese ist die Repräsentation als Bitfolge daher in beiden Repräsentationen gleich. Eine Übersicht für den Speicherverbrauch der Kodierung von 16-bit Zeichen nach UTF-8 ist im Listing 6.2 aufgezeigt. Zu beachten ist dabei, dass der Bereich im Intervall $[55.296, 57.343]$ nicht verwendet werden kann. Diese sogenannten *Surrogate Pairs* sind für die Verwendung in UTF-16 reserviert. Die UTF-8 Zeichen werden in einem präfixfreien Bytecode abgespeichert. Somit können nicht die kompletten Bytes verwendet werden, da diese, je nach verwendeten Werte-Bereich, führende Byte-Flags besitzen.

```
[    0,   127] -> 1 Byte (1  bit flag,  7 bits data) -> 0xxxxxxx
[  128,  2047] -> 2 Byte (5 bits flag, 11 bits data) -> 110xxxxx 10xxxxxx
[ 2048, 65536] -> 3 Byte (8 bits flag, 16 bits data) -> 1110xxxx 10xxxxxx 10xxxxxx
[55296, 57343] -> Surrogate-Pairs
```

Listing 6.2: UTF-8 verwendet abhängig vom zu speichernden Zeichen unterschiedlich viele Bytes. Dabei wird ein präfixfreier Bytecode verwendet.

Diese variable Kodierung kann nun ebenfalls verwendet werden, um Geometriedaten effektiv abzuspeichern. Wenn beispielsweise ein Index gespeichert werden soll, so wird das jeweilige UTF-8 Zeichen genutzt, welches diesen Integer-Wert repräsentiert. Mit UTF-8 können ungefähr 20 bit (genauer 1.114.112 Zeichen und somit $log(1.114.112)/log(2)$ bit) abdeckt werden. Es ist deshalb nicht möglich 32 bit Werte direkt abzuspeichern. Die Daten müssen vorher entsprechend quantisiert und als Integer-Variablen abgelegt werden.

UTF-8 benötigt je nach Größe des Zeichens zwischen 1 und 3 Byte. Für größere Zeichen werden von UTF-8 mehr Bytes gegenüber einer nicht variablen Kodierung benötigt. Eine Datei im UTF-8 Format in der lediglich sehr große 16 bit Integer-Variablen abgelegt werden, ist deswegen größer als eine Datei, in welcher direkt 16 bit Integer-Variablen abgespeichert werden. Dagegen ist es von Vorteil, falls möglichst viele Zeichen im Bereich von $[0, 127]$ abgelegt werden, da für diesen Bereich lediglich 1 Byte, gegenüber 2 Byte für eine 16 bit Unsigned Integer-Variable, benötigt wird.

Aus diesem Grund kombiniert das WebGL Loader Format diese Technik mit der Delta-Kodierung aller Attribute und Indices. Durch diese Kodierung werden zumindest für die Indices viele sehr kleine Zeichen abgespeichert. Damit kann es durch die Verwendung von UTF-8 als binäres Format zu einer insgesamt kleineren Datei kommen als durch die Verwendung eines Binärformates.

Die Kodierung erfolgt, indem für die zu kodierende Integer-Zahl die Bitfolge des UTF-8 Zeichens ermittelt wird. Dabei muss zuerst die Präfix Bitfolge ermittelt werden, welche sich nach der Größe der Integer-Zahl richtet. Sollen nur 7 bit kodiert werden, besteht die Präfix Bitfolge aus lediglich einer 0. Die restlichen 7 bit werden zum Speichern der Integer-Variable verwendet. Für Zahlen zwischen $[128, 2.047]$ müssen zwei Byte verwendet werden. Das erste Byte bekommt die Bitfolge 110 und das zweite Byte die Bitfolge 10 als Präfix. Die restlichen 11 bit werden zum Speichern der Zahl verwendet. Für Zahlen zwischen $[2.048, 65.536]$ werden letztendlich drei Byte verwendet, wobei das erste Byte das Präfix 1110 und die beiden folgenden Bytes das Präfix 10 zugeordnet bekommen. Insgesamt stehen so 16 bit zur Verfügung. Wie bereits erwähnt, muss beachtet werden, dass im Bereich von $[55.296, 57.343]$ keine Zeichen abgelegt werden können. Dieser Bereich ist zur Kodierung von UTF-16 Zeichen reserviert, die Daten müssen in diesem Bereich entsprechend abgeschnitten werden.

Die Dekodierung von UTF-8 String erfolgt in JavaScript über die Funktion *charCodeAt()*. Diese Funktion ermittelt für jedes Zeichen den entsprechenden Code und damit die abgespeicherte Integer-Variable. Damit kann der übergebene UTF-8 String iterativ durchlaufen und alle Geometriedaten ausgelesen werden. Für die Dekodierung muss in JavaScript somit keine gesonderte Funktion geschrieben werden.

6.3.2 Kompressionsalgorithmus

Der Algorithmus verfährt im Wesentlichen wie im Abschnitt 6.1 beschrieben. Das Format quantisiert ebenfalls die Attribute der Geometriedaten, allerdings nicht uniform. Die Positionen werden zu 14 bit quantisiert, die Normalen und Texturkoordinaten zu jeweils 10 bit. Für die Quantisierung wird vorerst eine Bounding Box des Modells ermittelt. Die Ausmaße dieser Box werden anschließend separat abgelegt und verwendet, um die Float-Variablen der Attribute in entsprechend quantisierte Integer-Variablen zu überführen. Danach wird die Indexliste ebenfalls nach dem Algorithmus von Forsyth optimiert und die Attribute entsprechend sortiert. Die zu Integer-Variablen quantisierten Attribute werden versetzt abgespeichert und zusammen mit der Indexliste Delta-Kodiert abgelegt.

Die quantisierten Integer-Variablen werden in UTF-8 Zeichen kodiert und abgespeichert. Für Modelle, welche mehr als 16 bit Integer-Variablen als Indices adressieren und so über 65.536 Indices besitzen, wird das Modell in mehrere UTF-8 Dateien zerlegt. Das Modell wird ebenfalls in mehrere UTF-8 Dateien zerteilt, wenn es unterschiedliche Materialien verwendet. Für jede UTF-8 Datei werden zusätzlich die entsprechenden Metadaten, welche die Bounding Box Größen enthalten, abgelegt. Mit Hilfe dieser Bounding Box können die Geometriedaten wieder in den alten Bereich zurückgeführt werden.

6.3.3 Bewertung

Die 3D-Modelle müssen vorab mit einem Konverter vom OBJ Format in das UTF-8 Format überführt werden. Dieser Konverter abhängig von der Größe des Modells unterschiedlich lange zum Kodieren. Für sehr große Testmodelle schlug die Konvertierung sogar ganz fehl. Das Format befindet sich noch in der Entwicklung, womit die Konvertierungswerkzeugs ebenfalls noch nicht ausgereift sind. Zusätzlich steht der Source-Code des Konverters zur Verfügung, so dass mit Aufwand auch weitere Modellformate unterstützt werden können.

Das Modell der Hinterhofszene wird mit dem Format und der anschließenden GZIP-Komprimierung auf 46 kB verkleinert. Dies entspricht einer Komprimierung um den Faktor 1 zu 2 gegenüber den mit GZIP komprimierten Binärformat. Für das Modell des vereinfachten Goldenen Reiters wird eine Komprimierung um den Faktor 1 zu 2,5 auf 1.285 kB erreicht. Der selbe Faktor wird mit dem ursprünglichen Modell des Goldenen Reiters erreicht. Dies deckt sich mit den Ergebnissen für die schrittweise Komprimierung aus Abschnitt 6.1, wobei mit diesem Format eine höhere Komprimierung durch die höhere Quantisierung der Daten erreicht wird. Mit einer Quantisierung auf 11 bit für alle Attribute in der schrittweisen Komprimierung ergeben beide Algorithmen ein ähnliches Resultat.

Das Format soll laut Aussage von Won Chun et al. schrittweise verbessert werden. Eine mögliche Verbesserung wäre, statt der einfachen Delta-Kodierung der Attribute, eine Parallelogram-Prediction [IIGS05] des nächsten Vertex zu verwenden. Diese Technik würde die letzten drei Vertices nutzen, um die Daten des nächsten Vertex zu bestimmen. Für bestimmte Modelle soll so eine höhere Kompressionsrate der Attribute erreicht werden. Allerdings kann diese Herangehensweise auch die Abschätzung verschlechtern. Dies ist zum Beispiel der Fall, sobald bei einem Würfel die Kante gewechselt wird. In diesem Fall liegt der neue Vertex in einer anderen Ebene als die beiden alten Vertices und die Vorhersage wird demnach wahrscheinlich schlechter als eine einfache Delta-Kodierung.

Da das Format keine progressive Vorschau unterstützt, müssen für diese, alle Level of Detail Stufen des Modells komprimiert und schrittweise übertragen werden. Dafür muss deshalb ein zusätzlicher Wrapper programmiert werden, welcher beim Client die einzelnen Levels anfordert und anzeigt. Die Dekodierungszeit für das Laden der Hinterhofszene im höchsten Level of Detail beträgt circa 1 ms. Der Goldene Reiter benötigt ungefähr 72 ms. Die Zeiten sind sehr viel schneller als die Dekodierungszeiten des MG2 Modus von OpenCTM. Ein Vergleich dieser befindet sich in Abschnitt 8.2. Der Kompressionsfaktor ist etwas schlechter gegenüber dem MG2 Modus von OpenCTM, jedoch kann durch die schnelle Dekodierungszeit bei besonders schnellen Verbindungen ein Vorteil erreicht werden.

7 Image Geometry

Die Hauptaufgabe von Browsern ist die Anzeige von Text und Bildern. Deshalb sind diese für die Darstellung beider Medien optimiert. Der Browser erlaubt die Anzeige und das Herunterladen von sehr vielen Bildern in relativ kurzer Zeit. Die Image Geometry Technik nutzt diese Eigenschaft, um die Geometriedaten der 3D-Modelle effektiv abzuspeichern.

Die hier aufgeführte Technik basiert auf der vom Fraunhofer IGD [Beh11] vorgestellten Technik. Die Implementation des Fraunhofer IGD nutzt das Framework X3DOM, welches erlaubt 3D-Inhalte, über eine XML Sprache zu definieren. Im Gegensatz dazu, kommt die in dieser Arbeit angepasste Technik ohne dieses Framework aus.

7.1 Algorithmus

Die Verwendung von Bildern bringt einige Vorteile, so kann die native Kompression der Bildformate genutzt werden, um eine hohe Komprimierung bei einer gleichzeitig sehr schnellen Dekomprimierungsgeschwindigkeit zu erreichen. Das Bild wird nach der Dekomprimierung direkt als Textur an die Grafikkarte geschickt. Womit es keine weitere Bearbeitung der Daten in JavaScript benötigt, welche je nach verwendeten Browser unterschiedlich lange dauern kann.

Für diese Arbeit wurde ein Exporter für das 3D-Modellierungsprogramm *Blender* geschrieben, der einzelne 3D-Modelle in das Image Geometry Format exportieren kann. Dieser unterstützt mehrere Materialien, welche durch je eine diffuse Textur definiert sind.

7.1.1 Kodierung

Der Algorithmus nutzt die einzelnen Farbkanäle des Bildes, um die Geometriedaten dort abzuspeichern. In diesen werden, anstatt der Farbwerte, die 3D-Vektoren abgelegt. Da jeder Farbkanal Integer-Werte im Bereich von $[0, 255]$ speichern kann, stehen pro Farbkanal insgesamt 8 bit zur Verfügung. Diese Kanäle können unterschiedlich genutzt werden, um die Geometriedaten abzulegen. Im Folgenden wird für jedes Attribut jeweils ein separates Bild genutzt, damit steht für ein Attribut, falls der Rot-, Grün-, Blau- und Alphakanal genutzt wird, jeweils 32 bit zur Verfügung. Da die Ursprungsdaten für ein einzelnes Positionsattribut 96 bit (für jede Komponente 32 bit) verwenden, müssen die Daten relativ stark quantisiert werden, um sie in einem einzelnen Bild abzuspeichern.

Jedes Attribut nutzt dabei jeweils ein separates Bild. In dieser Implementation wird der Alphakanal nicht beachtet, womit statt 32 bit nur noch 24 bit zur Verfügung stehen. So verringert sich die Dateigröße des Bildes und die Verwendung von JPEG Bildern, welche keinen Alphakanal besitzen, wird möglich. Jeder Pixel des Bildes speichert jeweils ein Attributs-Vektor. Die X-, Y- und Z-Komponente wird so im Rot-, Grün- und Blaukanal des zugeordneten Pixels abgespeichert. Die Vektoren werden Pixel für Pixel im Bild abgelegt. Die Positionen und Normalen belegen dabei alle drei Farbkanäle. Für die Texturkoordinaten wird nur jeweils der Rot- und Grünkanal verwendet, da diese lediglich aus einem 2D-Vektor bestehen.

Die Indices werden ebenfalls in einem separaten Bild abgelegt. Jeder Pixel speichert dabei im Rot- und Grünkanal die Pixelposition, an welcher das jeweilige zugehörige Attribut in seinem Bild abgelegt wurde. Die Attribute werden Pixel für Pixel in der gleichen Reihenfolge des Attribut-Array von links oben nach rechts unten abgelegt. Daher muss lediglich die Größe der Attribut-Bilder berücksichtigt werden, um den

Index einer Pixelposition zuzuordnen. Für ein 256×256 großes Bild ergibt sich die X- und Y-Koordinate aus den folgenden Formeln:

$$x = Index \mod (256) \tag{7.1}$$

$$y = \lfloor Index/256 \rfloor \tag{7.2}$$

7.1.2 Dekodierung

Um die Geometrieinformationen auszulesen, wird das Index-Bild Pixelweise durchlaufen. Dazu muss ein VBO erstellt werden, welcher die einzelnen Pixel des Index-Bildes nacheinander ausliest. Der VBO enthält Tupel von Pixelpositionen, welche das gesamte Index-Bild abdecken. Er wird mit den Image Geometry Bildern an die Grafikkarte übertragen. Das Vertex-Shader-Programm ermittelt anschließend mit Hilfe des aktuellen Tupels die jeweiligen Attribute. Das Codebeispiel 7.1 zeigt, wie aus den Image Geometry Bildern die Position des Vertex ermittelt wird. Um nicht den Rand eines Pixels auszulesen, wird vorerst die Größe eines halben Pixels errechnet, welcher sich nach der Größe der verwendeten Attribut-Bilder richtet.

Aus dem Index-Bild wird der Rot- und Grünkanal an der Position des aktuellen Tupels ausgelesen. Beide Kanäle enthalten die Pixelposition, an denen die Attribute gespeichert sind. Um die Mitte der Pixel auszulesen, muss die ermittelte Pixelposition skaliert und der halbe Pixel als Offset hinzugefügt werden. Anschließend kann an dieser Position das Attribut-Bild ausgelesen und so das Attribut, welches zu dem aktuellen Vertex gehört, ermittelt werden. Das Attribut wurde durch die Quantisierung auf einen Wert zwischen $[0, 255]$ transferiert. Mit der Bounding Box der Ursprungsdaten muss aus diesen Grund die Position in den Ursprungsbereich transferiert werden.

```
1 vec2 halfPixel = vec2(0.5 / 256., 0.5 / 256.);
2 vec2 index = texture2D(uIndexTex, aPosition).rg * (255./256.) + halfPixel;
3 vec3 pos = texture2D(uPositionTex, vec2(index.r, index.g)).rgb;
4 pos.x = xMin + (xMax - xMin) * pos.x;
5 pos.y = yMin + (yMax - yMin) * pos.y;
6 pos.z = zMin + (zMax - zMin) * pos.z;
```

Listing 7.1: Auslesen der Attribute aus den Image Geometry Bildern im Vertex Shader: Dazu wird zuerst aus der Index-Textur die Pixel-Position ermittelt, welche angibt, an welcher Stelle das jeweilige Attribut steht. Die Koordinate benötigt eine Skalierung, damit nicht die Ränder der Positionstextur ausgelesen werden. Mit dieser Information wird die Position aus der Positions-Textur ermittelt und anschließend diese der Bounding Box der Ursprungsdaten auf die Ausgangsgröße skaliert.

7.2 Komprimierung mit dem JPEG und PNG Format

Die Bilder können entweder mit dem JPEG oder PNG Format komprimiert werden. Andere Bildformate, wie zum Beispiel JPEG 2000, wären zwar durchaus geeigneter, werden aber nicht ausreichend von den aktuellen Browsern unterstützt. Bei der Speicherung der Bilder mittels JPEG muss beachtet werden, dass keine verlustfreie Komprimierung möglich ist. Es fallen beim Betrachten der Bilder in der höchsten Qualitätsstufe von JPEG, gegenüber der verlustfreien Darstellung, zwar keine Unterschiede auf, allerdings unterscheiden sich die konkreten Farbwerte in beiden Darstellungen teilweise wesentlich. JPEG besitzt zwar einen verlustfreien Modus mit der Erweiterung JPEG-LS, dieser Modus wird jedoch von keinem Browser standardmäßig unterstützt. Somit entstehen selbst in der höchsten Qualitätsstufe Artefakte bei der Darstellung der Geometriedaten. Besonders beim Index-Bild führt dies zu teilweise völlig falschen Resultaten. Es ist bei diesem aber besonders wichtig, dass der genaue Index-Wert abgelegt ist, da kleinste Abstufungen ein völlig anderes Dreieck definieren.

Um die Artefakte der Komprimierung im JPEG Format weitestgehend zu beseitigen, muss das Bild um den Faktor 8 vergrößert werden. JPEG komprimiert das Bild in separaten 8×8 Blöcken, diese Blockbildung wird mit der Skalierung umgangen und die Daten bleiben größtenteils unverändert. Durch die Skalierung wird das Bild jedoch wesentlich größer, womit es gegenüber der verlustfreien Kompression von PNG unbrauchbar wird. Die Abbildung 7.1 zeigt die Artefakte, welche beim Speichern mit JPEG auftreten. Links oben wurden für den Index und die Attribute das JPEG Format mit 100% als Qualitätseinstellung in einer Auflösung von 256×256 Pixel verwendet. Die verlustbehaftete Kompression des Index-Bildes führt dazu, dass die falschen Dreiecke miteinander verbunden werden.

Das Modell rechts oben verwendet für die Attribute ebenfalls das JPEG Format mit 100% als Qualitätseinstellung in einer Auflösung von 256×256 Pixel. Allerdings wird für das Index-Bild das PNG Format in der Auflösung von 256×256 Pixel verwendet und damit ein wesentlich besseres Resultat erzielt. Um die Artefakte weiter zu verringern wird die Auflösung der JPEG Bilder im Modell links unten um den Faktor 8 vergrößert. So entsteht ein noch besseres Resultat, allerdings benötigen die JPEG Bilder nun mehr Speicherplatz als im PNG Format. Links unten wird letztendlich für den Index und die Attribute das PNG Format verwendet. Das erzielt ein Ergebnis, welches fast nicht vom Ursprungsmodell zu unterscheiden ist.

Für eine effiziente Komprimierung der Bilder müssen möglichst viele ähnliche Pixel im Bild nebeneinander angeordnet sein, was besonders dem Komprimierungsalgorithmus von PNG entgegen kommt. Dieser wendet eine Delta-Kodierung zwischen den einzelnen Pixeln an und komprimiert das gesamte Bild anschließend mit dem Deflate-Algorithmus. Die Abbildung 7.2 zeigt eine vergrößerte Ansicht des Index- und Positions-Bildes. Falls die Indexliste für eine gute Vertex-Cache-Ausnutzung optimiert wurde, befinden sich ähnliche Pixel nebeneinander und können daher wesentlich besser komprimiert werden. Die Abbildung 4.1 zeigt den Unterschied zwischen einem unoptimierten und optimierten Index-Bild.

Wenn das Hinterhofszene Modell mit jeweils vier 256×256 Bildern für die Position, Normale, Texturkoordinate und den Index im PNG Format nach dem eingangs erläuterten Prinzip abgespeichert wird, beträgt die Dateigröße insgesamt 38 kB. Dies entspricht einer Einsparung um den Faktor 1 zu 2 gegenüber binären GZIP.

7.3 Progressive Vorschau des Modells über Streaming

Um eine progressive Vorschau des Modells zu ermöglichen, können die Level of Detail Stufen in einzelne Bildersätze kodiert und übertragen werden. Dies würde bei vier Stufen allerdings die benötigte Bandbreite vervierfachen. Das JPEG und PNG Format bietet zur Übertragung der Bilder einen progressiven Modus an, in dem diese in mehreren Schritten progressiv aufgebaut werden. Es erlaubt, im Gegensatz zum normalen zeilenweisen Aufbau des Bildes, eine sehr frühe Vorschau. Jeder Dekodierungsschritt, welcher eine progressive Zwischenstufe anzeigt, benötigt in etwa soviel Zeit zum Dekodieren wie der eines einzelnen nicht progressiven Bildes.

Bei JPEG wird das Bild dabei in mehreren Schritten übertragen. Der erste entspricht dem einer Speicherung in der geringsten Qualitätsstufe von JPEG. Jeder Schritt fügt dem Bild weitere Informationen hinzu und verfeinert dieses so schrittweise. Die Abbildung 7.3 zeigt auf der linken Seite, wie stückweise ein Image Geometry Bild im JPEG Format aufgebaut wird, welches die Position eines Modells enthält. Dabei enthalten alle Stufen im Wesentlichen Informationen des gesamten Bildes, allerdings sind diese in den ersten Stufen stark miteinander vermischt.

Der progressive Aufbau von PNG, welcher auf der rechten Seite der Abbildung 7.3 zu sehen ist, nutzt dagegen den Adam7-Algorithmus [Lav00]. Dieser unterteilt das Bild in sieben Stufen. In jeder dieser Stufen werden dem Bild nach dem Muster in Listing 7.2 Pixel hinzugefügt. Das Muster wird dabei in 8×8 Blöcken auf das Bild angewendet. Die Zahl in dem Muster gibt dabei die Stufe an, in welcher der Pixel an dieser Stelle hinzugefügt wird. Da in der ersten Stufe nur jeweils ein Pixel pro 8×8 Block übertragen

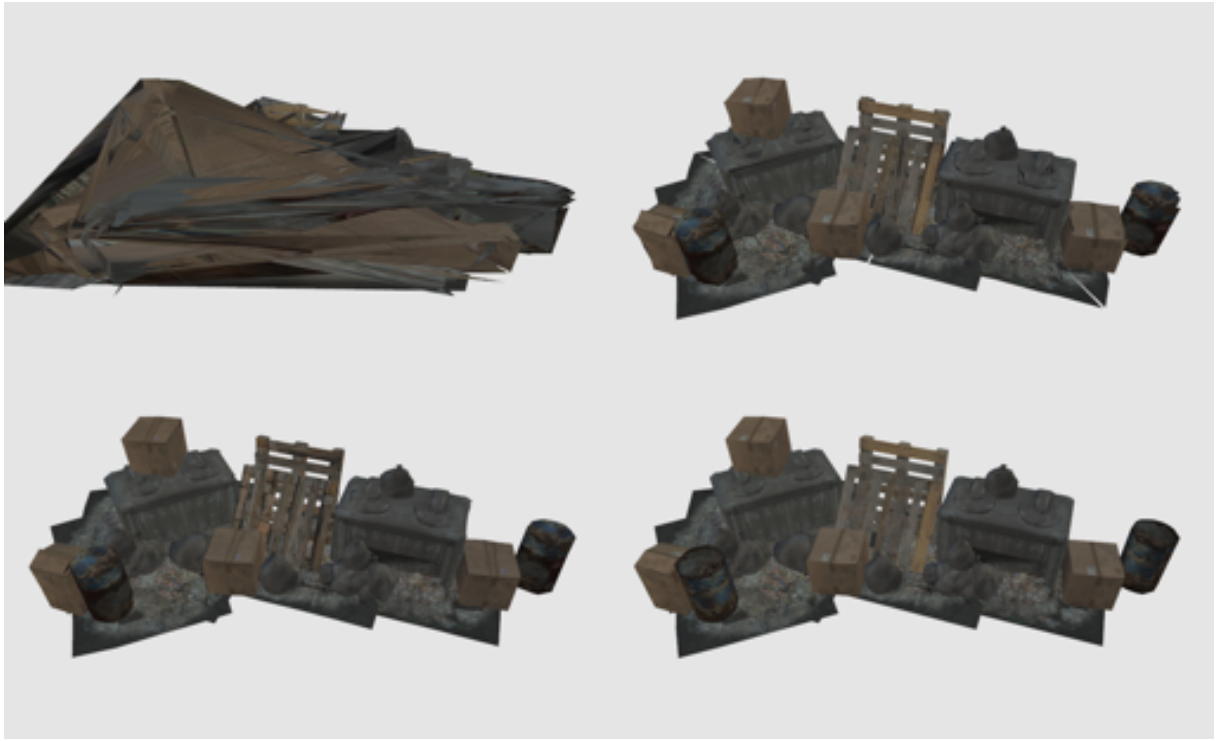

Abbildung 7.1: Links oben wurden für den Index und die Attribute das JPEG Format mit 100% als Qualitätseinstellung in einer Auflösung von 256×256 Pixel verwendet. Die verlustbehaftete Kompression des Index-Bildes führt dazu, dass die falschen Dreiecke miteinander verbunden werden.
Das Modell rechts oben verwendet für die Attribute ebenfalls das JPEG Format mit 100% als Qualitätseinstellung in einer Auflösung von 256×256 Pixel. Allerdings für den Index das PNG Format in der Auflösung von 256×256 Pixel. Damit wird ein wesentlich besseres Resultat erzielt.
Um die Artefakte weiter zu verringern, wird im Modell links unten die Auflösung der JPEG Bilder um den Faktor 8 vergrößert. So entsteht ein noch besseres Resultat, allerdings benötigen die JPEG Bilder nun mehr Speicherplatz als im PNG Format.
Links unten wird letztendlich für den Index und die Attribute das PNG Format mit einer Auflösung von 256×256 Pixel verwendet. Damit wird ein gutes Ergebnis erzielt, welches fast nicht vom Ursprungsmodell zu unterscheiden ist. ©Modell, Crytek GmbH

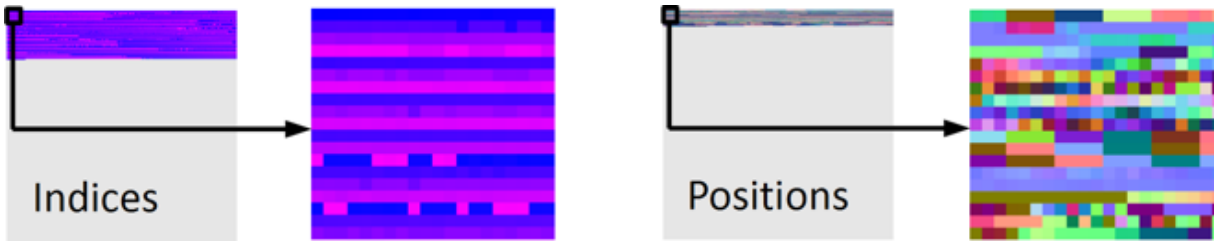

Abbildung 7.2: Nahaufnahme des Index- und Positions-Bildes gespeichert als Image Geometry. Da ähnliche Indices nach einer durchgeführten Vertex-Cache-Optimierung meist nebeneinander angeordnet sind, lässt sich dieses Bild sehr gut komprimieren. Die Positionen sind dagegen meist recht zufällig angeordnet und können somit nur wesentlich schlechter komprimiert werden.

wird, kann eine sehr schnelle Vorschau erzeugt werden. Die in den ersten sechs Stufen fehlenden Pixel werden interpoliert, es fließen somit in den ersten Stufen nicht alle Informationen des Bildes ein. Die Abbildung 7.4 zeigt wie dies an einem konkreten Beispiel ohne Interpolation aussieht.

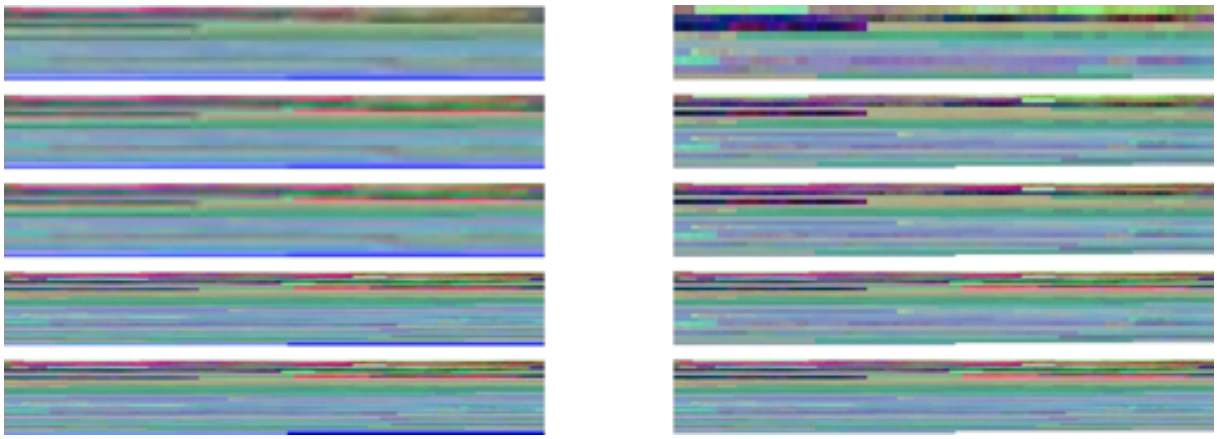

Abbildung 7.3: Vergleich der progressiven Transfermodi von JPEG und PNG am Beispiel eines Image Geometry Bildes, welches die Position enthält. Rechts ist der progressive Aufbau von JPEG zu sehen. Links der progressive Aufbau mit Adam7 des PNG Formates.

```
1 6 4 6 2 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
3 6 4 6 3 6 4 6
7 7 7 7 7 7 7 7
5 6 5 6 5 6 5 6
7 7 7 7 7 7 7 7
```

Listing 7.2: Mit dem Adam7 Modus von PNG wird das Bild in sieben Schritten aufgebaut. Es wird für jeden 8×8 Block das abgebildete Muster verwendet. Die Zahlen geben dabei den Schritt an, in welchem der Pixel zu dem Block eingefügt wird.

Abbildung 7.4: Die Pixel werden für jeden Schritt nach dem Muster aus Listing 7.2 eingefügt. Bei der Darstellung der Stufe wird zwischen diesen Pixelwerten interpoliert, anstatt lediglich die eingefügten Pixel darzustellen.

Die progressiven Modi der Bildformate können für die Übertragung der Image Geometry Daten verwendet werden. Damit muss nur jeweils das höchste Level of Detail des Modells übertragen werden und trotzdem erhält der Nutzer eine Vorschau auf dieses. In der vorliegenden Arbeit wurde das prototypisch anhand des Modells von Lee Perry Smith [PS12], der Hinterhofszene aus dem CryENGINE 3 Free SDK und dem vereinfachten Modell des Goldenen Reiters durchgeführt. Dabei wurden die Index-Informationen und Texturen vorab komplett übermittelt. Die vollständigen Index-Informationen werden vorher benötigt, da die Attribute in den progressiven Zwischenstufen über den gesamten Bereich vorliegen. Das Index-Bild kann jedoch, bei durchgeführter Vertex-Cache-Optimierung, im Verhältnis zu den Attribut-Bildern

gut komprimiert werden. Falls sogar nur ein aufsteigender Index ohne Wiederverwendung der Attribute genutzt wird, beträgt die Größe des Index Bildes im PNG Format lediglich 1 kB.

Als erstes soll die Übertragung mit dem progressiven Modus von JPEG betrachtet werden. Die Abbildung 7.5 zeigt von rechts oben bis links unten die Zwischenstufen 36 kB, 42 kB, 58 kB, 82 kB und 172 kB für das Modell von Lee Perry Smith, die bei der progressiven Übertragung mit dem JPEG Format entstehen. Die Attribut-Bilder verwenden das JPEG Format und wurden um den Faktor 8 skaliert, damit die Artefakte nicht zu stark ausfallen. So wird jedoch auch mehr Zeit benötigt, bis die erste Stufe dargestellt werden kann. Die Zwischenstufen lassen das Modell bereits sehr gut erahnen, allerdings benötigt die erste Stufe schon 36 kB.

Abbildung 7.5: Progressive Streaming mit dem progressiven Modus des JPEG Formates am Beispiel des Modells von Lee Perry Smith von rechts oben bis links unten mit den Zwischenstufen 36 kB, 42 kB, 58 kB, 82 kB und 172 kB. Die Attribut-Bilder verwenden das JPEG Format und wurden um den Faktor 8 skaliert, damit die Artefakte nicht zu stark ausfallen. Es wird so jedoch auch mehr Zeit benötigt, bis die erste Stufe dargestellt werden kann, welche allerdings das Modell bereits sehr gut erahnen lassen. ©Modell, Lee Perry Smith

Das PNG Format verbraucht wesentlich weniger Speicherplatz, da durch die verlustfreie Komprimierung bereits Bilder in der Auflösung von 256×256 Pixeln ausreichen. In Abbildung 7.6 wurde der Adam7 Modus von PNG genutzt, um das Modell zu übertragen. In dieser sind die Zwischenstufen für die Datengrößen 14 kB, 18 kB, 24 kB, 28 kB, 32 kB und 74 kB abgebildet. In den ersten Stufen ist das Modell nur schwer zu erkennen, ab ungefähr 28 kB ist die Geometrie des Modells weitestgehend übertragen.

Die nachfolgenden Stufen übermitteln die fehlenden Normalen und Texturkoordinaten. Zwar zeigen die ersten Stufen eine visuell schlechtere Vorschau als das JPEG Format, dafür benötigen die PNG Bilder wesentlich weniger Speicherplatz und können somit schneller übertragen werden. Zum Vergleich benötigt die fünfte Stufe des PNG Formates in Abbildung 7.6 ungefähr so viel Zeit, wie die erste Stufe des JPEG Formates in Abbildung 7.5.

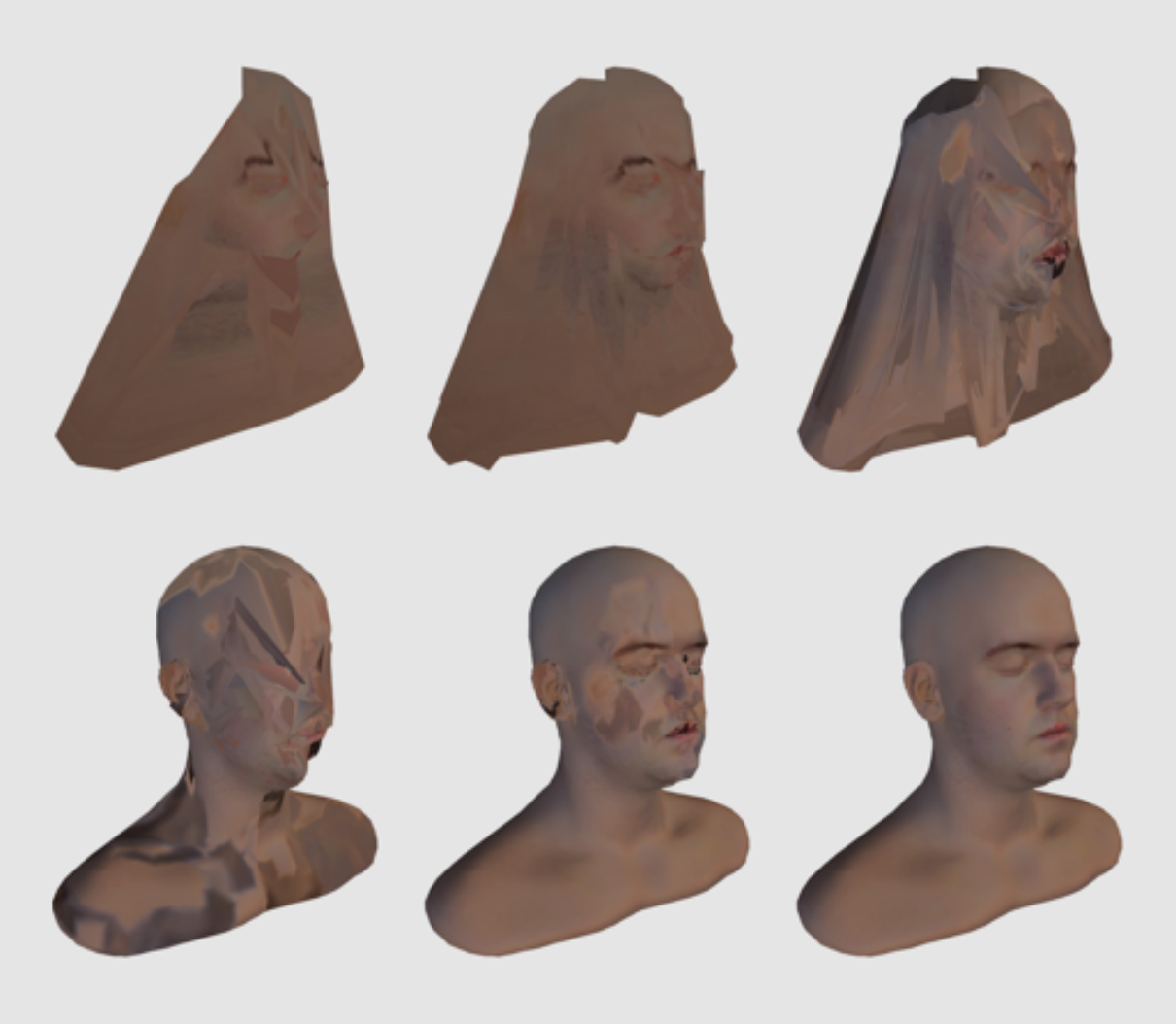

Abbildung 7.6: Progressive Streaming mit dem Adam7 Modus des PNG Formates am Beispiel des Modells von Lee Perry Smith von rechts oben bis links unten mit den Zwischenstufen 14 kB, 18 kB, 24 kB, 28 kB, 32 kB und 74 kB. In den ersten Stufen ist das Modell nur schwer zu erkennen. Ab ungefähr 28 kB ist die Geometrie des Modells weitestgehend übertragen und es werden lediglich die Normalen und Texturkoordinaten weiter übertragen. ©Modell, Lee Perry Smith

Bei einem geschlossenen Modell, in welchem die Dreiecke räumlich geordnet vorliegen, wie dies beim Modell von Lee Perry Smith der Fall ist, kann mit den progressiven Modi von PNG und JPEG eine recht gute Vorschau erzielt werden. Als Beispiel für ein komplexeres Modell soll das Modell der Hinterhofszene dienen, die aus mehreren getrennten kleineren Modellen besteht. In der Abbildung 7.7 sind auf der linken Seite die Zwischenstufen 48 kB, 62 kB, 70 kB, 110 kB und 154 kB für die Übertragung mit JPEG dargestellt. Auch hier sind die Attribute um den Faktor 8 vergrößert wurden. Ab ungefähr 30% ist das Modell bereits zu erkennen und wird anschließend weiter definiert, jedoch wird erst im letztem Schritt die endgültige Form des Modells erreicht.

Auf der rechten Seite der Abbildung 7.7 befinden sich die Zwischenstufen 30 kB, 34 kB, 40 kB, 44 kB und 48 kB, welche bei der Übertragung mit dem Adam7 Modus von PNG entstehen. Durch das Hinzufügen von Informationen mit Adam7 wird das Modell schrittweise verbessert, allerdings ist dieses erst ab etwa 44 kB eindeutig zu erkennen. Vorher verdecken die falsch verbundenen Dreiecke weitergehend die Details des Modells.

Für größere Modelle, wie dem Goldenen Reiter, wird im progressiven Modus von Adam7 keine sehr gute Vorschau erzeugt. Die Abbildung 7.10 zeigt dabei die entstandenen Zwischenstufen. Da die Dreiecke im Modell nicht räumlich angeordnet sind, werden sie nach ihrer Indexlisten Reihenfolge verbunden. Die ungefähre Form des Modells ist allerdings zu erahnen, da alle Dreiecke in einer Hülle bleiben, die dieses umschließt.

Die Geometrieinformationen können auch ohne den progressiven Modus zeilenweise übertragen werden. Je nach Fortschritt des Ladevorganges wird mit dieser Variante, das Modell Dreieck für Dreieck aufgebaut. Die Abbildung 7.9 zeigt den Interlace-Modus von PNG mit den Zwischenstufen 14 kB, 26 kB, 30 kB, 42 kB, 58 kB und 72 kB für das Modell von Lee Perry Smith. Da die Bilder zeilenweise aufgebaut werden, bekommt das Modell stückweise Informationen hinzugefügt, bis es komplett sichtbar ist. Das Positions-Bild wird dabei als erstes übertragen, womit das Modell sehr früh zu erkennen ist. In den späteren Stufen werden lediglich die Normalen und Texturkoordinaten übermittelt.

Für den Goldenen Reiter erzeugt die Übertragung mit dem Interlace-Modus von PNG bessere Zwischenstufen als der Adam7 Modus. In Abbildung 7.10 sind die Zwischenstufen dieser Methode abgebildet. Durch die ungeordnete Anordnung der Indexliste werden in jeder Stufe, an einer scheinbar zufälligen Stelle, Dreiecke hinzugefügt. Die Form des Modells ist somit früh zu erkennen, jedoch beinhaltet es bis zur letzten Stufe Löcher. In der vorletzten abgebildeten Stufe sind die Positionen bereits vollständig übertragen, lediglich die Normalen sind noch nicht vollständig übermittelt.

7.4 Streaming Animations

Ein Satz von Image Geometry Bildern speichert jeweils ein komplettes Modell. Durch die hohe Dekodierungsgeschwindigkeit kann durch einfaches austauschen der Bilder eine Animation des Modells realisiert werden. Dies wurde von Briceno et al. [BH03] für eine Folge von Geometry Images untersucht. Mit der von Briceno et al. untersuchte Technik können allerdings lediglich Animationen für geschlossene Modelle realisiert werden. Mit der Verwendung von Image Geometry Bildern ist dagegen die Animation von allgemeineren Modellen möglich.

Um die Bilderserie effektiv zu komprimieren und zu übertragen, kann sie in ein Videoformat konvertiert werden. Mit HTML5 ist es möglich, diese Videos, ohne die Verwendung von zusätzlichen Erweiterungen, einzubinden und abzuspielen [Pil12]. Zum Zeitpunkt der Arbeit existiert jedoch kein Codec, welcher ein verlustfreies Abspeichern der PNG Bilder ermöglicht und gleichzeitig von HTML5 unterstützt wird. Mit den getesteten Codecs Theora und h264 führt selbst die höchste Qualitätsstufe zu starken Artefakten. Die Abbildung 7.11 zeigt das Ergebnis, sobald der Codec h264 mit den höchsten Qualitätseinstellungen verwendet wird. Deswegen kann die Videocodierung für diesen Anwendungsfall nicht verwendet werden. In dieser Arbeit werden daher lediglich PNG Bilder verwendet, welche nacheinander ausgetauscht werden und so ein verlustfreies Video simulieren. Damit entfällt jedoch die potentielle Komprimierung zwischen den einzelnen Bildern.

Um die Technik zu erproben, wurde in dieser Arbeit ein Exporter für *Blender* geschrieben, welcher für jeden Frame eines animierten Modells die transformierten Geometriedaten exportiert und einen Satz von Image Geometry Bildern abspeichert. Da sich die Indices und Texturkoordinaten bei einem einzelnen Modell nicht ändern, muss in der Anwendung für jeden Frame nur jeweils das Bild für die Positionen und Normalen ausgetauscht werden. Das wird in JavaScript mit einer Timer Funktion realisiert. In Abbildung 7.12 ist die Technik am Beispiel einer mit *Blender* exportierten Kleidersimulation dargestellt.

Abbildung 7.7: Progressive Streaming am Beispiel der Hinterhofszene aus dem CryENGINE 3 Free SDK. Die linke Reihe zeigt den Ladevorgang für den progressiven Modus von JPEG für die Zwischenstufen 48 kB, 62 kB, 70 kB, 110 kB und 154 kB. Dabei wurden die Attribut-Bilder um den Faktor 8 vergrößert. Das Modell ist bereits in den ersten Stufen zu erkennen, erhält allerdings erst spät seine finale Form. Die rechte Reihe zeigt den Adam7 Modus von PNG für die Zwischenstufen 30 kB, 34 kB, 40 kB, 44 kB und 48 kB. Hierbei baut sich das Modell schrittweise auf, wobei in den Zwischenstufen Vertices miteinander verbunden werden, welche keinen Zusammenhang besitzen. ©Modell, Crytek GmbH

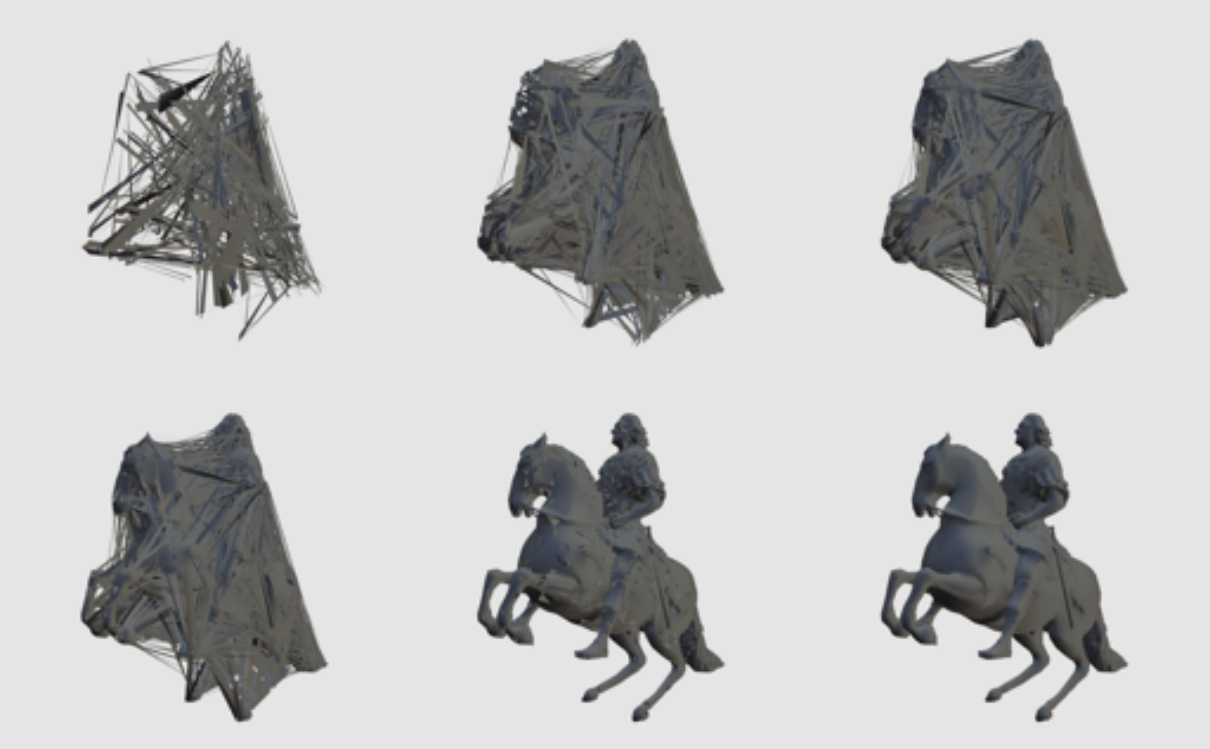

Abbildung 7.8: Progressive Streaming des Goldenen Reiters mit PNG Adam7. Ab ungefähr 80% ist das Modell erstmals ohne größere Fehler zu erkennen. In den vorhergehenden Stufen werden Vertices falsch verbunden. ©Modell, TU Dresden

Die Technik kann durchaus für normale Charakter Animationen genutzt werden. Jedoch eignen sich, für solche Animationen, Skelett basierte Techniken wesentlich besser, da dabei weniger Datenaufkommen anfällt und bessere Resultate erzielt werden können.

Jeder Frame der Animation benötigt eine eigene Bounding Box, andernfalls kommt es zu Verzerrungen. Diese Bounding Box Information muss deshalb für jeden Frame in einer JSON Datei hinterlegt werden. Alternativ kann die Information auch im *iTXt* oder *tEXt* Chunk des Index-Bildes gespeichert werden. Diese Chunks, welche im PNG Format bereitgestellt werden, bieten ein Key/Value System zum Ablegen von beliebigen TXT Daten. Ein weiterer Vorteil ist, dass alle benötigten Informationen in einem Bild und nicht getrennt vorliegen.

7.5 Bewertung

Die Technik bringt einige Vorteile, wie den progressiven Transfer, die schnelle Dekodierung und die geringe Dateigröße. Der Nachteil durch die starke Quantisierung kann durch weitere Maßnahmen, wie der Verwendung von zwei Bildern, welche jeweils 8 bit abspeichern, umgangen werden. Damit verdoppelt sich allerdings ebenfalls der benötigte Speicherplatz. Alternativ kann durch Ausnutzung des Alphakanals die Präzision verbessert werden.

Da in einem einzelnen Index-Bild nur jeweils 8 bit pro Koordinate gespeichert werden können, sind für die Attribute nur Texturen bis zur Größe 256×256 möglich. Damit können maximal 65.536 Attribute abgespeichert werden. Statt direkt die Pixel-Koordinate im Index-Bild zu speichern, können alle drei Farbkanäle genutzt werden, um insgesamt ein 24 bit Integer-Wert abzulegen. Dazu wird der Index in eine Bit-Zeichenfolge zerlegt und jeweils ein Byte in jedem Farbkanal abgespeichert. Die Kanäle werden im Shader anschließend ausgelesen und die Bitfolgen zum ursprünglichen Integer-Wert zusammengesetzt.

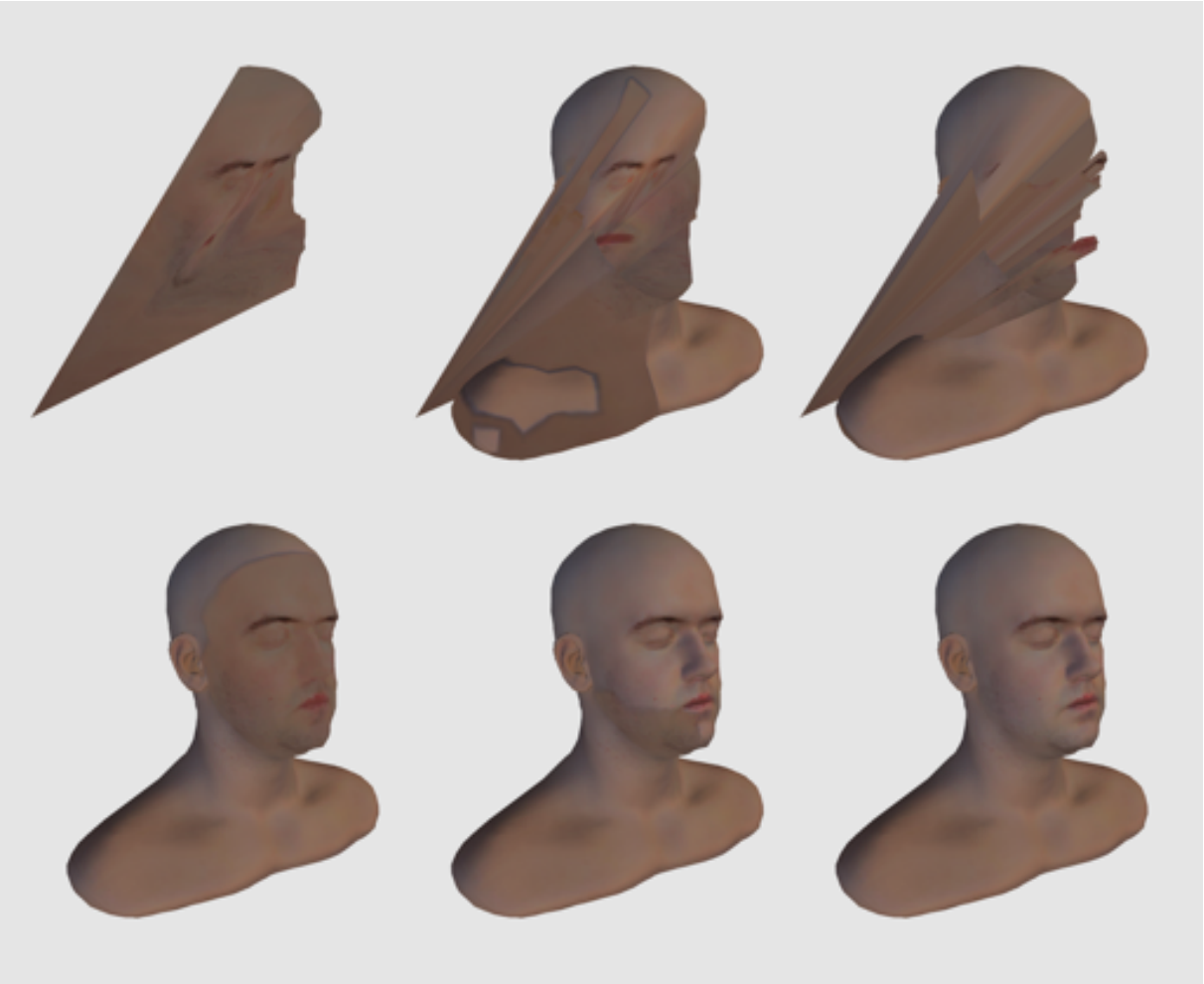

Abbildung 7.9: Progressive Streaming mit dem Interlace-Modus des PNG Formates am Beispiel des Modells von Lee Perry Smith von rechts oben bis links unten mit den Zwischenstufen 14 kB, 26 kB, 30 kB, 42 kB, 58 kB und 72 kB. Da die Bilder zeilenweise aufgebaut werden, wird dem Modell stückweise Informationen hinzugefügt, bis dieses komplett sichtbar ist. ©Modell, Lee Perry Smith

Abbildung 7.10: Progressive Streaming des Goldenen Reiters mit dem Interlace-Modus von PNG. Das Modell baut sich nacheinander auf. Da die Dreiecke zufällig im Modell verstreut sind, ist bereits früh die ungefähre Form des Modells zu erkennen. ©Modell, TU Dresden

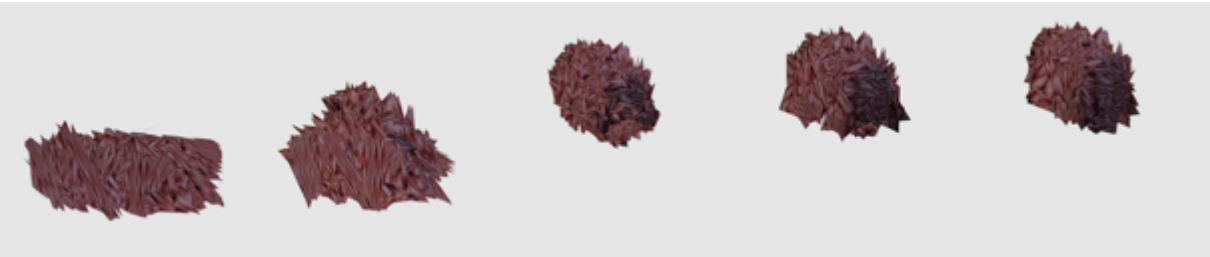

Abbildung 7.11: Durch das Streaming eines h264 Videos können die Positionen und Normalen eines Modells animiert werden. Jedoch reicht selbst die höchste Qualitätsstufe des h264 Codec nicht aus, um eine Darstellung ohne Artefakte zu erreichen.

Abbildung 7.12: Durch das Streaming von mehreren Bildern hintereinander können die Positionen und Normalen eines Modells animiert werden. Das Beispiel zeigt eine einfache Kleider Simulation, welche vorher in *Blender* erstellt wurde.

Dieser Wert kann zusammen mit der Größe der Attribut-Bilder genutzt werden, um direkt im Shader-Programm die Pixel-Koordinate zu ermitteln. Damit können potentiell Attribut-Bilder mit einer Auflösung von 4096×4096 adressiert werden. Für noch größere Modelle muss das Modell zerlegt und die Geometriedaten in mehreren Index- und Attribut-Bildern abgelegt werden. In dieser Arbeit wurden jedoch nur 256×256 Bilder genutzt. Größere Modelle müssen somit vorher entsprechend reduziert werden.

Die Quantisierung zu 8 bit pro Komponente resultiert in einigen Artefakten. Dies ist besonders in den Positionen und Texturen zu erkennen. Die Auswirkungen der Quantisierung der Position wurde bereits im Abschnitt 4.4 untersucht. Bei kleineren Modellen, welche viele Texturenübergänge enthalten, kommt es des Weiteren insbesondere an den Rändern zu Artefakten. Je nach Anwendungsfall kann dies dazu führen, dass die Technik nicht eingesetzt werden kann. In Abbildung 7.13 ist diese Quantisierung an einer Nahaufnahme eines Bootsmotoren Modells zu sehen. Darin sind deutliche Artefakte bei den Texturkoordinaten des abgebildeten Modells zu erkennen. Falls dieses Modell nicht aus der Nähe betrachtet werden soll, ist diese Quantisierung ausreichend. Für Detailansichten ist sie jedoch eher ungeeignet.

Abbildung 7.13: Artefakte bei der Quantisierung der Texturkoordinaten am Beispiel eines Motorenmodells aus dem CryENGINE 3 Free SDK. Links ist das Ursprungsmodell abgebildet, rechts die mit Image Geometry auf 8 bit quantisierte Version. Es sind deutliche Artefakte an den Rändern der Texturkoordinate zu erkennen. ©Modell, Crytek GmbH

Um die Präzision zu erhöhen, können statt einem, jeweils zwei Attribut-Bilder verwendet werden. Damit stehen pro Komponente 16 bit zur Verfügung, was für die meisten Anwendungsfälle ausreicht. Der benötigte Speicherplatz erhöht sich so jedoch ebenfalls um das Doppelte.

Alternativ kann zusätzlich der Alphakanal des Bildes genutzt werden, damit stehen insgesamt 32 bit statt 24 bit zur Verfügung. Um dies effektiver auszunutzen, wird pro Komponente jeweils ein Bild verwendet. Damit sind die einzelnen Komponenten der Attribute gemeinsam in einem Bild gespeichert. Im Rot- und Grünkanal wird die Position mit einer Präzision von 16 bit abgelegt. Dazu muss die vorerst quantisierte Integer-Variable in zwei Short-Variablen zerlegt werden. Dies ist im Codebeispiel 7.3 aufgeführt. Die führenden 8 bit der Integer-Variable, welche durch $\lfloor Wert/256 \rfloor$ ermittelt werden, sind im Rotkanal abgelegt. Die hintere Bitfolge wird durch $Wert \mod (256)$ errechnet und im Blaukanal abgespeichert.

```
1 Encode:     color.r = floor(pos.x/256);
2             color.g = modulo(pos.x,256);
3
4 Decode:     pos.x = (color.r*256 + color.g)
```

Listing 7.3: Zerlegt eine 16 bit Integer-Variable, welche in pos.x gespeichert ist, in zwei 8 bit Short-Variablen. Diese beiden Short-Variablen werden im Rot- und Grünkanal abgelegt. Im Shader können beide Short-Variablen wieder zusammengesetzt werden.

Im Blaukanal wird die X-Komponente der Normale abgespeichert. Der Alphakanal speichert letztendlich die U-Komponente der Texturkoordinate. Für Normale und Texturkoordinate werden also jeweils 8 bit verwendet. Analog wird dies für die Y- und Z-Komponenten durchgeführt.

Als Bildformat zur Speicherung der Attribute bietet sich lediglich PNG an. Mit JPEG müssen die Bilder um den Faktor 8 vergrößert werden, was wiederum zu einer wesentlich größeren Datengröße gegenüber PNG führt. Um während des Streaming eine progressive Vorschau des Modells zu ermöglichen, kann der Adam7- oder der Interlace-Modus von PNG genutzt werden. Der Adam7 Modus liefert dabei besonders für kleinere Modelle eine recht gute Vorschau des Modells. Um die progressive Darstellung mit Adam7 weiter zu verbessern, werden die Vertices so angeordnet, dass diese das Muster von Adam7 beachten. Damit können in den einzelnen Stufen von Adam7 einzelne Level of Detail Modelle verpackt werden. Dies bedeutet jedoch einen nicht unwesentlichen Implementierungsaufwand, da eine passende Anordnung der Dreiecke gefunden werden muss. Zusätzlich liegt die Vermutung nahe, dass nach dieser Anordnung die Kompressionsrate wesentlich schlechter ist. Für Modelle, in welchen die Dreiecke ungeordnet abgespeichert wurden, bietet sich die Übertragung mit dem Interlace-Modus von PNG an. Damit ist zumindest eine einfache Vorschau der bereits übertragenen Daten möglich.

Die Dekodierungszeit der Bilder ist unabhängig von JavaScript und lässt sich dementsprechend schnell durchführen. Die Auswertung in Abschnitt 8.2 hat gezeigt, dass selbst mit ausgeschalteten Just-In-Time Compiler die selbe Dekodierungszeit benötigt wird. In der hier aufgeführten Implementation können lediglich Bilder im Zweierpotenzverhältnis verwendet werden und sind damit oft größer als für die Geometriedaten nötig. Da größere Bilder längere Zeit für die Dekodierung benötigen, entsteht hierbei ein Overhead, welcher durch eine bessere Implementation vermieden werden kann.

Verglichen mit den anderen Kompressionsformaten bietet das Image Geometry Format einen guten Kompressionsfaktor bei gleichzeitig geringer Dekodierungszeit. Das Format ist jedoch durch die hohe Quantisierung nicht ohne weitere Bearbeitung für alle Anwendungsfälle geeignet, besitzt allerdings durchaus Potenzial.

8 Download- und Dekodierungsgeschwindigkeit

Nicht allein der erzielte Kompressionsgrad ist entscheidend für die Bewertung einer Technik, es müssen ihm darüber hinaus die benötigte Dekomprimierungszeit gegenübergestellt werden. Bei schnellen Verbindungsgeschwindigkeiten kann diese schnell einen entscheidenden Einfluss auf die Gesamtgeschwindigkeit der Anwendung ausüben. Dieser Abschnitt ermittelt den Einfluss, indem für alle Modelle die Dateigröße und Dekodierungszeit mit den verschiedenen Techniken gemessen wurde. Bei den hier aufgeführten Messungen ist zu beachten, dass die Techniken nur bedingt vergleichbar sind. So bietet das Image Geometry Format in der hier verwendeten Implementation lediglich eine Präzision von 8 bit. Der MG2 Modus des OpenCTM Formates unterstützt dagegen keine Modelle mit mehreren Materialien.

Es werden alle Techniken neben der Hinterhofszene und dem hochaufgelösten Modell des Goldenen Reiters (870.069 Dreiecke) auf weitere fünf Modelle aus dem CryENGINE 3 Free SDK angewandt, welche in Abbildung 8.1 abgebildet sind. Dabei ist anzumerken, dass bei Modellen mit unterschiedlicher Topologie andere Resultate auftreten. So kann zum Beispiel eine hochaufgelöste planare Fläche wesentlich besser komprimiert werden, als ein gängiges 3D-Modell.

Alle Tests wurden auf einem Intel Core 2 Duo mit 3,17 GHz mit dem Browser Chrome 21 durchgeführt, wobei diese je 100 mal ausgeführt und anschließend der Median ermittelt wurde. Als Orientierung für gängige Verbindungsgeschwindigkeiten sollen die Maximalgeschwindigkeit der folgenden Standards dienen: GPRS (56 kbit/s), EDGE (216 kbit/s), UMTS (384 kbit/s), DSL1000 (1 Mbit/s), HSDPA (7,2 Mbit/s), DSL16000 (16 Mbit/s) und DSL50000 (50 Mbit/s).

8.1 Kompressionsverhältnis

Dieser Abschnitt vergleicht die Dateigrößen der einzelnen Modelle, nachdem diese mit den Kompressionsformaten entsprechend gepackt wurden. Die Daten im JSON, BINÄR und WebGL Loader (UTF-8) Format sind zusätzlich mit GZIP komprimiert. Für die Image Geometry Attribut- und Index-Bilder wurde das PNG Format genutzt, welche vorher mit Hilfe der Bildbearbeitungssoftware GIMP in der höchsten Kompressionsstufe abgespeichert wurden. Für Modelle mit weniger als 65k Vertices und 65k Indices wurden PNG Bilder in der Auflösung von 256×256 Pixeln verwendet. Aufgrund der Implementation der Image Geometry Technik in dieser Arbeit, muss das Bild eine Zweierpotenz Größe besitzen. Da das Modell des Traktors mehr als 65k Vertices nutzt, benötigt es Bilder in der Auflösung von 512×512 Pixeln.

Beim Modell des Goldenen Reiters sind sogar Bilder in der Auflösung von 2048×2048 Pixel nötig. Damit ist das Bild oft größer, als es für die Modelldaten erforderlich ist, womit auch die Dateigröße und Dekodierungszeit der PNG Bilder ansteigt. Dies ist ebenfalls beim nachfolgenden Vergleich zu beachten. Die Tabelle 8.1 zeigt die Dateigrößen der Modelle in den verschiedenen Kompressionsformaten. Das Image Geometry Format bietet dabei die im Schnitt kleinsten Dateigrößen. Dies resultiert jedoch primär aus der starken Quantisierung auf 8 bit. Weniger stark quantisieren die Formate WebGL Loader und der MG2 Modus von OpenCTM. Beide bieten dennoch vergleichbar gute Ergebnisse und durchschnittlich ein Kompressionsverhältnis von 50% gegenüber den mit GZIP komprimierten binären Daten.

Abbildung 8.1: Die Abbildung zeigt von links oben bis rechts unten die im Abschnitt 8 verwendeten 3D-Modelle aus dem CryENGINE 3 Free SDK. Die Vertex-Anzahl der einzelnen Modelle beträgt aufsteigend: 1.218, 3.391, 6.997, 12.444, 22.367 und 42.852 Vertices. ©Modelle, Crytek GmbH

Dateigrößen in Byte						
	JSON GZIP	BIN GZIP	UTF-8 GZIP	CTM MG1	CTM MG2	I. G.
Hinterhof	190.319	89.556	45.254	92.754	37.779	**36.184**
Goldener Reiter	21.806.483	16.239.655	6.206.623	12.222.787	**3.557.172**	4.565.780
Wasserpumpe	50.045	22.385	10.854	23.382	**9.177**	9.877
Außenbootmotor	172.104	77.923	30.870	73.629	27.591	**27.204**
Fischer Haus	311.187	130.347	55.707	129.862	40.904	**37.512**
Wald Ruine	715.794	315.004	129.790	276.940	109.776	**84.869**
Leuchtturm innen	730.439	334.540	93.559	278.375	150.021	**76.073**
Traktor	1.691.981	771.327	336.671	755.841	323.402	**308.428**

Tabelle 8.1: Die resultierenden Dateigrößen für die verschiedenen Techniken. Die JSON, BINARY und UTF-8 Daten wurden mit GZIP komprimiert. Die Formate WebGL Loader (UTF-8), OpenCTM MG2 und Image Geometry bieten das beste Kompressionsverhältnis, wobei zu beachten ist, dass in der Implementation vom Image Geometry lediglich eine Präzision von 8 bit unterstützt wird. Nur beim Modell des Goldenen Reiters ergibt das MG2 Format ein wesentlich besseres Ergebnis.

8.2 Dekodierungsgeschwindigkeit

Die Tests zeigten, dass für eine Verbindungsgeschwindigkeit langsamer als HSDPA die Dateigröße entscheidend für die Gesamtzeit bis zur Darstellung des Modells ist. Die Dekodierungszeit, für die getesteten Formate, ist bei dieser Verbindungsgeschwindigkeit im Vergleich zur Downloadzeit zu klein, um sich entscheidend auf die Gesamtzeit auszuwirken. Ab einer Verbindungsgeschwindigkeit von ungefähr 4 Mbit spielt die Dekodierungszeit jedoch eine wichtige Rolle.

Diese wurde ermittelt, indem die Zeit gemessen wurde, welche ab dem Moment des Bereitstehen der Daten (Downloadende) bis zu deren Konvertierung zu einem 32 bit Float Array für die Attribute beziehungsweise 32 bit Integer Array für die Indices vergangen ist. Im Idealfall sollte diese Dekodierungszeit sehr klein sein. Als Referenz dient hierbei das Laden von Daten im Binärformat. Sie können über einen XHR Request direkt als 32 bit Float oder 32 bit Integer Array geladen werden und benötigen somit keine zusätzliche Dekodierungszeit. Ist die Verbindungsgeschwindigkeit sehr schnell und die Dekodierungszeit des Kompressionsformates sehr hoch, führt dies dazu, dass die Daten im Binärformat insgesamt schneller dargestellt werden können. Für die Image Geometry Technik wurde die Zeit zum Erstellen der Textur ermittelt. Dies ist nur bedingt ein fairer Vergleich, da hier unter Umständen die Textur bereits auf die Grafikkarte geladen wird und daher ein zusätzlicher Overhead entsteht.

Die Tabelle 8.2 zeigt die Dekodierungszeiten für die getesteten Modelle und Kompressionstechniken. Die Standardabweichung der Werte ist in Tabelle B.1 aufgeführt. Für alle getesteten Modelle benötigt das Format WebGL Loader (UTF-8) die geringste Dekodierungszeit. Das Image Geometry Format benötigt abhängig von der Größe der PNG Bilder ungefähr die gleiche Zeit zum Dekodieren. Da hier nur Bilder in Zweierpotenzgrößen genutzt werden können, ist die Dekodierungszeit für alle Modelle mit weniger als 65k Indices, aufgrund der Verwendung von Bildern in der Auflösung von 256×256 Pixeln, annähernd gleich.

Da der JIT Compiler den ausgeführten JavaScript Code zur Laufzeit optimiert, kommt es zu Schwankungen in der Dekodierungszeit, sobald mehr als nur ein Modell heruntergeladen wird. Die Abbildung 8.2 zeigt die Zeiten für das Modell der Hinterhofszene. Es wurde 100 mal hintereinander dekodiert, was wiederum mit dem Laden von 100 gleichen Modellen gleichgestellt werden kann. Die Dekodierungszeiten in den ersten Durchläufen sind länger als die Nachfolgenden, was darauf deutet das dem JIT Compiler nicht

Dekodierungszeiten in ms für Chrome 21					
	JSON GZIP	UTF-8 GZIP	CTM MG1	CTM MG2	Image Geometry
Hinterhof	23	**1**	54	36	2
Goldener Reiter	2.467	**161**	6.861	5.493	193
Wasserpumpe	6	**1**	16	10	8
Außenbootmotor	17	**1**	10	29	5
Fischerhaus	34	**1**	73	42	7
Waldruine	63	**1**	150	93	8
Leuchtturm innen	113	**3**	168	109	5
Traktor	214	**9**	457	276	31

Tabelle 8.2: Überblick der einzelnen Dekodierungszeiten unter Chrome 21. Das Format WebGL Loader (UTF-8) benötigt weniger Zeit als die restlichen Formate. Die Dekodierungszeit des Image Geometry Formates lässt sich jedoch weiter verbessern, indem statt 256×256 Bilder, an die Größe angepasste Bilder verwendet werden.

genug Zeit bereitgestellt wurde, um eine Optimierung des Codes durchzuführen. Des Weiteren zeigt der Graph einige Ausschläge, welche implizieren, dass der Garbage Collector periodisch aufgerufen wurden. Insgesamt bleiben die Zeiten jedoch auf dem selben Niveau, womit sich das Laden von mehreren kleinen Modellen linear verhält.

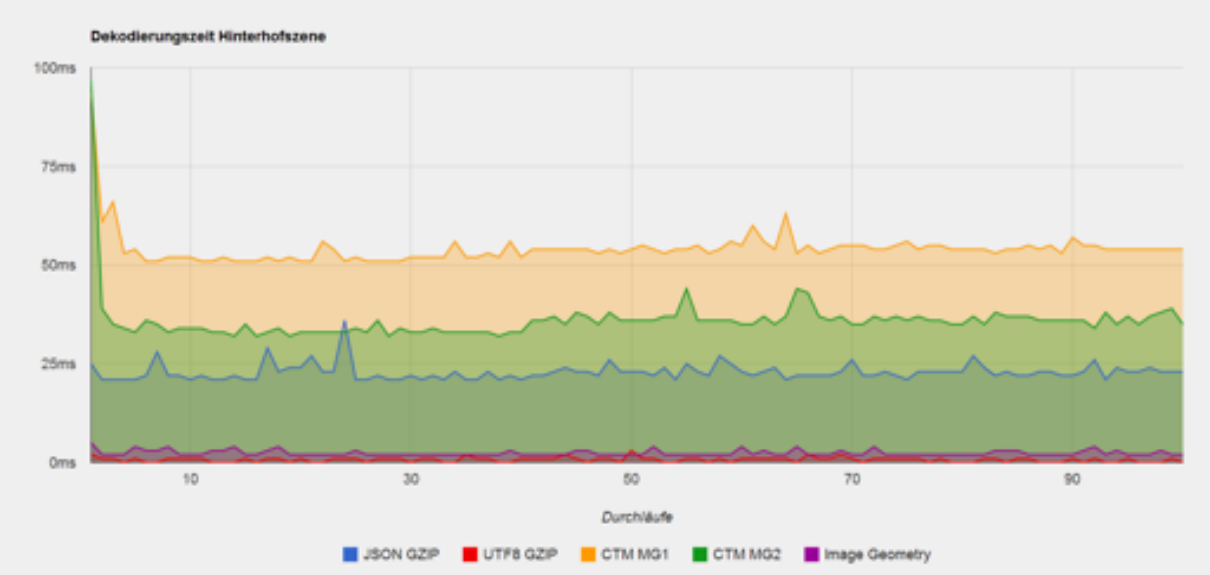

Abbildung 8.2: Dekodierungszeit der verschiedenen Techniken gemessen für 100 Durchläufe in Chrome 21 am Beispiel der Hinterhofszene. Die Zeiten in den ersten Durchläufen sind länger als die Nachfolgenden, was darauf deutet das der JIT Compiler nicht genug Zeit hatte, um eine Optimierung des Codes durchzuführen. Des Weiteren zeigt der Graph einige Ausschläge, welche implizieren, dass der Garbage Collector periodisch aufgerufen wurden.

Für größere Modelle fällt der Geschwindigkeitsanstieg durch den optimierten Code des JIT bei mehreren Durchläufen nicht mehr so deutlich auf. Die Abbildung 8.3 zeigt dies am Beispiel des Modells des Goldenen Reiters, welches ebenfalls 100 mal hintereinander dekodiert wurde. Für das WebGL Loader Format fällt auf, dass ab ungefähr 25 Durchläufen die Dekodierungszeit auf das Doppelte ansteigt. Der Code des Prototypen allokiert für jeden Durchlauf neuen Speicher. Bereits nach einigen Durchgängen ist

der Hauptspeicher dadurch vollständig belegt und der Garbage Collector muss für jeden neuen Durchlauf Speicher bereitstellen. Die unzureichende Speicherverwaltung liegt an der prototypischen Implementation der Dekodierung, da sich das WebGL Loader Format noch in Entwicklung befindet.

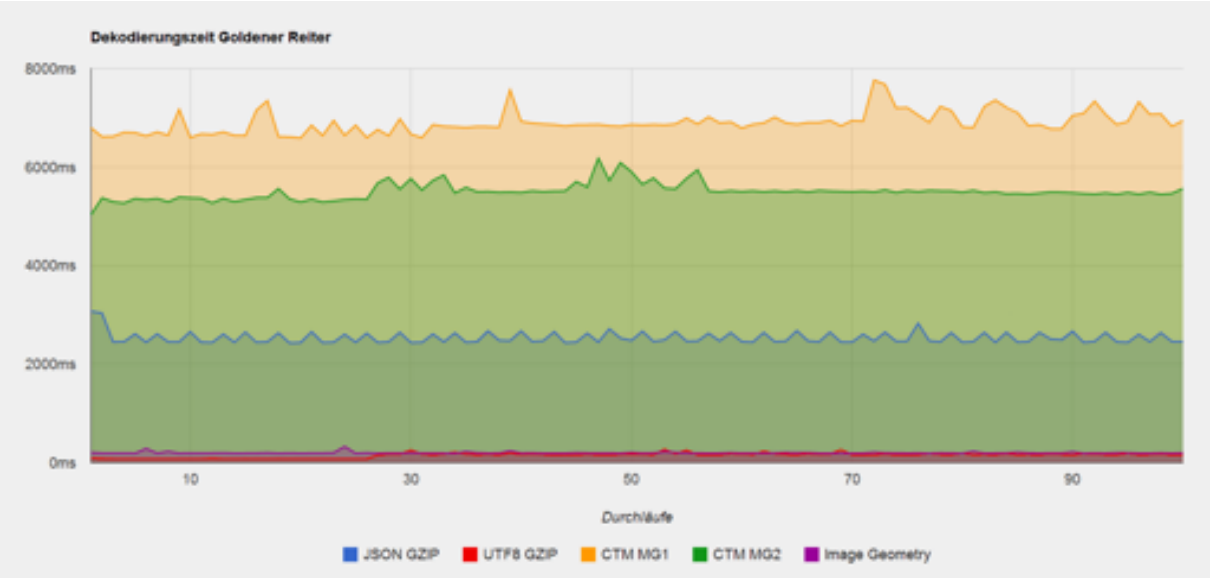

Abbildung 8.3: Dekodierungszeit der verschiedenen Techniken gemessen für 100 Durchläufe in Chrome 21 am Beispiel des Goldenen Reiters. Das Modell ist wesentlich größer, womit der JIT bereits bei den ersten Durchläufen ausgeführt wird. Das WebGL Loader Format benötigt ab dem 25. Durchlauf ungefähr die doppelte Dekodierungszeit.

8.3 Gesamtzeit zur Anzeige der Modelle

Die Zeit für den Download und der Dekodierung lässt sich ins Verhältnis zur Verbindungsgeschwindigkeit setzen. Damit wird ermittelt, bis zu welcher Geschwindigkeit sich der Einsatz der Kompressionsformate lohnt. Die Abbildung 8.4 und 8.5 zeigen den Graphen für die Modelle der Hinterhofszene und des Goldenen Reiters. Diese Graphen sind nur aussagekräftig, sobald die benötigte Zeit bis zur Anzeige des Modells im Vordergrund steht. Ist dagegen die maximale Bandbreite ein wichtigerer Faktor, sollte das Format mit der jeweils besten Kompressionsrate ausgewählt werden. Aus den Graphen wird deutlich, dass sich ab einer Verbindungsgeschwindigkeit von ungefähr DSL14000 (Hinterhofszene) und DSL24000 (Goldener Reiter) der Einsatz des MG2 Modus von OpenCTM nicht mehr lohnt. Die benötigte Dekodierungszeit sorgt in dem Fall dafür, dass die Datei schneller dargestellt werden kann, wenn diese lediglich mit dem GZIP komprimierten Binärformat heruntergeladen wird. Die Formate WebGL Loader (UTF-8) und Image Geometry lassen sich schneller dekodieren, womit diese auch bei hohen Geschwindigkeiten eine insgesamt schnellere Darstellung ermöglichen.

Es muss also vorher abgewogen werden, ob und welches Kompressionsformat verwendet werden soll. Eine Möglichkeit wäre, die Verbindungsgeschwindigkeit des Nutzers im Vorfeld zu ermitteln und diesem entsprechend das bestmögliche Format bereitzustellen. Für sehr schnelle Verbindungsgeschwindigkeiten würde somit lediglich das mit GZIP komprimierte Binärformat übertragen und langsameren Clients wird dagegen das OpenCTM MG2 Format angeboten.

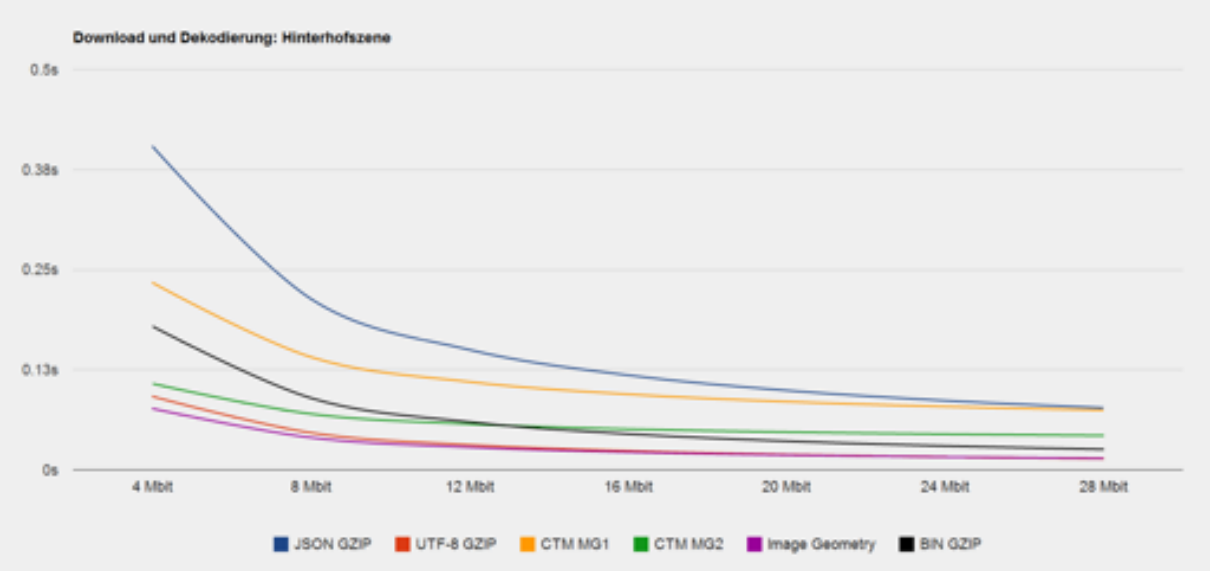

Abbildung 8.4: Vergleich der verschiedenen Techniken für Download und Dekodierung der Hinterhofszene. Die Formate WebGL Loader (UTF-8) und Image Geometry benötigen am wenigsten Zeit, wobei das WebGL Loader Format ab einer Geschwindigkeit von ungefähr 16 Mbit/s insgesamt schneller ist.

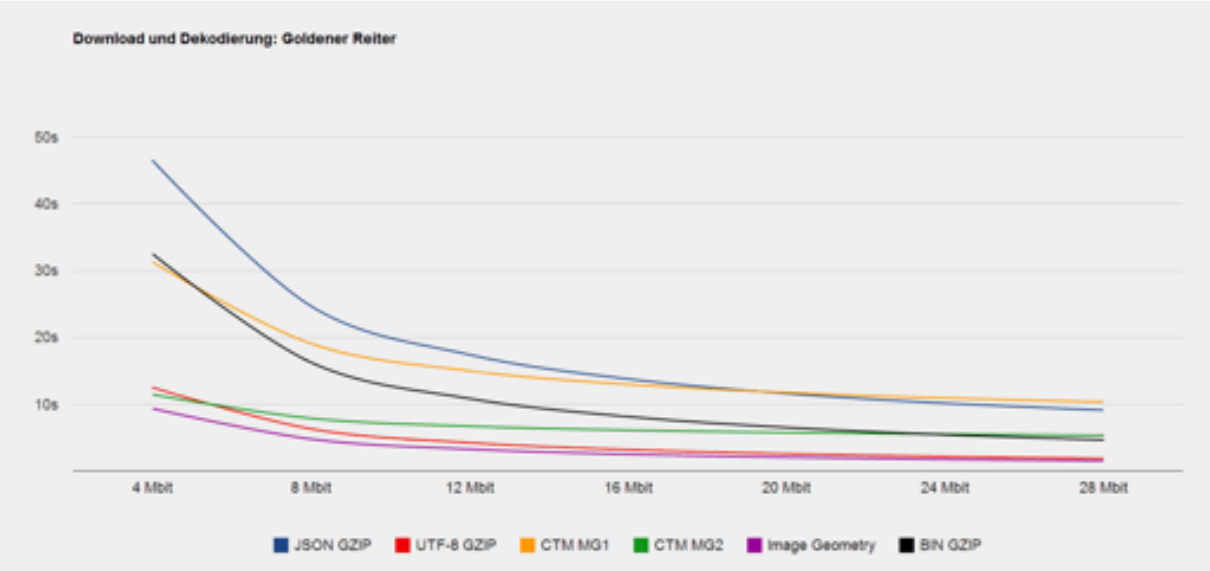

Abbildung 8.5: Vergleich der verschiedenen Techniken für Download und Dekodierung des Goldenen Reiters. Das schnellste Ergebnis liefert das Image Geometry Format, wobei in der Implementation lediglich eine Präzision von 8 bit unterstützt wird.

9 Texturen

Neben den Geometriedaten bestehen 3D-Modelle häufig aus einer oder mehreren Texturen. Diese Texturen liegen meist in bereits komprimierten Formaten vor. So besitzt zum Beispiel die Hinterhofszene aus Abbildung 3.1 insgesamt sechs Materialien, welche gemeinsam 395 kB benötigen. Alle Texturen sind bereits mit dem JPEG Format komprimiert. Da die Geometriedaten mit dem WebGL Loader Format auf ungefähr 45 kB reduziert werden können, machen nicht diese sondern die Texturen den Hauptteil des Datenaufkommens aus. Bildformate wie WebP [Goo12c] bieten für bestimmte Anwendungsfälle eine bessere Komprimierungsrate als JPEG, werden jedoch nicht von allen Browsern unterstützt. Für die Komprimierung der Texturen kommen lediglich die Formate JPEG und PNG in Frage. Das PNG Format erzielt jedoch für die meisten Texturen (sofern es sich um Fotomaterial handelt) eine wesentlich schlechtere Kompressionsrate als das JPEG Format.

Um dennoch die Dateigröße weiter zu verringern, kann die Qualität der Texturen verringert werden, indem eine kleinere Qualitätsstufe von JPEG ausgewählt wird. Des Weiteren kann die Auflösung der Texturen verkleinert werden. Bei einer Skalierung der Texturen auf 256×256 Pixel und einer Qualitätsstufe von 80% beträgt die Dateigröße der Texturen allerdings immer noch 130 kB. Damit sind die Texturen dennoch größer als die komprimierten Geometriedaten.

9.1 Textur-Streaming

Anstatt die Qualität der Texturen zu verringern, können sie auch schrittweise geladen (Streaming) werden. Dazu wird, wie schon beim Streaming der Image Geometry, der Progressive-Modus von JPEG verwendet. Um eine progressive Vorschau eines Bildes mit HTML zu ermöglichen, genügt es bereits ein Image-Element im HTML Code über den Tag *IMG* zu erstellen und als Quelle ein progressives JPEG Bild anzugeben. Dieses Element verwaltet anschließend selbstständig das Streaming und Anzeigen des progressiven Bildes.

In JavaScript kann ebenfalls ein solches Image-Element, welches anschließend ein progressives JPEG Bild lädt, dynamisch erzeugt werden. Allerdings ist es zum Stand der Arbeit nicht möglich, über ein *onprogress* Event oder ähnliches, die progressiven Zwischenstufen während des Transfers auszulesen und an eine Textur zu binden. Das Image-Element unterstützt lediglich das *onload* Event, welches erst ausgelöst wird, sobald das komplette Bild heruntergeladen wurde.

Um dennoch die progressiven Zwischenstufen anzuzeigen, muss das Bild über einen XMLHttpRequest (XHR) heruntergeladen werden. Der Request unterstützt in der Version von Juni 2012 mehrere Dateitypen, wie *arraybuffer* und *blob*, um binäre Daten herunterzuladen. Um eine Abwärtskompatibilität zu älteren Implementationen zu gewährleisten, werden die Daten allerdings über einen anderen Weg im Binärformat angefordert. Dazu wird nicht der *responseType* abgeändert, sondern der *Mime-Type* entsprechend mit dem String *text/plain; charset=x-user-defined* überschrieben. Dies weist den Server an, die Daten direkt als binären Stream zu senden. Über das *onprogress* Event des XHR können diese stückweise während der Übertragung ausgelesen werden.

Der binäre Stream kann nicht direkt in ein Bild dekodiert werden, da als Quelle des Image-Elements lediglich eine URL oder ein mit Base64 kodierter Byte-Stream angegeben werden kann. Daher wird der bisher übertragene Byte-Stream mit JavaScript in das Base64 Format kodiert. Diese Kodierung kann über einen Web Worker auf einen anderen Thread ausgelagert werden, um so die Ausführungsgeschwindigkeit insge-

samt zu verbessern. Anschließend kann der Base64 String als Quelle für das Image-Element angegeben werden, welches wiederum die progressive Zwischenstufe der bisher übertragenen Daten anzeigt. Dieses Bild kann anschließend direkt an eine Textur gebunden werden. Dies ist allerdings nur mit dem Browser Firefox möglich. Der Browser Chrome seit der Version 21 zeigt, statt der progressiven Zwischenstufen, lediglich eine schwarze Textur.

Um auch in Chrome 21 eine progressive Anzeige des JPEG Bildes über JavaScript zu ermöglichen, muss ein Umweg über das Canvas Element genommen werden. Dabei wird der Inhalt des Image-Elements, anstatt direkt in eine Textur, in einen separaten 2D Canvas gezeichnet. Dessen Inhalt kann anschließend für die Erstellung der Textur genutzt werden.

Sobald mit beiden Ansätzen jedoch Texturen, mit einer Auflösung größer als 1024×1024 Pixel, gebunden werden sollen, kommt es zu kleineren Einbrüchen der Framerate in der Anwendung. Die Übertragung einer solch großen Textur innerhalb eines Frames benötigt eine schnelle Grafikkarte. Je nach Geschwindigkeit der Grafikkarte können so unter Umständen die 60 FPS nicht mehr eingehalten werden. Die Textur muss somit in kleinere Teilstücke zerlegt werden. Jedes Teilstück kann anschließend Frame für Frame über *texSubImage2D* übertragen werden.

Dies löst das Problem jedoch nicht vollständig. Neben der Übertragungszeit der Textur zur Grafikkarte nimmt die Dekodierung des Bildes weitere Zeit in Anspruch. Die Dekodierung eines einzelnen 2048×2048 JPEG Bildes benötigt auf einem Intel Core 2 Duo (3.17 Ghz) ungefähr 50 ms. Es werden also ungefähr drei Frames bei einer Framerate von 60 FPS für die Dekodierung benötigt. Dieser Vorgang wird im Haupt-Thread der Anwendung ausgeführt und blockiert den gesamten Rendervorgang. Bei mehreren zu streamenden Texturen kommt es dabei zu großen Aussetzern.

Dieser Vorgang kann nicht ohne Weiteres in einem separaten Thread ausgeführt werden. Den Web Workern ist es nicht erlaubt, den DOM zu verändern, da sonst unvorhersehbare Ergebnisse auftreten könnten. Durch diese Einschränkung ist es ebenso nicht möglich, Elemente des DOMs innerhalb eines Web Workers zu verwenden. Das Image-Element verändert zwar den DOM nicht, jedoch ist es dennoch nicht möglich, dieses innerhalb eines Web Workers zu nutzen.

Damit die Dekodierung trotzdem in einem Web Worker stattfinden kann, muss die Dekodierung des JPEG Byte-Streams mit JavaScript durchgeführt werden. Es existieren bereits JavaScript-Bibliotheken, welche JPEG Bilder dekodieren können [DD11]. Die in dieser Arbeit untersuchten Bibliotheken unterstützten jedoch nicht den vollen Umfang von JPEG. So wurde insbesondere der Progressive-Modus von JPEG nicht unterstützt.

Um dennoch progressive JPEGs ohne ein Neuschreiben einer solchen Dekodierungsfunktion in JPEG zu unterstützen, kann das Werkzeug Emscripten [Zak12] verwendet werden. Mit diesem kann C oder C++ Code zu JavaScript überführt werden. Es nutzt dazu den Compiler clang [Cla12] um eine vorhandene C oder C++ Bibliothek zu LLVM-Bytecode (Low Level Virtual Machine) zu übersetzen, welcher anschließend zu JavaScript transformiert werden kann. In dieser Arbeit wurde mit diesem Programm die C++ Code Referenzimplementierung von JPEG [Ind12] nach JavaScript übersetzt. Dabei mussten jedoch einige Besonderheiten beachtet werden. So musste zum Beispiel vorher ein virtuelles FileSystem mit Emscripten eingerichtet werden, da die C++ Bibliothek FileSystem Operationen nutzt.

Nach dieser Übersetzung steht schließlich eine JavaScript Bibliothek zur Verfügung, welche im Web Worker verwendet werden kann. Diese benötigt für die Dekodierung eines 2048×2048 Bildes allerdings vier Sekunden statt 50 ms. Wird für ein 2048×2048 JPEG Bild 2 MB benötigt, lohnt sich die Verwendung dieser Technik daher nur für Verbindungen, welche langsamer als 4 Mbit sind. Bei schnelleren Verbindungen ist das Bild heruntergeladen, bevor überhaupt eine progressive Vorschau angezeigt werden kann. Falls jedoch mehr als nur ein Bild heruntergeladen werden soll, lohnt sich der Einsatz auch bei schnelleren Verbindungen. So ist es zum Beispiel mit Firefox möglich, bis zu sechs parallele Verbindungen aufzubauen, womit sich der Ansatz bis zu einer Geschwindigkeit von 24 Mbit lohnt.

9.2 Bewertung

Durch das Streaming ist eine relativ gute Vorschau der Textur möglich. Die Abbildung 9.1 zeigt die progressiven Zwischenstufen einer JPEG Textur während des Streaming. In dieser ist zu erkennen, dass bereits ab ungefähr 30% der heruntergeladenen Datenmenge die Textur gut dargestellt wird. Danach wird diese weiter verfeinert, dies ist allerdings nur in Detailansichten zu erkennen. Daher kann ohne einen zusätzlichen Daten-Overhead die ursprüngliche Auflösung beibehalten werden und dennoch eine schnelle Vorschau ermöglicht werden.

Das Streaming von progressiven JPEG Texturen ist nur sinnvoll, falls der Dekodierer wesentlich schneller als die Verbindungsgeschwindigkeit ist und sehr viele JPEG Bilder gleichzeitig übertragen werden. Falls eine schnelle Verbindungsgeschwindigkeit vorliegt, lohnt sich dieser Ansatz nicht und das Bild sollte stattdessen ohne progressive Vorschau übertragen werden. Besonders mit dem Ansatz über Emscripten ist die Dekodierungsgeschwindigkeit zu langsam. Um diese zu verbessern, muss eine schnellere JPEG Bibliothek als die Referenzimplementierung bei der Übersetzung durch das Programm verwendet werden.

Die schnellste Dekodierungsgeschwindigkeit bietet das native Image-Element von JavaScript. Da dieses aber nicht innerhalb eines Web Workers verwendet werden kann, führt die Dekodierung von großen Bildern zu Aussetzern. Da das Element den DOM nicht verändert, ist es vorstellbar, dass diese Einschränkung in einer weiteren Iteration der Web Worker wegfällt. Damit wäre der Ansatz wesentlich praxistauglicher.

In bestimmten Anwendungsfällen, in denen kurze Ruckler akzeptabel sind, kann auf den Einsatz von Web Workern verzichtet werden. Dies kann zum Beispiel bei 3D-Objektkatalogen annehmbar sein, in denen das Modell nur durch die Interaktion des Nutzers beeinflusst wird und so die kleineren Ruckler nicht weiter auffallen.

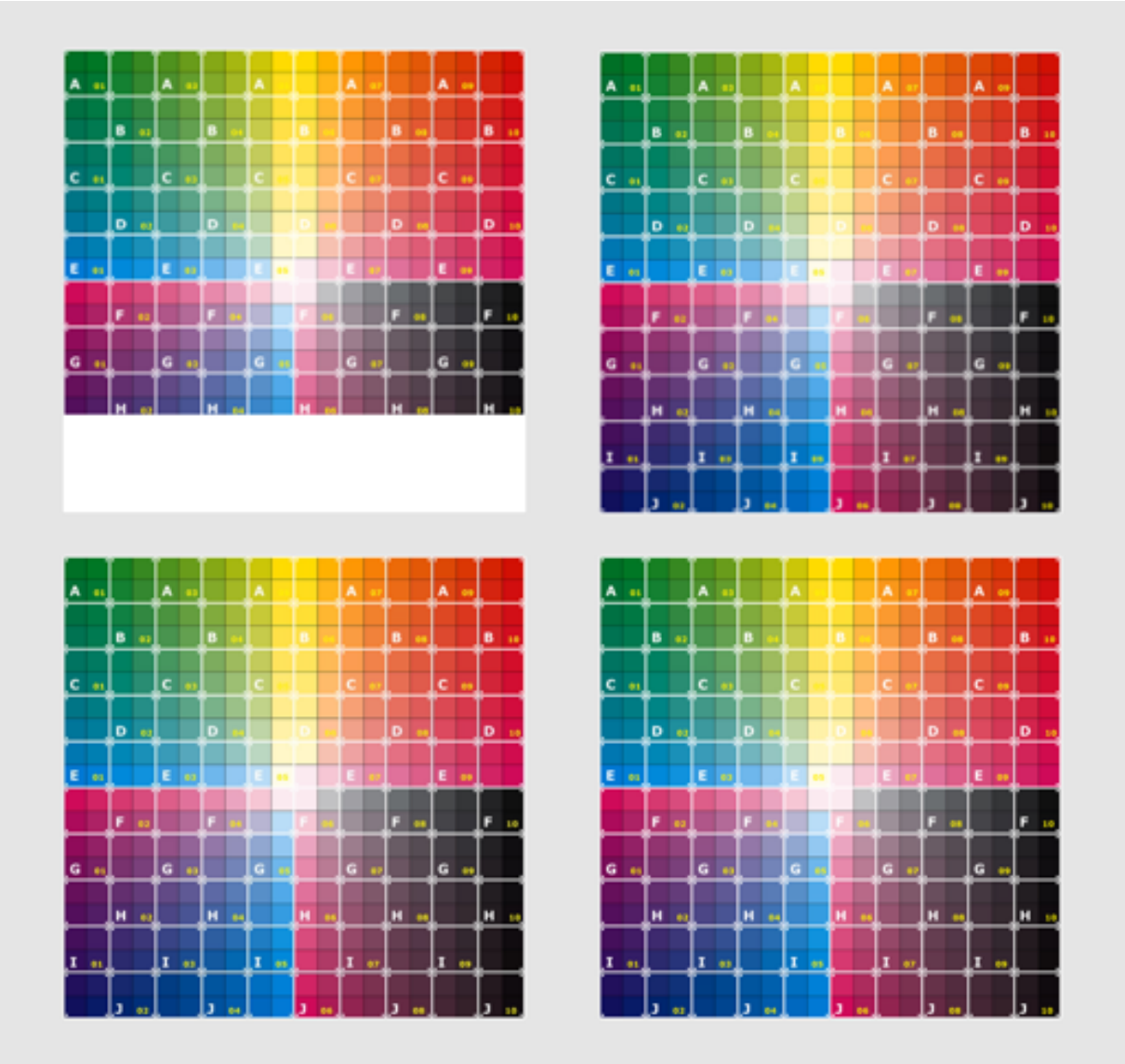

Abbildung 9.1: Die Abbildung zeigt von links oben bis rechts unten die Zwischenstufen 100 kB, 250 kB, 500 kB und 1,57 MB. Ab 250 kB ist bereits eine gute Vorschau der Textur erreicht. Die letzten beiden Stufen sind in der gerenderten Auflösung kaum voneinander zu unterscheiden.

10 Caching

Für moderne Webanwendungen, welche viele Ressourcen nutzen, reicht es nicht aus, die benötigten Daten nur möglichst kompakt zu übertragen. Zusätzlich sollen Daten, welche bereits heruntergeladen wurden, möglichst persistent beim Client gespeichert werden. Damit soll ein erneutes Laden aller Daten, beim zweiten Besuch der Webanwendung, vermieden werden. Neben dem Geschwindigkeitsvorteil ist Caching notwendig, um die Kosten für den Datentransfer vom Nutzer zum Server gering zu halten.

Vor der Einführung von HTML5 musste sich der Entwickler dabei generell mit einem Limit von 5 MB zufrieden geben. Mittlerweile existieren verschiedene Lösungen, um über diese Begrenzung hinaus Daten zu speichern. Der folgende Abschnitt beschäftigt sich mit der Analyse der dabei zur Verfügung stehenden Schnittstellen sowie deren jeweiligen Vor- und Nachteile.

10.1 Browser Cache

Da die Verbindungsgeschwindigkeit zum Netzwerk oft den limitierenden Faktor beim Rendern von Webseiten darstellt, ist das Caching bereits heruntergeladener Daten notwendig. „Der schnellste HTTP Request ist der, welcher nicht ausgeführt wird“ ist dabei ein oft erwähntes Motto. Nahezu jeder moderne Browser bietet hierfür einen internen Cache, über welchen alle Verbindungen geleitet werden. Bei den meisten Browsern ist dieser Cache zudem in einem Hauptspeicher und Festplatten Cache unterteilt.

Der typische Ablauf des Caching ist folgender [Law10]:
Der Client fordert über einen *GET* Request eine Datei vom Server an. Er übergibt dabei als Parameter *max-age=0*. Damit fordert er den Webserver auf, ihm eine aktuelle Version der Datei zu schicken. Wird die Datei beim Server gefunden und der Client hat ausreichende Rechte diese aufzurufen, so antwortet der Server mit einem *HTTP/200 OK* Response und dem Inhalt der Datei. Unterstützt der Server Cache-Control, so wird in dem Header zusätzlich der Parameter *max-age* angegeben. Dieser Parameter kann vom Serverbetreiber zum Beispiel in der *.htaccess* Datei für jeden Datentyp festgelegt werden. Anschließend kann er vom Browser genutzt werden, um festzustellen, wie lange eine Datei im Cache gültig ist. Wenn vom Server kein Cache-Control Parameter übergeben wurde, übernimmt bei einigen Browsern eine Heuristik die Abschätzung des *max-age* Parameters.

Wird die Datei nun nochmals vom Client benötigt, kann dieser über den *max-age* Parameter ermitteln, ob die Version dieser aus dem Cache genutzt werden kann. Ist dies möglich, fällt keine zusätzliche Anfrage an den Server an und die Datei aus dem Cache wird verwendet. Falls der *max-age* Parameter die aktuelle Zeit überschritten hat, muss eine Anfrage an den Server gesendet werden. Dabei wird dem Server im Header der Anfrage ein *if-modified-since* Parameter übergeben. Der Server kann über diesen Parameter schließlich ermitteln, ob die Datei tatsächlich gesendet werden soll. Ist dies der Fall, wird die Anfrage mit dem Dateiinhalt und einer *HTTP/200 OK* Response bestätigt. Falls die Datei in der Zwischenzeit auf dem Server nicht geändert wurde, antwortet der Server lediglich mit einer *HTTP/304 Not-Modified* Response und weist den Browser somit an, die Datei im Cache zu verwenden.

Da der Browser Cache mit anderen Webseiten geteilt wird und eine begrenzte Größe aufweist, müssen, sobald der Cache gefüllt wurde, Dateien entfernt werden. Welche von diesen konkret entfernt werden, hängt je nach Implementation der Cache Strategie des Browsers ab. Der Internet Explorer 9 vergibt zum Beispiel an jede Datei maximal 66.000 Punkte. Die ersten 40.000 Punkte werden danach vergeben, wann die Datei zuletzt verwendet wurde. Weitere 20.000 Punkte werden danach zugeteilt, wie oft sie verwendet

wurde. Die restlichen 6.000 Punkte richten sich nach den Informationen im Header *Last-Modified* und dem *ETag* [Law12]. JavaScript, CSS und HTML Dateien erhalten dabei die vollen 66.000 Punkte zur Vergabe, da diese besonders wichtig für die Darstellung einer Internet-Seite sind. Die restlichen Daten erhalten lediglich 33.000 Punkte. Die Dateien mit der geringsten Punkteanzahl werden aus dem Cache entfernt. Der Anwendungsentwickler hat also keinen direkten Einfluss, ob eine Datei tatsächlich für eine längere Zeit gecacht wird.

Die Größe des Browser Caches variiert abhängig vom Browserhersteller sehr stark [Sou12b]. Im Listing 10.1 sind die Cachegrößen gängiger Browser aufgeführt.

Cachegrößen gängiger Browser	
Browser	Cachegröße
Chrome 21	320 MB
Internet Explorer 9	250 MB
Firefox 11	830 MB
Opera 11	20 MB
Safari Mobile 5.1	30 - 35 MB
Galaxy Nexus	18 MB

Tabelle 10.1: Die Cachegröße variiert stark je nach Implementation der Browserhersteller [Sou12b][Law12]. So unterstützt Opera nur einen Cache von 20 MB, während Firefox einen großzügigeren Cache von 830 MB bietet.

Anzumerken ist dabei, dass diese Abbildung jeweils nur die Größe des Caches auf der Festplatte des Nutzer angibt. Zusätzlich nutzen die meisten Browser einen weiteren Cache im Hauptspeicher des Nutzers. Die Annahme und damit der Grund des kleinen Caches von Opera ist dabei, dass der Nutzer den Browser lange geöffnet hält und so der Cache im Hauptspeicher für die aktuelle Sitzung wichtiger als der Festplatten Cache ist. Für eine Webanwendung, welche viele Daten auf der Festplatte des Nutzers zwischenspeichert, ist diese Annahme jedoch ungünstig. Der Cache im Hauptspeicher wird sofort gelöscht, sobald der Browser geschlossen wird und steht so einer neuen Instanz nicht mehr zur Verfügung. Die kleine Cachegröße auf der Festplatte sorgt dafür, dass bereits heruntergeladene Modelle sehr schnell aus dem Cache entfernt werden, sobald der Nutzer andere Webseiten besucht.

Chrome nutzt je nach Ressource, Media (Audio und Video) oder andere, einen separaten Cache auf der Festplatte des Nutzers. Die Größe des Caches richtet sich dabei nach der Größe der Festplatte des Nutzers mit einem Limit von 320 MB [Goo12b]. Dieses Limit wurde von dem Chromium Team über eine Nutzerstudie festgelegt. Dabei fand man heraus, dass über diese Begrenzung die Suchzeit der Daten auf der Festplatte des Nutzers die Zeit zum Herunterladen der Ressource übersteigen würde und somit die Performance generell verringern würde. Dieser Test wurde jedoch durchgeführt, als SSD Festplatten noch nicht gängig waren. Es ist also denkbar, dass dieses Limit zu gegebener Zeit weiter angehoben wird.

Für den hier aufgeführten Anwendungsfall ist es wünschenswert, dass die heruntergeladenen Dateien persistent über mehrere Browserinstanzen beim Client abgespeichert werden. Da der Browser zwischen den einzelnen Aufrufen des Spieles jedoch wahrscheinlich für weitere Aktivitäten verwendet wird, ist anzunehmen, dass der Cache des Browsers für eine persistente Speicherung der Modelle und Texturen wenig geeignet ist. Demnach müssen andere Schnittstellen zum Caching dieser gefunden werden. Seit HTML5 existieren dabei eine Vielzahl von verschiedenen Möglichkeiten, auf welche im Folgenden genauer eingegangen wird.

10.2 Application Cache

Ziel des Application Caches ist es, dem Nutzer Webseiten offline zur Verfügung zu stellen. Die Spezifikation des Application Caches ist Teil des HTML5 Standards, welcher sich zum Stand der Arbeit noch ein Arbeitsentwurf des W3C ist. Trotzdem ist dieser bereits in Chrome, Firefox, Safari, Opera und einigen mobilen Browsern verfügbar.

Mit Application Cache werden alle benötigten Daten einer Webanwendung in einem separaten Cache abgelegt. Vom Entwickler müssen die Pfade dieser Daten dabei in einem speziellen Manifest angegeben werden. Dieses legt fest, welche Dateien im Cache gespeichert oder, falls nicht vorhanden, über das Netzwerk aufgerufen werden sollen. Das Manifest erlaubt zusätzlich die Angabe einer Fallback Ressource, falls der Nutzer sich im Offline-Zustand befindet. Letzteres ist besonders für Anwendungen, wie zum Beispiel E-Mail Clients, wichtig. Diese können im Offline-Zustand die bereits gespeicherten E-Mails anzeigen.

Das Codebeispiel 10.1 zeigt dabei ein mögliches Manifest, um die angegebenen Dateien im Cache abzulegen. Diese Dateien werden im Application Cache des Browsers gespeichert. Alle restlichen Dateien werden über das Netzwerk heruntergeladen.

```
1 CACHE MANIFEST
2 CACHE:
3 js/resourceLoader.js
4 data/model.js
5 data/uv.jpg
6 NETWORK:
7 *
```

Listing 10.1: Application Cache Manifest Beispiel, welches nur die angegebenen Dateien im Cache speichert. Die restlichen Daten werden durch die Sternnotation im *NETWORK* Abschnitt über das Netzwerk geladen.

Damit der Application Cache vom Browser genutzt werden kann, muss das Manifest mit dem MIME Typ *text/cache-manifest* übertragen werden. Zusätzlich muss der Entwickler das Manifest in jeder HTML Datei, welche den Application Cache verwenden soll, im HTML Header entsprechend verlinken.

Nach dem Aufruf einer Internet-Seite überprüft der Browser, ob eine Manifest Datei im HTML Header angegeben wurde [WHA12]. Dieses Manifest und alle in ihm angegebenen Ressourcen werden anschließend heruntergeladen, nachdem die Seite gerendert wurde. Falls der Download für eine Ressource fehlschlägt, wird das Manifest mitsamt den heruntergeladenen Ressourcen verworfen. Bei einem erneuten Aufruf der Internet-Seite wird überprüft, ob das Manifest verändert wurde. Bei einer Veränderung, wird dies vermerkt, aber vorerst die alten Ressourcen verwendet. Bei dem nächsten Aufruf der Seite werden alle im Manifest angegebenen Daten über XHR Requests heruntergeladen. Dementsprechend werden Dateien, welche bereits im Browser-Cache vorliegen, nicht nochmals heruntergeladen.

10.2.1 Bewertung

Die konkrete Implementation des Application Caches ist stark vom Browserhersteller abhängig. So bieten Firefox und Opera einen geteilten persistenten Speicher für alle Anwendungen. Ist dieser voll, wird keine ältere Datei aus dem Speicher gelöscht. Neue Daten können nicht mehr hinzugefügt werden, bis der Nutzer den Speicher manuell leert. Damit eine Anwendung diesen persistenten Speicher beschreiben kann, muss diese vorher die entsprechende Berechtigung vom Nutzer anfordern. Dies wird dem Nutzer über ein Popup, ähnlich dem Passwort Speichern Dialog, angezeigt.

In Chrome wird dagegen ein geteilter temporärer Speicher verwendet [Goo12a]. Für diesen muss keine zusätzliche Berechtigung vom Nutzer angefordert werden. Dafür kann allerdings der gesamte Application Cache einer Anwendung gelöscht werden, sobald der Speicher des geteilten temporären Caches das Limit übersteigt.

Der Application Cache erlaubt kein dynamisches Hinzufügen von Ressourcen über JavaScript. Lediglich für Firefox existiert die Erweiterung *applicationCache.mozAdd()* [She12], welche das dynamische Hinzufügen erlaubt. Für den hier aufgeführten Anwendungsfall müssen daher alle von der Anwendung verwendeten Modelle und Texturen bereits im Vorfeld in der Manifest Datei angegeben werden. Dies führt allerdings dazu, dass alle angegebenen Dateien beim ersten Aufruf der Seite heruntergeladen werden müssen. Ganz unabhängig davon, ob diese auch tatsächlich benötigt werden. Es kann somit keine Priorisierung der Daten vorgenommen werden. Für Spiele wäre es zum Beispiel wünschenswert, dass vorerst alle Modelle eines Levels mit dem geringsten Level of Detail geladen werden.

Der Vorteil des Application Caches liegt in der einfachen Implementation. Es sind nur wenige Schritte nötig, um Dateien im Application Cache zu speichern. Der gesamte Vorgang ist zudem sehr transparent. Es können normale XHR Requests durchgeführt werden, welche automatisch den Application Cache nutzen, sobald dieser vorliegt. Besonders für mobile Browser und Browser mit kleinem Festplatten Cache lohnt sich der Einsatz des Application Cache [Sou12a].

10.3 File System API

Oft ist es wünschenswert, Daten strukturiert abzulegen. Die Schnittstelle File System API erlaubt es einer Webanwendung ein lokales File System bereitzustellen [Lim12b]. Dieses File System kann dazu verwendet werden, um Dateien und Ordner zu erstellen und zu lesen. Alle Daten werden in einer Sandbox abgelegt und sind somit nicht für andere Anwendungen zugänglich. Zum Stand dieser Arbeit ist diese Schnittstelle nur in Chrome verfügbar.

Sie ist dafür ausgelegt, binäre und größere Blöcke von Daten abzuspeichern. Der Zugriff erfolgt dabei, im Gegensatz zu üblichen normalen File System Zugriffen, asynchron, damit die GUI nicht blockiert wird. Für die Verwendung der File System API in einem Web Worker, um den Prozess auf einen anderen Thread auszulagern, kann eine synchrone Implementation verwendet werden. In Chrome existieren drei Speicherbereiche, welche jeweils unterschiedliche Anwendungsfälle abdecken. Zur Auswahl stehen ein temporärer, persistenter und „unbegrenzter“Speicherbereich. Letzterer ist nur verfügbar, falls die Anwendung über dem Chrome Store ausgeliefert wird, weshalb dieser im Folgendem nicht weiter beachtet wird.

Der temporäre Speicherbereich wird von allen Webanwendungen geteilt. Die Größe richtet sich dabei nach dem verfügbaren Speicherplatz auf der Festplatte des Nutzers. Sie beträgt 50% des verfügbaren Speicherplatzes. Jede Anwendung kann zudem jeweils bis zu 20% dieses Speicherplatzes einnehmen. Falls nicht genügend Speicherplatz zur Verfügung steht, löscht der Browser Daten älterer Anwendungen aus dem Speicherbereich. Er wählt diese Anwendungen nach dem Least Recently Used Prinzip [Bid12] aus. Es werden dabei alle Dateien einer Anwendung gelöscht, um Inkonsistenzen zu vermeiden.

Persistenter Speicher muss vom Entwickler explizit angefordert und vom Nutzer bestätigt werden. Die angeforderte Größe kann dabei so groß wie der verfügbare Speicherplatz der Festplatte sein. Nachdem der Nutzer die Verwendung bestätigt hat, steht der Inhalt des Speichers solange zur Verfügung, bis dieser manuell von ihm gelöscht wird.

Das Codebeispiel 10.2 zeigt, wie ein einfaches Filesystem angelegt wird. Dabei fordert der Entwickler eine Größe von insgesamt 5 MB an. Falls genügend freier Speicherplatz im geteilten temporären Bereich bereitgestellt werden kann, wird die Funktion *initFS()* aufgerufen. Falls nicht genügend Speicher zur Verfügung steht oder ein anderer Fehler auftritt, wird die Funktion *errorFs()* aufgerufen.

```
window.requestFileSystem(
  TEMPORARY,           // persistenter oder temporärer Speicher
  5 * 1024 * 1024,     // Größe in Bytes, welche angefordert werden soll (hier 5 MB)
  initFs,              // Callback bei erfolgreichem Anlegen
  errorFs              // Callback bei einem Fehler
);
```

Listing 10.2: Ein asynchrones, temporäres File System wird angelegt. Die angeforderte Größe des File Systems beträgt 5 MB. Falls dieser Speicher bereitgestellt werden kann, wird die Funktion *initFS* aufgerufen. Bei einem Fehler wird die Funktion *errorFs* aufgerufen.

10.3.1 Bewertung

Die Verwendung der File System API kann unter Umständen einen Performance-Vorteil gegenüber der Verwendung des Application Caches bringen. So können zum Beispiel heruntergeladene Daten in JavaScript auf dem Client bearbeitet und das Ergebnis, zur Wiederverwendung, gespeichert werden.

Der Temporäre Speicher ist gut geeignet, um Dateien zu cachen. Falls genügend Speicherplatz auf der Festplatte des Nutzers zur Verfügung steht und der Nutzer nicht mehrere Anwendungen, welche alle den maximalen Speicher anfordern, aktiv nutzt, ist es wahrscheinlich, dass die Daten im Cache auch über eine längere Zeit verfügbar sind.

Persistenter Speicherplatz wird für Daten verwendet, welche vom Nutzer erstellt wurden oder auch offline zur Verfügung stehen sollen. Da persistenter Speicher jedoch immer eine Bestätigung des Nutzers erfordert, muss vom Entwickler ein entsprechendes Fallback zu dem temporären Speicherbereich eingerichtet werden, falls der Nutzer die Verwendung des persistenten Speichers verweigert. Dies kann allerdings nur vorgenommen werden, falls die Anwendung auch ohne die Persistenz der Daten auskommt. Das ist zum Beispiel der Fall, wenn mehrere größere Modelle und Texturen abgespeichert werden sollen. Hier wäre ein persistenter Speicher wünschenswert, um die Anwendung auch im offline Modus verwenden zu können. Allerdings reicht es für viele Nutzer, die Daten im temporären Speicherbereich abzulegen. Möglich wäre es, einen Mix von temporären File System und Web Storage zu nutzen.

Der größte Nachteil der File System API liegt in der limitierten Verfügbarkeit. Nur Chrome unterstützt zur Zeit die File System API. Die anderen Hersteller haben zum Stand der Arbeit noch keine feste Aussage getroffen, ob diese Schnittstelle unterstützt werden soll.

10.4 Web Storage

Vor der Einführung von HTML5 war es gängig, Daten des Clients in Cookies abzuspeichern. Diese bieten allerdings lediglich 4 kB Speicherplatz. Des Weiteren werden diese über den Server ausgetauscht, was einige Sicherheitsprobleme mit sich bringt. Web Storage [Hei12], auch DOM Storage genannt, ist als Ersatz für Cookies entworfen wurden [Bid12]. Die Spezifikation legt kein Limit für den Speicherplatz fest, empfiehlt allerdings ein Limit von 5 MB pro Origin. Ein Host teilt sich so für jeweils ein Protokoll (http, https) diesen Speicherplatz. Das Limit von 5 MB wird von den meisten Browsern als Obergrenze unterstützt. Die Größe beträgt lediglich 5 MB, da es sich bei Web Storage um eine synchrone API handelt [Orl10]. Jeder Zugriff auf den Web Storage ist daher ein blockierender Vorgang. Web Storage ist ein Kandidat für eine W3C Empfehlung und wird von den meisten Browsern bereits unterstützt.

Web Storage erlaubt das Abspeichern über eine Key/Value Hashtable. Dabei werden lediglich Strings als Typ unterstützt. Um JavaScript Objekte abzuspeichern, muss somit das Objekt vorher über *JSON.stringify()* konvertiert werden. Für binäre Daten, wie zum Beispiel Texturen, bietet sich eine Konvertierung in Base64 an. Das Codebeispiel 10.3 zeigt, wie eine Zeichenkette abgespeichert und wieder abgerufen werden kann.

```
localStorage.setItem('storeKey', 'store me');
localStorage.getItem('storeKey'); // gibt 'store me' aus
```

Listing 10.3: Über Web Storage wird der String *store me* persistent abgespeichert. Als Key wurde *storeKey* gewählt. Mit diesem ist es möglich, später den gespeicherten Wert zu ermitteln.

Die Daten können entweder nur für eine Sitzung oder persistent gespeichert werden. Eine Sitzung ist solange gültig, wie der Browser geöffnet ist. Zur Unterscheidung beider Speicherarten muss der Entwickler unterschiedliche Objekte nutzen. Der persistente Speicher wird über das Objekt *localStorage* angesprochen. Das Objekt *sessionStorage* ist dagegen für die Speicherung von temporären Daten zuständig.

10.4.1 Bewertung

Durch die synchrone API und das 5 MB Limit ist Web Storage nicht geeignet, um mehrere größere Daten, wie 3D-Modelle, abzulegen [Hei12]. Web Storage führt File I/O Operationen synchron aus, was bedeutet, dass unter Umständen die gesamte Webanwendung blockiert wird, solange Web Storage ausgeführt wird. Des Weiteren wird der gesamte Inhalt des Web Storage beim Aufruf der Anwendung in den Hauptspeicher geladen. Ganz unabhängig davon, ob dieser auch tatsächlich genutzt wird. Somit ist es unwahrscheinlich, dass das Limit von 5 MB in Zukunft erhöht wird.

Die Schnittstelle kann jedoch dafür genutzt werden, um kleinere Daten wie Konfigurationseinstellungen persistent abzuspeichern. In dem Browser Chrome ist dies andernfalls nur über die File System API und mit Einstimmung des Nutzers möglich.

10.5 Web SQL Database

Die vorangegangenen Lösungen sind wenig geeignet, um Beziehungen zwischen den Daten abzulegen. Dafür wird eine Datenbank benötigt, welche gezielte Abfragen erlaubt. Mit dieser ist es möglich diese Beziehungen untereinander auszunutzen, um komplexe Anfragen aufzulösen. So können zum Beispiel alle Modelle des geringsten Level of Detail, welche im ersten Level platziert sind, mit einer einzigen Anfrage gefunden werden.

Web SQL Database bietet eine auf SQLite basierende Datenbank an. Zum Stand dieser Arbeit wurden die Arbeiten an der Schnittstelle eingestellt, da alle Implementationen als SQL backend SQLite verwenden. Für einen W3C Standard ist es allerdings nötig, dass mindestens zwei unabhängige Implementationen vorliegen. Mozilla und Microsoft werden die Schnittstelle nach Stand dieser Arbeit nicht verwenden. Beide Hersteller wollen nicht an eine Version von SQLite gebunden sein, bei der unsicher ist, ob diese in Zukunft noch kompatibel mit neueren Versionen ist [Ran09] [Ran10]. Implementiert wurde die Schnittstelle in Chrome, Safari, Opera und einigen mobilen Browsern. Web SQL Database wurde entwickelt, um Datenbank Funktionen auf dem Client auszuführen. Damit kann zum Beispiel Last vom Server auf die Clients verteilt oder komplexe Beziehungen zwischen den Daten ermittelt werden.

Die Größe der Datenbank kann beim Erstellen dieser vom Browser angefordert werden. Der Nutzer wird in einigen Browsern aufgefordert, diese Größe zu bestätigen, da es sich um persistenten Speicher handelt. In Chrome wird der temporäre Speicherplatz verwendet, weshalb der Nutzer keine Benachrichtigung erhält. Die Schnittstelle bietet eine synchrone und asynchrone Implementation. Das Codebeispiel 10.4 legt eine Datenbank mit dem Namen *dbname* in der Version 1.0 und einer initialen Größe von 2 MB an. Nach erfolgreichem Anlegen der Datenbank wird die Funktion *initDB()* aufgerufen.
Nachdem die Datenbank angelegt wurde, können Tabellen erstellt und diesen Daten mittels SQL Funktionen hinzugefügt werden. Mehrere Abfragen werden über Transaktionen, welche die ACID (atomicity, consistency, isolation, durability) Kriterien sicherstellen, durchgeführt. Binäre Daten können in der Da-

```
var db = window.openDatabase(
  'dbname',          // Datenbankname
  '1.0',             // Version
  'my database',     // Kommentar
  2 * 1024 * 1024,   // Initiale Größe in Bytes, hier 2 MB
  initDB             // Callback, wenn Datenbank angelegt wurde
);
```

Listing 10.4: Im Beispiel wird mit der Schnittstelle Web SQL eine Datenbank mit dem Namen *dbname* in der Version 1.0 und einer initialen Größe von 2 MB angelegt. Ist dies erfolgreich, so wird die Funktion *initDB()* aufgerufen.

tenbank in einem BLOB Objekt verpackt und abgelegt werden. Codebeispiel 10.5 zeigt, wie ein solches Objekt in einer Datenbank gespeichert wird.

```
var model = {
  "id": new Date().getTime(),
  "data": blob
};

database.transaction(function(tx) {
  tx.executeSql('INSERT INTO models(id,data) values(?,?)',[model.id,model.data]);
});
```

Listing 10.5: Speichern von binären Daten in einer Web SQL Datenbank. Vorerst wird ein JavaScript Objekt erstellt, welches eine ID und ein Datenattribut besitzt. Diesem Datenattribut wird ein binäres BLOB Element zugewiesen. Das Konstrukt wird anschließend in der Datenbanktabelle *models* gespeichert. Dabei wird über eine Transaktion sichergestellt, dass dies erfolgreich durchgeführt wird.

10.5.1 Bewertung

In der Datenbank können binäre Daten flexibel abgelegt werden. Durch die Nutzung von SQL können ebenfalls komplexe Anfragen an die Datenbank gestellt werden. Damit können zur Laufzeit des Programms die Modelle ermittelt werden, welche für die aktuelle Szene nötig sind. Jedoch ist die Implementierung dieser Schnittstelle komplexer als beim Application Cache.

Mozilla und Microsoft werden die Schnittstelle, nach Stand der Arbeit, nicht implementieren. Die beiden Hersteller wollen statt Web SQL Database, die Schnittstelle IndexedDB als Standard vorantreiben [Ran10]. Beide Hersteller befürchten, dass durch die Bindung von Web SQL Database an SQLite möglicherweise Sicherheitslücken und Inkompatibilitäten bei späteren Updates auftreten. Die Spezifikation von Web SQL Database wurde vorerst eingestellt. Da IndexedDB jedoch speziell auf mobilen Plattformen noch nicht überall verfügbar ist, ist Web SQL Database die einzige Alternative für die Nutzung einer Datenbank auf diesen Plattformen.

10.6 IndexedDB

Ähnlich zu Web SQL Database ist es mit der Schnittstelle IndexedDB [MSG+11] möglich, Daten strukturiert abzulegen. Die Schnittstelle wird von Mozilla und Microsoft als Datenbank-Standard fürs Web vorangetrieben, da beide Web SQL Database als ungeeignet erachten [Ran10]. Im Gegensatz zu Web SQL Database handelt es sich bei IndexedDB um keine relationale sondern um eine NoSQL Datenbank. Daten werden in key/value Paaren abgespeichert. Sämtliche Operationen auf der Datenbank werden ebenfalls, wie in Web SQL Database, in Transaktionen durchgeführt. Das Codebeispiel 10.6 zeigt wie eine

Datenbank angelegt wird und in dieser anschließend ein Bild gespeichert und wieder aufgerufen wird. Dabei wird zuerst eine Datenbank angefordert und, nach dem erfolgreichen anlegen dieser, in dieser ein *ObjectStore* mit dem Namen *Images* erstellt. In diesem wird über eine Transaktion schließlich ein BLOB Objekt abgelegt, welches als Key den Wert *image* erhält. Nun kann ebenfalls über eine Transaktion über den selben Key aus dem *ObjectStore* das BLOB Object wieder ausgelesen werden.

```
1  // create database
2  var db;
3  var request = indexedDB.open('Database');
4  request.onerror = function(event) {};
5  request.onsuccess = function(event) {
6    db = request.result;
7    db.createObjectStore('images');
8  };
9
10 // put image into models
11 var transaction = db.transaction(["images"], IDBTransaction.READ_WRITE);
12 transaction.objectStore("images").put(blob, "image");
13
14 // get image file
15 transaction.objectStore("images").get("image").onsuccess = function (event) {
16   var imgFile = event.target.result;
17   ...
18 };
```

Listing 10.6: Verwendung der IndexedDB Schnittstelle zum Speichern von Bildern. In Zeile 2 bis 8 wird eine Datenbank angelegt beziehungsweise geöffnet. In dieser wird ein neuer *ObjectStore* angelegt. Über eine Transaktion, welche diesen *ObjectStore* nutzt, wird das Bild, welches sich in einem BLOB Objekt befindet, über die Funktion *put()* gespeichert. Anschließend kann das Bild über die Funktion *get()* wieder aufgerufen werden.

Die Schnittstelle ist zum Stand der Arbeit in Firefox und Chrome verfügbar. Für den Internet Explorer 10 existiert zur Zeit ein Prototyp [CC12]. Die Standardisierung beim W3C befindet sich im Working Draft Zustand. Die Spezifikation bietet neben der asynchronen zusätzlich eine synchrone Version [Lim12a], welche allerdings derzeit von keinen Browserhersteller implementiert wurde. Die Größe des Speichers richtet sich nach dem verfügbaren Platz auf der Festplatte des Nutzers. Er gilt für jede Anwendung der selben Origin. In Firefox muss der Speicher explizit vom Nutzer angefordert werden. In Chrome nutzt IndexedDB den temporären Speicherbereich. Die Nutzung des Speicherbereiches muss nicht extra bestätigt werden.

10.6.1 Bewertung

IndexedDB wurde konzipiert, um größere Daten effektiv abzulegen. Die Schnittstelle bietet mehr Speicherplatz als Web Storage und kann zu dem flexibel Strukturen von Daten ablegen. Ähnlich zur File System API können auch hier heruntergeladene Daten in JavaScript auf dem Client bearbeitet und das Ergebnis gespeichert werden. Dies bringt einen Geschwindigkeitsvorteil, falls die übertragenen Daten vorerst entpackt werden müssen. Zum Stand der Arbeit ist die Schnittstelle noch nicht in Safari, Opera und den mobilen Browsern verfügbar. Somit stellt auch diese Schnittstelle keine Lösung für alle Plattformen dar.

10.7 Einsatz der Caching Schnittstellen

Für Anwendungsfälle in denen bereits im Vorfeld bekannt ist, welche Daten zu speichern sind, bietet sich die Schnittstelle Application Cache an. So eine Anwendung könnte zum Beispiel lediglich eine Szene

enthalten, in welcher alle Objekte sofort angezeigt werden. Die Schnittstelle Application Cache kann bei solch einer Anwendung ohne größeren Aufwand in eine vorhandenen Codebasis integriert werden. Wenn keine weiteren Dateien aus dem Netzwerk nötig sind, kann mit dieser Schnittstelle ebenfalls ein Offline-Modus realisiert werden.

Sie ist jedoch ungeeignet, falls nicht im Vorfeld ermittelt werden kann, welche Dateien beim Client gecacht werden sollen. Dies ist zum Beispiel bei einem Spiel der Fall, welches dynamisch für jeden Abschnitt des Spieles die Dateien herunterlädt. Hierbei ist nicht im Vorfeld bekannt, welches Level der Spieler auswählt und welche Modelle daher benötigt werden. Würde die Schnittstelle Application Cache verwenden, müssten alle Daten heruntergeladen werden, ganz unabhängig davon ob der Spieler diese überhaupt benötigt. Mit den Schnittstellen IndexedDB und Web SQL Database kann der Entwickler diese Daten dynamisch ablegen. Es ist sogar möglich im ersten Ladevorgang komprimierte Modelle im OpenCTM MG2 Format an den Client zu schicken. Dieser entpackt das Format lokal und legt über die Schnittstellen die entpackte Version ab. Damit entfällt die erneute Dekodierung und die Anwendung kann beim zweiten Besuch wesentlich schneller gestartet werden.

Es existiert zum Stand der Arbeit für das dynamische Ablegen der Daten jedoch keine Schnittstelle, welche auf allen Browsern verfügbar ist. Deswegen muss vom Entwickler ein entsprechender Fallback programmiert werden, in welchem je nach Verfügbarkeit, die richtige Schnittstelle genutzt wird. Dies bedeutet allerdings einen erheblichen Implementierungsaufwand.

Der dynamische und statische Ansatz kann zusätzlich miteinander verbunden werden. So können die Daten, welche von allen Bereichen der Anwendung genutzt werden, mit der Schnittstelle Application Cache abgelegt werden. Dynamische Daten können dagegen mit IndexedDB oder Web SQL Database abgespeichert werden.

11 Fazit und Ausblick

In dieser Arbeit wurden Methoden und Formate vorgestellt, welche für die Kompression der Geometriedaten verwendet werden können. Ist das Ziel, möglichst viel Bandbreite einzusparen, bietet sich der MG2 Modus von OpenCTM an, da mit diesem Format die besten Kompressionsraten erzielt werden. Steht jedoch die Zeit vom Download bis zur Anzeige des Modells beim Nutzer im Vordergrund, liefert das WebGL Loader Format bei schnellen Verbindungsgeschwindigkeiten die bessere Performance.

Für eine progressive Vorschau kann das Image Geometrie Format verwendet werden. Es eignet sich besonders, da so keine weiteren Daten bei der Übertragung anfallen. Die Qualität der Vorschau ist jedoch nicht für alle Anwendungsfälle geeignet. Die beste Qualität wird erreicht, indem die einzelnen Level of Details als progressive Stufen übertragen werden. Diese lassen sich durch ein Kompressionsformat, wie WebGL Loader oder dem MG2 Modus von OpenCTM, komprimieren und schrittweise übertragen.

Der Ansatz, Modelle progressiv nach Hoppe et al. zu übertragen, eignet sich nur in einigen Fällen, in denen die Vertex Splits effektiv abgelegt werden können. Das Modell darf dabei nicht aus mehreren getrennten Untermodellen bestehen, da andernfalls die progressive Zerlegung keine guten Resultate erzielt. Das verwendete Vereinfachungsverfahren muss dies entweder beachten oder das Modell sollte vorher in seine einzelnen Teile zerlegt werden. Ebenfalls sollte eine effektivere Speicherung der Vertex Splits, ohne redundante Daten, gefunden werden.

Das Streaming der Texturen kann darüber hinaus weiterhin verbessert werden, indem eine schnellere Implementation der Dekodierungsbibliothek von progressiven JPEG Bildern in JavaScript geschrieben wird. Allerdings ist ebenso eine Erweiterung der Web Worker Spezifikation denkbar, damit das Image-Element in diesen zugelassen wird. In diesem Fall wäre der Ansatz des Textur-Streamings wesentlich praxistauglicher.

Für Animationen von Modellen ist die Image Geometry durch die Einschränkungen der HTML5 Video Codecs zum Stand der Arbeit nicht geeignet. Mit neueren Codecs könnte sich dies allerdings ändern. So lohnt sich diese Technik besonders für komplexe Animationen, welche aus Physiksimulationen stammen.

Welche Technik zum Caching der Daten verwendet werden soll, hängt ebenfalls vom jeweiligen Anwendungsfall ab. Der Application Cache eignet sich, um Daten zur Verfügung zu stellen, selbst wenn keine Verbindung zum Internet besteht. Sollen dagegen die Daten dynamisch gespeichert werden, ist die Schnittstelle IndexedDB die passende Wahl. Ebenfalls ist ein Mix aus beiden Techniken möglich.

Die Arbeit zeigt insbesondere, dass keine der betrachteten Formate und Schnittstellen für jeden Anwendungsfall anwendbar ist. Die Komprimierung der Modelle ist zu sehr vom Aufbau des jeweiligen Modells abhängig. Vielmehr muss für den konkreten Fall die richtige Entscheidung getroffen werden. Ein Server könnte diese Aufgabe übernehmen, indem er vorab für das jeweilige Modell feststellt, welches Format die beste Kompressionsrate erzeugt. Anschließend könnte dieser die Verbindungsgeschwindigkeit zum Client ermitteln und demnach das passende Format bereitstellen. Ebenfalls kann der Client nur Modelle anfordern, die er tatsächlich gerade benötigt und dabei für jedes eine Art Prioritätenliste erstellen. Somit würde er für weit entfernte Objekte lediglich die erste Level of Detail Stufe herunterladen. Die Arbeit von Lee et al. [ML10] und Fogel et al. [FCO12] untersuchten bereits diese Möglichkeiten zur Steuerung durch Server und Client und könnten nun mit den hier behandelten Techniken erweitert werden.

Literaturverzeichnis

[Any12] ANYURU, Andreas: *Professional WebGL Programming: Developing 3D Graphics for the Web*. John Wiley and Sons, Ltd., 2012

[Beh11] BEHR, Johannes: X3DOM - Fast content delivery for declarative 3D / W3C: Chair of Declarative 3D Community Group. 2011. – Forschungsbericht

[BH03] BRICENO, Hector ; HOPPE, Hugues: Geometry videos: A new representation for 3D animations. In: *Symposium on Computer Animation 2003* (2003)

[Bid12] BIDELMAN, Eric: *HTML5 Storage*. `http://html5-demos.appspot.com/static/html5storage/index.html`, 2012. – Zugegriffen am 14.09.2012

[Boe00] BOER, Willem: Fast Terrain Rendering Using Geometrical MipMapping. In: *E-mersion Project* (2000)

[Bou12] BOURKE, Paul: *Object Files*. `http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/`, 2012. – Zugegriffen am 14.09.2012

[Cas09] CASTAÑO, Ignacio: *ACMR*. `http://www.ludicon.com/castano/blog/2009/01/acmr-2/`, 2009. – Zugegriffen am 14.09.2012

[CC12] CALDATO, Claudio ; CASTRO, Pablo: *IndexedDB Prototype*. `http://html5labs.interoperabilitybridges.com/html5labs/prototypes/indexeddb/indexeddb/info/`, 2012. – Zugegriffen am 14.09.2012

[Chu12] CHUN, Won: *WebGL Loader*. `http://code.google.com/p/webgl-loader/`, 2012. – Zugegriffen am 14.09.2012

[Cla12] CLANG: *clang: a C language family frontend for LLVM*. `http://clang.llvm.org/`, 2012

[Cro06] CROCKFORD, Douglas: *The application/json Media Type for JavaScript Object Notation (JSON)*. `http://tools.ietf.org/html/rfc4627`, 2006. – Zugegriffen am 14.09.2012

[Cry12a] CRYTEK GMBH: *CryENGINE 3 Free SDK*. `http://freesdk.crydev.net`, 2012

[Cry12b] CRYTEK GMBH: *Using the Resource Compiler*. `http://freesdk.crydev.net/display/SDKDOC3/Using+the+Resource+Compiler`, 2012

[DD11] DELENDIK, Yury ; DAHL, Brendan: *JPEG/DCT data decoder*. `https://github.com/notmasteryet/jpgjs`, 2011. – Zugegriffen am 14.09.2012

[FCO12] FOGEL, Efi ; COHEN-OR, Daniel: A Web Architecture for progressive delivery of 3D content. In: *Unknown Journal* (2012)

[Fel97] FELDSPAR, Antaeus: An Explanation of the Deflate Algorithm. In: *comp.compression* (1997). – Zugegriffen am 14.09.2012

[For06] FORSYTH, Tom: *Linear-Speed Vertex Cache Optimisation*. `http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html`, 2006. – Zugegriffen am 14.09.2012

[Fra12] FRAUNHOFER IGD: *Progressive Loading via BitLODGeometry*. http://examples.x3dom.org/buddha/bitLODGeometry.html, 2012. – Zugegriffen am 14.09.2012

[Gee10a] GEELNARD, Marcus: OpenCTM Developers Manual. (2010). – Zugegriffen am 14.09.2012

[Gee10b] GEELNARD, Marcus: OpenCTM Performance. (2010). – Zugegriffen am 29.09.2012

[GH97] GARLAND, Michael ; HECKBERT, Paul: Surface simplification using quadric error metrics. In: *Siggraph 97 Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997)

[Goo12a] GOOGLE INC.: *Managing HTML5 Offline Storage*. https://developers.google.com/chrome/whitepapers/storage, 2012. – Zugegriffen am 14.09.2012

[Goo12b] GOOGLE INC.: *Source Code Browser Cache Größe Chromium*. http://code.google.com/codesearch#OAMlx_jo-ck/src/net/disk_cache/backend_impl.cc&exact_package=chromium&q=PreferedCacheSize&l=274, 2012. – Zugegriffen am 14.09.2012

[Goo12c] GOOGLE INC.: *WebP - A new image format for the Web*. https://developers.google.com/speed/webp/, 2012. – Zugegriffen am 14.09.2012

[Hei12] HEILMANN, Chris: *There is no simple solution for local storage*. http://hacks.mozilla.org/2012/03/there-is-no-simple-solution-for-local-storage/, 2012. – Zugegriffen am 14.09.2012

[Hof12] HOFFMAN, Billy: *Lose the Wait: HTTP Compression*. http://zoompf.com/2012/02/lose-the-wait-http-compression, 2012. – Zugegriffen am 14.09.2012

[Hop96] HOPPE, Hugues: Progressive meshes. In: *ACM SIGGRAPH 1996 Proceedings,* (1996), 99-108. http://research.microsoft.com/en-us/um/people/hoppe/proj/pm/

[Hop98] HOPPE, Hugues: Efficient implementation of progressive meshes. In: *Computers Graphics, 22(1), 1998, 27-36.* (1998). http://research.microsoft.com/en-us/um/people/hoppe/proj/efficientpm/

[HP03] HOPPE, Hugues ; PRAUN, Emil: Shape Compression using Spherical Geometry. In: *Advances in Multiresolution for Geometric Modelling* (2003)

[Huf52] HUFFMANN, David: A Method for the Construction of Minimum-Redundancy Codes. In: *IRE* (1952)

[IIGS05] ISENBURG, Martin ; IVRISSIMTZIS, Ioannis ; GUMHOLD, Stefan ; SEIDEL, Hans-Peter: Geometry Prediction for High Degree Polygons / Max-Planck-Institut für Informatik. 2005. – Forschungsbericht

[Ind12] INDEPENDENT JPEG GROUP: *Independent JPEG Group*. http://www.ijg.org/. http://www.ijg.org/. Version: 2012

[Lav00] LAVAVEJ, Stephan: *Introduction to PNG*. http://nuwen.net/png.html. http://nuwen.net/png.html. Version: 2000. – Zugegriffen am 14.09.2012

[Law10] LAWRENCE, Eric: Caching Improvements in Internet Explorer 9 / Microsoft Corporation. 2010. – Forschungsbericht. – Zugegriffen am 14.09.2012

[Law12] LAWRENCE, Eric: *Internet Explorer 9 Network Performance Improvements*. http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx. http://blogs.msdn.com/b/ie/archive/2011/03/17/

internet-explorer-9-network-performance-improvements.aspx. Version: 2012. – Zugegriffen am 14.09.2012

[Lim12a] LIM, Kevin: Basic Concepts About IndexedDB / Mozilla. 2012. – Forschungsbericht. – Zugegriffen am 14.09.2012

[Lim12b] LIM, Kevin: *Basic Concepts About the Filesystem API*. https://developer.mozilla.org/en/DOM/File_APIs/Filesystem/Basic_Concepts_About_the_Filesystem_API, 2012. – Zugegriffen am 14.09.2012

[LRC+02] LUEBKE, David ; REDDY, Martin ; COHEN, Jonathan ; VARSHNEY, Amitabh ; WATSON, Benjamin ; HUEBNER, Robert: *Level of Detail for 3D Graphics*. Elsevier Science, 2002 http://research.microsoft.com/en-us/um/people/hoppe/proj/efficientpm/

[LT98] LINDSTROM, Peter ; TURK, Greg: Fast and memory efficient polygonal simplification. In: *VIS 98 Proceedings of the conference on Visualization 98* (1998)

[Mel98] MELAX, Stan: *A Simple, Fast and Effective Polygon Reduction Algorithm*. http://www.melax.com/polychop/, 1998. – Zugegriffen am 14.09.2012

[Mel12] MELLADO, Juan: *js-openctm is a JavaScript library for reading OpenCTM files*. http://code.google.com/p/js-openctm/, 2012. – Letzter Zugriff am 19.09.2012

[MGS09] MUNSHI, Aafftab ; GINSBURG, Dan ; SHREINER, Dave: *OpenGL ES 2.0 Programming Guide*. Addison-Wesley Professional, 2009. – ISBN 0321502795

[ML10] MAGLO, Adrien ; LAVOUE, Guillaume: Remote scientific visualization of progressive 3D meshes with X3D. In: *COSINUS* (2010)

[MSG+11] MEHTA, Nikunj ; SICKING, Jonas ; GRAFF, Eliot ; POPESCU, Andrei ; ORLOW, Jeremy: *Indexed Database API*. http://www.w3.org/TR/IndexedDB/, 2011. – Zugegriffen am 14.09.2012

[Net99] NETWORK WORKING GROUP: Hypertext Transfer Protocol – HTTP/1.1 / Network Working Group. 1999. – Forschungsbericht

[OEC11] OECD: *OECD Communications Outlook 2011*. http://www.oecd.org/sti/broadbandandtelecom/oecdcommunicationsoutlook2011.htm, 2011. – Zugegriffen am 14.09.2012

[Orl10] ORLOW, Jeremy: *Unlimited Storage permission should apply to Local Storage*. http://code.google.com/p/chromium/issues/detail?id=58985, 2010. – Zugegriffen am 14.09.2012

[Pav12] PAVLOV, Igor: *7-Zip*. http://www.7-zip.org/, 2012. – Zugegriffen am 14.09.2012

[Pil12] PILGRIM, Mark: *Dive into HTML5 - Video on the Web*. O'Reilly, 2012 http://diveintohtml5.info/video.html

[PS12] PERRY-SMITH, Lee: Infinite, 3D Head Scan / Infinite-Realities. 2012. – Forschungsbericht. – Zugegriffen am 14.09.2012

[Ran09] RANGANATHAN, Arun: SQL Storage | What Should Be Done? 2009. – Forschungsbericht

[Ran10] RANGANATHAN, Arun: *Beyond HTML5: Database APIs and the Road to IndexedDB*. http://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/, 2010. – Zugegriffen am 14.09.2012

[SB12] SAWICKI, Bartosz ; BARTOSZ, Chaber: 3D Mesh Viewer Using HTML5 Technology. In: *Electrical Review* (2012)

[She12] SHEPHERD, Eric: nsIDOMOfflineResourceList / Mozilla. 2012. – Forschungsbericht. – Zugegriffen am 14.09.2012

[Sol12] SOLEK, Krzysztof: *Blender OpenGL / C header exporter*. http://ksolek.fm.interia.pl/Blender/, 2012. – Zugegriffen am 14.09.2012

[Sou12a] SOUDERS, Steve: *App cache localStorage survey*. http://www.stevesouders.com/blog/2011/09/26/app-cache-localstorage-survey/, 2012. – Zugegriffen am 14.09.2012

[Sou12b] SOUDERS, Steve: *Cache them if you can*. http://www.stevesouders.com/blog/2012/03/22/cache-them-if-you-can/, 2012. – Zugegriffen am 14.09.2012

[Sup10] SUPNIK, Benjamin: *To Strip or Not To Strip*. http://hacksoflife.blogspot.de/2010/01/to-strip-or-not-to-strip.html, 2010. – Zugegriffen am 14.09.2012

[TG98] TOUMA, Costa ; GOTSMAN, Craig: Triangle Mesh Compression / Technion - Israel Institure of Technology. 1998. – Forschungsbericht

[WHA12] WHATWG: *HTML Living Standard - Offline Web applications*. http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html, 2012. – Zugegriffen am 14.09.2012

[Zak12] ZAKAI, Alon: *Emscripten: An LLVM-to-JavaScript Compiler*. https://github.com/kripken/emscripten, 2012. – Zugegriffen am 14.09.2012

[ZL77] ZIV, Jacob ; LEMPEL, Abraham: A Universal Algorithm for Sequential Data Compression. In: *IEEE Transactions on Information Theory, Vol. IT-23, No. 3* (1977). http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

[Zyg12] ZYGOTE MEDIA GROUP: *Zygote Body*. http://www.zygotebody.com/, 2012. – Zugegriffen am 14.09.2012

Abbildungsverzeichnis

3.1 Hinterhofszene aus dem CryENGINE 3 Free SDK 13
3.2 Detailstufen der Hinterhofszene 14
3.3 3D-Scan des Goldenen Reiters 15

4.1 Cache-Optimierung der Indices 21
4.2 Quantisierung des 3D-Scans des Goldenen Reiters 24
4.3 Quantisierung der Hinterhofszene 25

5.1 Half Edge Collapse Visualisierung 30
5.2 Progressive Mesh Beispiel Lee Perry Smith 32
5.3 Progressive Mesh Beispiel Hinterhofszene 33

6.1 Schrittweise Komprimierung Goldener Reiter Diagramm 39
6.2 Schrittweise Komprimierung Hinterhofszene Diagramm 40
6.3 Zerlegung des Modells in einzelne Segmente 43
6.4 Hinterhofszene im OpenCTM MG2 Format 45

7.1 Artefakte bei der Speicherung von Image Geometry mit JPEG und PNG 52
7.2 Nahaufnahme des Index- und Positions-Bildes gespeichert als Image Geometry 52
7.3 Vergleich der progressiven Transfermodi von JPEG und PNG 53
7.4 Beispiel für das Blockmuster von Adam7 53
7.5 Progressive Streaming mit dem progressiven Modus des JPEG Formates am Beispiel des Modells von Lee Perry Smith 54
7.6 Progressive Streaming mit dem Adam7 Modus des PNG Formates am Beispiel des Modells von Lee Perry Smith 55
7.7 Progressive Streaming am Beispiel der Hinterhofszene 57
7.8 Progressive Streaming des Goldenen Reiters mit PNG Adam7 58
7.9 Progressive Streaming mit dem Interlace-Modus des PNG Formates am Beispiel des Modells von Lee Perry Smith 59
7.10 Progressive Streaming des Goldenen Reiters mit PNG 60
7.11 Animation mit Image Geometry mit Videocodec h264 60
7.12 Animation mit Image Geometry durch Austauschen der PNG Bilder 60
7.13 Artefakte bei der Quantisierung der Texturkoordinaten 61

8.1 Verwendete 3D-Modelle im Benchmark 64
8.2 Dekodierungszeit der Hinterhofszene 66
8.3 Dekodierungszeit des Goldenen Reiters 67
8.4 Gesamtzeit für Download und Dekodierung der Hinterhofszene 68
8.5 Gesamtzeit für Download und Dekodierung des Goldenen Reiters 68

9.1 Progressives Streaming einer 2048×2048 JPEG Textur 72

Tabellenverzeichnis

4.1 Quantisierung Vergleich zwischen Short-Variable und Integer-Variable 26

6.1 Schrittweise Komprimierung Goldener Reiter Tabelle 39
6.2 Schrittweise Komprimierung Hinterhofszene Tabelle 40
6.3 Modell Format in OpenCTM Attribute 41
6.4 Modell Format in OpenCTM Index 41

8.1 Benchmark Dateigrößen für die verschiedenen Techniken 65
8.2 Dekodierungszeiten für Chrome 21 66

10.1 Cachegröße 74

B.1 Standardabweichung der Dekodierungszeiten in ms für Chrome 21 97
B.2 Überblick über die Größenverhältnisse der getesteten Modelle 97

Listings

3.1 JSON Modell Format 12

4.1 LZ77 Kodierung 18
4.2 LZ77 Dekodierung 19
4.3 Float und Integer Repräsentation 22
4.4 Quantisierung von Float zu Integer 26

5.1 Datenstruktur 29
5.2 Half Edge Collapse 29
5.3 Vertex Split 31

6.1 OpenCTM MG1 Beispiel 43
6.2 Encoding von UTF-8 46

7.1 Auslesen der Attribute aus der Image Geometry im Vertex Shader 50
7.2 Blockmuster von Adam7 53
7.3 Zerlegen einer 16 bit Integer-Variable zu zwei 8 bit Short-Variablen 61

10.1 Application Cache Beispiel 75
10.2 Anlegen eines asynchronen File Systems 77
10.3 Benutzung des Web Storage 78
10.4 Anlegen einer Datenbank über Web SQL Database 79
10.5 Speichern von binären Daten in Web SQL Datenbank 79
10.6 Verwendung der IndexedDB Schnittstelle 80

A Strukturierung der Vertex Buffer Objects

Die Attribute der Vertices können auf mehrere Arten in VBO verpackt werden. Bei modernen Grafikkarten hat sich das Darstellen der Objekte durch Dreiecke, welche ihre Daten aus VBOs und einer Indexliste erhalten, durchgesetzt. Die Verwendung von Triangle Strips bringt in modernen Anwendungen durch die Ausnutzung von Vertex-Caches oft keinen nennenswerten Geschwindigkeitsanstieg [Sup10]. Auch beim Speicherverbrauch können durch Triangle Strips nur 32 Byte pro wiederverwendeten Vertex (Position, Normale und Texturkoordinate in Float-Variablen) eingespart werden. Für Modelle bei denen sehr lange Triangle Strips möglich sind, konnte so der Speicherverbrauch durch die Wiederverwendung der Vertices beachtlich verringert werden.

Bei indizierten Modellen werden jedoch nur noch zwei Byte für jeden Index (bei der Verwendung von Unsigned Short) eingespart. Folglich wird nur noch $1/8$ der ursprünglichen Einsparungen erreicht. Dies gilt zudem lediglich für Modelle bei denen sehr lange Strips gebildet werden können. Bei den meisten Modellen ist die Einsparung wesentlich geringer. Aus diesen Gründen wird die Verwendung von Triangle Strips in dieser Arbeit nicht weiter betrachtet. Die Attribute können in unterschiedlichen Formaten an die Grafikkarte übertragen werden. Die gängigen Formate sind dabei *Structure of Arrays* und *Array of Structures*.

A.0.1 Structure of Arrays

Die Structure of Arrays [MGS09] speichert alle Daten eines Vertex-Attribut in jeweils einem separaten Array ab. Diese Attribut-Arrays werden über eine Indexliste angesprochen. Für jeden Index wird das Vertex-Attribut aus den jeweiligen Attribut-Array ermittelt und an die Grafikkarte gesendet. Für die Grafikkarte hat dies den Nachteil, dass sie aus mehreren Attribut-Arrays die jeweiligen Attribute, welche zu dem Vertex gehören, ermitteln muss. Allerdings eignet sich diese Struktur gut, falls sich ein Teil der Attribute ständig ändert. Das Attribut, welches geändert werden soll, kann in einem separaten dynamischen VBO gespeichert werden. Deswegen können statische Attribute auf der Grafikkarte bleiben, während dynamische zur Laufzeit angepasst werden.

A.0.2 Array of Structures

Mit dieser Struktur [MGS09] werden alle Vertex-Attribute in einem einzigen gemeinsamen VBO abgespeichert. Dabei werden alle Attribute für jeden Vertex versetzt (interleaved) in das VBO geschrieben:

$x_0, y_0, z_0, nx_0, ny_0, nz_0, uvx_0, uvy_0, x_1, y_1, z_1, nx_1, ny_1, nz_1, uvx_1, uvy_1, ...$

Diese Struktur kommt der Arbeitsweise moderner Grafikkarten entgegen. Die Daten können von der Grafikkarte sequentiell gelesen und somit letztendlich schneller dargestellt werden. Falls jedoch Attribute in diesem kombinierten Buffer geändert werden sollen, muss der gesamte Buffer manipuliert werden, was je nach Anwendungsfall langsamer sein kann, als *Structure of Arrays* zu verwenden.

B Zusatztabellen

Standardabweichung der Dekodierungszeiten in ms für Chrome 21					
	JSON GZIP	UTF-8 GZIP	CTM MG1	CTM MG2	Image Geometry
Hinterhof	2,10	0,62	4,65	6,53	0,69
Goldener Reiter	121,26	47,95	228,31	165,78	18,20
Wasserpumpe	1,57	0,42	4,58	10,00	0,86
Außenbootmotor	3,58	0,57	10,00	4,88	0,86
Fischerhaus	4,52	0,88	8,07	8,24	1,10
Waldruine	5,86	1,55	14,35	9,44	1,05
Leuchtturm innen	7,16	1,17	11,42	9,84	1,05
Traktor	13,19	3,88	26,16	15,79	2,76

Tabelle B.1: Überblick der Standardabweichung der Dekodierungszeiten unter Chrome 21. Durch den Einsatz des JIT entstehen Schwankungen in den Messungen.

Übersicht über die verwendeten Modelle			
	Vertices	OBJ (byte)	JSON (byte)
Hinterhof	4.657	813.637	788.827
Goldener Reiter	435.057	95.090.092	73.351.306
Wasserpumpe	1.218	195.572	195.151
Außenbootmotor	3.391	626.126	593.214
Fischer Haus	6.997	1.157.270	1.196.696
Wald Ruine	12.444	2.274.198	2.177.437
Leuchtturm innen	22.367	3.998.598	3.873.547
Traktor	42.852	8.277.454	7.525.806

Tabelle B.2: Überblick über die Anzahl der Vertices und der Dateigröße im OBJ und JSON Format der getesteten Modelle.

C CD Inhalt

Auf der CD befinden sich alle vorgestellten Prototypen und Benchmarks. Die WebGL Beispiele benötigen einen Server, da die Texturen und Modelle nicht lokal bereit gestellt werden können. Ein einfacher Python Server kann mit dem Kommando *python -m SimpleHTTPServer* gestartet werden. Für einige Beispiele muss im Code der Anwendung auf die Modelle im Ordner „Testdata“ verwiesen werden.

Benchmark Benchmarks aus dem Kapitel Download- und Dekodierungsgeschwindigkeit

- *benchmarkjson.htdocs/* Benchmark für das JSON Format
- *benchmarkctm.htdocs/* Benchmark für das OpenCTM Format
- *benchmarkwebglloader.htdocs/* Benchmark für das WebGL Loader UTF8 Format
- *benchmarkwebglloaderx.htdocs/* Benchmark für das WebGL Loader UTF8x Format
- *benchmarkimagegeometry.htdocs/* Benchmark für das Image Geometry Format

Converters Enthält die Konvertierungsprogramme für die verschiedenen Formate

- *Blender/* Erweiterungen für *Blender* um Image Geometry und JSON zu exportieren

Code Enthält Code zur Optimierung und Schrittweisen Komprimierung von CGF Dateien

- *ForsythCacheOptimization_compressed/* JAVA Code um die schrittweise Komprimierung eines JSON Modells vorzunehmen
- *GenerateApplicationCacheFile/* Erstellt das Application Cache Manifest
  Python Code um das CGF Format schrittweise zu transformieren
  - *CGFToWebGL_cgf_to_json/* Konvertiert CGF zu JSON
  - *CFGToWebGL_json_to_obj_to_utf8/* Konvertiert JSON zu BINÄR und UTF8

Testdata Modelle welche in dem Kapitel Download- und Dekodierungsgeschwindigkeit verwendet wurden

- *1218_water_crane/* Wasserpumpe in allen Formaten
- *3391_outboard_motor/* Außenbootmotor in allen Formaten
- *4657_trash/* Hinterhofszene in allen Formaten
- *6997_fishing_house/* Fischerhaus in allen Formaten
- *12444_forest_ruin/* Waldruine in allen Formaten
- *22367_lighthouse_interior_base/* Leuchtturm (innen) in allen Formaten
- *42852_traktor/* Traktor in allen Formaten
- *435057_goldener_reiter/* Goldener Reiter in allen Formaten
- *obj_models/* Objekte aus dem CryENGINE 3 Free SDK konvertiert zu OBJ

Prototypes Prototypen

- *modelbrowser.htdocs/* Beispiel für einen Browser für Modelle des CryENGINE 3 Free SDK
  Progressive Mesh
  - *createjson.htdocs/* Erstellt PMESH Dateien
  - *readjson.htdocs/* Liest PMESH Dateien
  - *progressivejpg.htdocs/* Lädt und Dekodiert das Model von Lee Perry Smith
  - *pmesh.htdocs/* Schrittweises dekodieren des Model von Lee Perry Smith

  OpenCTM
  - *progressivejpg.htdocs/* Lädt und Dekodiert das Model von Lee Perry Smith

  WebGL Loader
  - *progressivejpg.htdocs/* Lädt und Dekodiert das Model von Lee Perry Smith
  - *webglloader.htdocs/samples/happy/happy.html* Lädt und Dekodiert das Model des Traktors
  - *webglloader.htdocs/sample_textures/viewer/viewer.html* Lädt und Dekodiert das Model des Goldenen Reiters

  Image Geometry
  - *geometryimagegen.htdocs/* Erstellt die Image Geometry Bilder
  - *geometryimagerelease.htdocs/index.html?assets/trash_d/trash_d_joined.json* Lädt die Hinterhofszene
  - *geometryimagerelease.htdocs/index_goldener_reiter.html?assets/goldener_reiter/goldener_reiter.json* Lädt das Modell des Goldenen Reiters
  - *geometryimagerelease.htdocs/index_leeperrysmith.html?assets/boat_motor/boat_motor.json* Lädt das Modell des Motors

  Image Geometry Streaming
  - *geometryimagereleaseprogressive.htdocs/index.html?assets/trash_d/trash_d_joined.json* Streaming der Hinterhofszene
  - *geometryimagereleaseprogressive.htdocs/index_goldener_reiter.html?assets/goldener_reiter/goldener_reiter.json* Streaming des Goldenen Reiters
  - *geometryimagereleaseprogressive.htdocs/index_leeperrysmith.html?assets/leeperrysmith/LeePerrySmith.json* Streaming des Modells von Lee Perry Smith

  Image Geometry Animation
  - *geometryimagerelease.htdocs/animation_video.html?assets/cloth_animation/cloth.json* Beispiel der Kleidersimulation mit einem Video (h264)
  - *geometryimagerelease.htdocs/animation_images.html?assets/cloth_animation/cloth.json* Beispiel der Kleidersimulation mit einer Serie von PNG Bildern
  - *readpngitxt.htdocs/* Liest iTXT Information aus

  Caching
  - *modelbrowserappfilesystemapi.htdocs/* Caching mit der File System API
  - *modelbrowserappcachemozadd.htdocs/* Caching mit Application Cache und MozAdd
  - *modelbrowserappcache.htdocs/* Caching mit Application Cache

  Texture Streaming
  - *jpegworkerwebgl.htdocs/streaming_native.html* Lädt progressive JPEG mit dem Image-Element

- *jpegworkerwebgl.htdocs/streaming_native_canvas.html* Lädt progressive JPEG mit dem Image-Element und Canvas
- *jpegworkerwebgl.htdocs/streaming_native_image.html* Lädt progressive JPEG mit XHR Requests und Canvas
- *jpegworkerwebgl.htdocs/streaming_worker.html* Lädt progressive JPEG mit XHR Requests und Canvas im Web Worker
- *jpegworkerwebgl.htdocs/streaming_worker_custom.html* Lädt progressive JPEG mit XHR Requests und Canvas im Web Worker mit zusätzlicher Kontrolle
- *j2k.htdocs/* Lädt JPEGs mit einer JavaScript Bibliothek
- *progressivejpg.htdocs/* Lädt Texturen auf einigen Beispielmodellen

Verschiedenes

- *screenshotgen.htdocs/* Erstellt Thumbnails
- *multimaterialmesh_ibl.htdocs/* Humvee mit IBL und Level of Details
- *renderengine.htdocs/* Engine Integration (Tag/Nacht Zyklus)

Danksagung

Ich danke allen, die mich bei der Anfertigung dieser Diplomarbeit unterstützt haben, vor allem meinen Betreuern Michael Kopietz, Andreas Stahl und Prof. Gumhold, sowie Tristan Heinig, Claudia Keil, Sabrina Keil, Susanne Krenkel und Christian Krenkel.

Erklärungen zum Urheberrecht

Die in dieser Arbeit genannten Marken-, Firmen- oder Produktnamen sind Marken-, Firmen- oder Produktnamen der jeweiligen Hersteller und/oder Eigentümer. Die 3D-Modelle der Firma Crytek GmbH stammen aus dem CryENGINE 3 Free SDK und unterliegen diesen Lizenzbedingungen.