



**School of
Engineering**

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit in Informatik

Level of Detail Pipeline für 3D- Webapplikationen

Autoren

Marc Berli
Simon Stucki

Hauptbetreuung

Dr. Gerrit Burkert

Datum

21.05.2021

Bitte füllen Sie das Titelblatt aus und berücksichtigen Sie Folgendes:

- ➔ Bitte auf keinen Fall Schriftart und Schriftgrösse ändern. Text soll lediglich überschrieben werden!
- ➔ Bitte pro Tabellenzeile max. 4 Textzeilen!
- Vorlage: Haben Sie die richtige Vorlage gewählt? → Logo Institut/Zentrum
- Titel: Fügen Sie Ihren Studiengang direkt nach dem Wort „Bachelorarbeit“ ein (max. 2 Zeilen).
- Titel der Arbeit: Überschreiben Sie den Lauftext mit dem Titel Ihrer Arbeit (max. 4 Zeilen).
- Autoren: Tragen Sie Ihre Vor- und Nachnamen ein (bitte alphabetisch nach Name).
- Betreuer: Tragen Sie Ihren Betreuer / Ihre Betreuer ein (bitte alphabetisch nach Name).
- Nebenbetreuung: Falls Sie keine Nebenbetreuung haben → bitte ganze Tabellenzeile löschen.
- Industriepartner: Falls Sie keinen Industriepartner haben → bitte ganze Tabellenzeile löschen.
- Externe Betreuung: Falls Sie keine ext. Betreuung haben → bitte ganze Tabellenzeile löschen.
- Datum: Bitte aktuelles Datum eintragen.
- Schluss: Am Schluss löschen Sie bitte den ganzen Beschrieb (grau) und speichern das Dokument als pdf. ab.

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

.....

Unterschriften:

.....

.....

.....

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Die Webplattform erlaubt es, 3D-Visualisierungen einem breiten Publikum zugänglich zu machen. Durch die Vielzahl an Geräten, auf welchen eine solche Applikation verwendet werden kann, ist das Optimieren der Performanz unabdingbar. In dieser Arbeit wird eine Option zur Verbesserung der Laufzeitperformanz aufgezeigt, indem mehrere Detailstufen (*Level of Details – LOD*) eines Modells generiert und je nach Entfernung der virtuellen Kamera angezeigt werden. Zurzeit ist für das Generieren und Einsetzen solcher Artefakte im Web beträchtlicher manueller Aufwand erforderlich.

Verschiedene Ansätze zur Generierung von LOD-Artefakten, sowie ihre Vor- und Nachteile, werden aufgezeigt. Desweiteren werden Algorithmen für die Vereinfachung von polygonalen Modellen analysiert und der gewählte Algorithmus implementiert.

Das Resultat ist ein Tool, welches es ermöglicht, mit geringem manuellen Aufwand die Laufzeitperformanz von 3D-Webapplikation zu verbessern. Es gliedert sich nahtlos in den Arbeitsprozess der Entwickler ein und bietet eine einfache interaktive Konfigurationsmöglichkeit. Mittels eines Benchmarks wird bewiesen, dass der Einsatz von LOD, die Laufzeitperformanz von spezifischen 3D-Webapplikationen verbessern kann.

Abstract

The web platform allows 3D visualizations to be made available to a wide audience. Due to the large number of devices on which such an application can be used, performance optimization is essential. In this work, an option to improve runtime performance is shown by generating multiple levels of details (LOD) of a model and displaying them depending on the distance of the virtual camera. Currently, considerable manual effort is required to generate and deploy such artifacts on the web.

Different approaches for generating LOD artifacts, as well as their advantages and disadvantages, will be demonstrated. Algorithms for simplifying polygonal models are analyzed, the chosen algorithm is implemented, and a tool is designed to integrate it into the workflow.

The result is a tool that seamlessly integrates into the developer's workflow and allows to improve the runtime performance of 3D web applications with little manual effort. By means of a benchmark it is proven that the use of LOD can improve the runtime performance of specific 3D web applications.

Vorwort

Die Autoren der Arbeit sind seit mehreren Jahren in der Webentwicklung tätig. Der rapide Fortschritt in der Webentwicklung ist hauptsächlich dem Engagement der vielen Freiwilligen zu verdanken. Insbesondere für 3D-Visualisierungen sind in den vergangenen Jahren viele neue Möglichkeiten entstanden. Die Autoren teilen die Begeisterung für 3D-Visualisierungen. Aufgrund des noch jungen Alters sind die Tools noch nicht auf dem Level der anderen Plattformen. Diese Arbeit stellt insofern einen Versuch dar, etwas an die Community zurückzugeben und die Arbeit im Web in Bezug auf 3D-Applikationen zu erweitern und zu erleichtern.

Die Performanz in Webapplikationen hinkt derer nativer Applikationen hinterher. Dies bedeutet, dass jede mögliche Optimierung in dieser Hinsicht sehr wertvoll ist. Die sogenannten *Level of Details* sind eine Methode, um die Performanz zu verbessern. Es gibt bereits Ansätze, diese sind jedoch noch nicht ausreichend ausgereift, um in einer produktiven Applikation eingesetzt werden zu können.

Insbesondere die Anzahl der manuellen Arbeitsschritte, welche bei der Entwicklung von 3D-Applikationen notwendig sind, sind den Autoren der Arbeit ein Dorn im Auge. Ziel der Arbeit ist es somit, Prozesse, welche sich in anderen Teilen der Webentwicklung bewährt haben, auch für die Entwicklung von 3D-Applikationen einsetzen zu können und den manuellen Aufwand möglichst gering zu halten.

Ein besonderes Dankeschön geht an unseren Hauptbetreuer Gerrit Burkert, welcher uns während des Verfassens der Arbeit unterstützt hat.

Inhaltsverzeichnis

| | |
|---|-----------|
| 1. Einleitung | 7 |
| 1.1. Kontext | 7 |
| 1.2. Ausgangslage | 7 |
| 1.2.1. 3D-Rendering im Web | 8 |
| 1.2.2. JavaScript Bibliotheken | 9 |
| 1.2.3. Stand der Technik | 10 |
| 1.3. Zielsetzung | 10 |
| 2. Theoretische Grundlagen | 12 |
| 2.1. 3D-Modelle | 12 |
| 2.2. Transformation von Modellen | 16 |
| 2.3. Grafikpipeline | 19 |
| 2.4. Performanzoptimierung | 20 |
| 2.5. Einführung Level of Detail | 23 |
| 2.5.1. Ansätze für LOD Artefakte | 23 |
| 2.5.2. Vergleich Algorithmen | 24 |
| 3. Vorgehen | 26 |
| 3.1. Vergleich LOD Systeme | 26 |
| 3.1.1. Bestehende Systeme | 27 |
| 3.2. Surface Simplification Algorithmus | 28 |
| 3.2.1. Grobablauf | 28 |
| 3.2.2. Fehlermetrik | 28 |
| 3.2.3. Referenzimplementation | 29 |
| 3.3. Pipeline Integration | 29 |
| 3.4. Nutzen LOD | 29 |
| 3.4.1. Aspekte für Nutzen | 29 |
| 3.4.2. Vergleichbare Arbeiten | 31 |
| 3.4.3. Mögliche Ansätze | 31 |
| 4. Resultate | 33 |
| 4.1. LOD Pipeline | 33 |
| 4.1.1. Aktueller Workflow | 33 |
| 4.1.2. lode Pipeline | 33 |
| 4.1.3. lode Ablauf | 33 |
| 4.2. LOD Generierung | 36 |
| 4.2.1. Implementation | 36 |

| | |
|---------------------------------------|-----------|
| 4.2.2. Artefakte | 36 |
| 4.2.3. Fehlermetrik | 37 |
| 4.2.4. Texturen | 37 |
| 4.2.5. Vergleich | 37 |
| 4.3. Benchmark | 39 |
| 4.3.1. Auswertung | 43 |
| 5. Diskussion und Ausblick | 45 |
| 5.1. LOD System | 45 |
| 5.1.1. Allgemeingültigkeit | 45 |
| 5.1.2. Anwendbarkeit | 45 |
| 5.1.3. Abgrenzung | 45 |
| 5.2. Mögliche Erweiterungen | 46 |
| 6. Verzeichnisse | 48 |
| A. Anhang | 56 |
| A.1. Aufgabenstellung | 56 |

1. Einleitung

3D-Visualisierungen werden in vielen Branchen verwendet. Virtual Reality, CAD-Programme oder Computerspiele sind einige bekannte Anwendungsgebiete. Dank leistungsstärkeren Geräten sind in vielen weiteren Bereichen realere Visualisierungen möglich.

Anwendungen können auf spezifische Hardware, wie zum Beispiel die Spielkonsole PlayStation, ausgerichtet sein. Es gibt Unterschiede zwischen den Konsolen, grundsätzlich sind die verschiedenen Geräte jedoch überschaubar. In anderen Anwendungsfällen kann eine Applikation sogar für nur eine dedizierte Hardware entwickelt werden. Somit kann eine Applikation auf die Anforderungen dieser Hardware zugeschnitten werden. Der primäre Nachteil ist dann die Zugänglichkeit der Software, da spezifische Hardware notwendig ist. So ist es einerseits aufwändig die Hardware zu beschaffen, andererseits aber auch eine Kostenfrage. Für gewisse Anwendungsgebiete ist dies ein zweitrangiges Problem. Für hardwareunabhängige Anwendungen eignet sich die Web-Plattform hervorragend. Immer mehr Benutzer haben Zugang zu einem Desktop, Tablet, Mobiltelefon oder ein anderes Gerät, welches Zugang zum Internet bietet [1]. Somit ermöglicht die Web-Plattform, Anwendungen mit weniger Aufwand einem grossen Zielpublikum zugänglich zu machen. Seit einigen Jahren ist es auch möglich, 3D-Visualisierungen im Web zu realisieren.

1.1. Kontext

In der Arbeit wird der Fokus auf die Web-Plattform gelegt. Die meisten Grundlagen sind jedoch auch unabhängig davon zutreffend. Für die Vereinfachung wird im Folgenden der Begriff Web für die Plattform von verschiedenen Web-Technologien genutzt. Dies beinhaltet insbesondere vom *World Wide Web Consortium*, kurz W3C, veröffentlichte Standards.

1.2. Ausgangslage

Dank der Rechenleistung auf modernen Geräten ist es möglich anspruchsvolle 3D-Visualisierungen in Echtzeit auf diversen Geräten anzuwenden. Da diese Applikationen sehr rechenintensiv sind und gerade mobile Geräte in ihrer Rechenleistung beschränkt sind, ist Performance Optimierung im 3D-Rendering unabdinglich. Insbesondere die Komplexität der Modelle hat einen signifikanten Einfluss auf die Leistung. Eine Möglichkeit zur Optimierung ist das Anzeigen von vereinfachten Modellen ab bestimmten Distanzen zum Betrachter. So kann z.B. ein Modell in grosser Distanz vereinfacht (Abbildung 1.1a) dargestellt werden, solange bei genauer Betrachtung mehr Details (Abbildung 1.1b) sichtbar werden.



Abbildung 1.1.: Vergleich vereinfachtes Modell (entfernt) Originalmodell (nah)

In diversen Rendering Engines ¹ gibt es deshalb Möglichkeiten für das Verwenden von sogenannten Level Of Details (LOD) Artefakten. Für Game Engines ² wie *Unreal Engine* oder *Unity* gibt es bewährte Möglichkeiten, um den Einsatz von LOD-Artefakten zu vereinfachen. Zurzeit gibt es in der Webentwicklung keine weitverbreitete Möglichkeit für das Generieren von solchen Artefakten.

1.2.1. 3D-Rendering im Web

Als Basis für 3D-Visualisierungen im Web dient meist das von der Khronos Group entwickelte WebGL, das von allen modernen Browsern unterstützt wird. WebGL ist eine low-level JavaScript API für 3D-Visualisierungen [2]. Alternativ zu WebGL wird zurzeit ein weiterer Standard entwickelt: WebGPU. Dieser ist zur Zeit des Schreibens noch in Entwicklung und wird deshalb nicht weiter berücksichtigt, auch wenn ein grosses Potenzial vorhanden ist [3].

Die Unabhängigkeit der Hardware bedeutet jedoch auch, dass Optimierung der Performance in Webanwendungen unabdinglich ist, um allen Benutzern ein optimales Erlebnis zu ermöglichen. Im Vergleich zu fixen Hardware Anwendungen ist es realistisch, dass eine Webanwendung sowohl auf einem leistungsfähigen Desktop Computer als auch auf einem günstigen Mobilgerät verwendet wird.

Zudem ist WebGL eine junge Technologie und wurde erst 2011 veröffentlicht – verglichen mit dem initialen Release Date von OpenGL ³ welches im Jahre 1992 publiziert wurde [2, 4]. Nicht nur das Alter, sondern auch die Natur der Web-Plattform haben dazu beigetragen, dass WebGL ein langsames Wachstum verspürt hat. Um einen Webstandard

¹Teilprogramm, das zuständig für die Darstellung von Grafiken ist

²Framework, das für den Spielverlauf und dessen Darstellung verantwortlich ist

³Spezifikation einer Programmierschnittstelle zur Entwicklung von 2D- und 3D-Grafikanwendungen

wie WebGL einsetzen zu können, müssen alle grossen Browser die Spezifikation implementieren. Ansonsten kann der grosse Vorteil des Webs – einfache Verteilung an alle Benutzer – nicht in vollem Umfang genutzt werden. So hat zum Beispiel Internet Explorer 10 keinen Support und es wurde erstmals Ende 2013 möglich, im Internet Explorer 11 3D-Anwendungen für ein breites Publikum zu entwickeln.

1.2.2. JavaScript Bibliotheken

Um die Arbeit mit WebGL zu vereinfachen, gibt es verschiedene JavaScript Bibliotheken. Die Bekanntesten werden hier kurz erwähnt. Als Indikator für die Popularität wurden die wöchentlichen Downloads auf npm⁴ verwendet. Wichtig ist hierbei zu betonen, dass dies alleine keinen verlässlichen Indikator darstellt – für eine kurze Übersicht jedoch ausreichend geeignet ist.

Three.js

Ist die wohl weitverbreiteste Bibliothek für 3D-Rendering im Web [5]. Die Community hinter *Three.js* ist aktiv und das offene Produkt wird somit konstant weiterentwickelt. Die Hauptvorteile liegen in der weiten Verbreitung, dem grossen Fokus auf Performanz und einem soliden Set an Basisfeatures. Elemente wie eine Physik-Engine fehlen jedoch und müssen vom Entwickler integriert werden. Zudem wird *Three.js* nicht mit Semantic Versioning⁵ entwickelt, was bedeutet, dass jede neue Version potenziell nicht rückwärtskompatibel Änderungen beinhalten kann. Insbesondere die Erweiterungen für *React*, eine populäre Bibliothek für die Entwicklung von Frontend Applikationen, zeigen den Innovationsdrang [6, 7].

Babylon.js

Ein weiterer Kandidat ist *Babylon.js*, eine offene Bibliothek, welche die Entwicklung von 3D-Applikationen vereinfacht. *Babylon.js* zeichnet sich vor allem durch ein breites *Featureset* aus [8]. *Babylon.js* verfügt zum Beispiel über Erweiterungen für verschiedene Physik-Engines und erleichtert somit den Einsatz von anspruchsvolleren Features. Ausserdem setzt es auf *Semantic Versioning*.

PlayCanvas

PlayCanvas ist eine offene 3D-Engine, welche primär für das Web entwickelt wurde. Als Erweiterung wird zudem ein proprietärer Cloud Service angeboten [9]. Die Community

⁴Node Package Manager

⁵Jede Version wird mit einer aus drei Zahlen bestehenden Versionsbezeichnung (z.B. v1.0.5) versehen. Wobei jede dieser drei Zahlen eine gewisse Gewichtung hat. So wird die vorderste Zahl als *Major* bezeichnet. Diese wird erhöht, wenn grössere Änderungen hinzukommen, die möglicherweise nicht rückwärtskompatibel sind. Die mittlere Zahl – *Minor* genannt – wird erhöht, wenn neue Funktionen hinzugefügt werden, welche rückwärtskompatibel sind. Die hinterste Zahl ist für *Patches*, welche Probleme beheben, ohne die Kompatibilität zu beeinträchtigen.

ist – im Vergleich zur direkten Konkurrenten, wie zum Beispiel *Three.js* – klein. PlayCanvas verfolgt jedoch einen anderen Ansatz und liefert zum Beispiel eine standardmässige Integration für eine Physik-Engine.

Unity

Unity, welches vor allem für die Entwicklung von Mobile Applikationen bekannt ist, bietet seit längerer Zeit die Möglichkeit Projekte für das Web zu exportieren [10]. *Unity* bietet zwar den *Sourcecode* für Teile der *Engine* offen zur Verfügung, die Lizenz verbietet es jedoch den Code weiterzuverwenden [11]. Zudem bietet *Unity* keine zu *Three.js*, *Babylon.js* oder *PlayCanvas* vergleichbare Integration für die Webentwicklung an. *Unity* Projekte sind bezüglich der Integration in Webapplikationen vergleichbar mit Flash Anwendungen. *Unity* wird deshalb in dieser Arbeit nicht weiter berücksichtigt.

1.2.3. Stand der Technik

In anderen Umgebungen gibt es bereits umfangreiche LOD-Systeme für komplexe Anwendungsgebiete. Auf diese wird in der Sektion Unterabschnitt 3.1.1 detaillierter eingegangen. Die erläuterten Bibliotheken bieten Funktionen für das Laden von LOD-Artefakten. Teilweise gibt es die Möglichkeit Vereinfachungen im Browser zu generieren. Das Generieren von LOD-Artefakten zur Laufzeit ist jedoch für Webapplikationen nicht geeignet, da dies signifikante Auswirkungen auf das Laufzeitverhalten hat und insbesondere auf schwächeren Geräten die Applikation zusätzlich verlangsamt. So dauert das Optimieren eines komplexen Modells wie zum Beispiel bei *Babylon.js* demonstriert auch auf rechnungsstarken Geräten mehrere Sekunden [12].

Des Weiteren sind die Systeme nicht kompatibel mit anderen Bibliotheken. So kann die automatische LOD-Lösung von *Babylon.js* nicht in *Three.js* verwendet werden, obwohl die Problemstellung dieselbe ist.

1.3. Zielsetzung

Das Ziel der Arbeit ist es, ein Tool zu entwickeln, das den Umgang mit LOD-Artefakten im Web vereinfacht. Hierfür muss ein Algorithmus entwickelt werden, der es erlaubt, Modelle drastisch zu vereinfachen, ohne die grobe geometrische Form zu verlieren. Im Anschluss muss das Tool zur Verfügung gestellt werden, sodass es in der Praxis eingesetzt werden kann. Hierbei ist es wichtig, dass die Artefakte nicht innerhalb des Browsers generiert werden. Zudem soll der Einsatz des Tools einen möglichst geringen Zusatzaufwand bedeuten und für ein breites Spektrum an Modellen angewendet werden können. Ersteres wird dadurch ermöglicht, in dem ein Konfigurationstool bereitgestellt wird, mit dem man Feinjustierungen an den generierten Modellen vornehmen kann. Des Weiteren soll das Tool soweit erweiterbar sein, dass es für verschiedene Bibliotheken eingesetzt werden kann, ohne den Kern neu entwickeln zu müssen. Am Schluss muss zudem der Beleg erbracht werden, dass das Laufzeitverhalten der Applikation mit LOD-Artefakten

verbessert werden kann. Hierfür muss ein geeignetes Werkzeug zur Messung der wichtigsten Faktoren der Laufzeit definiert werden.

2. Theoretische Grundlagen

Im folgenden Abschnitt werden die verschiedenen Grundlagen für die Entwicklung eines LOD-Systems kurz erläutert. Es werden zuerst die Grundlagen der 3D-Modellierungen und die verschiedenen Dateitypen erläutert. Danach wird auf die Theorie der Transformationen und die Grafik-Pipeline eingegangen. Zum Schluss werden noch einige Optionen zur Performanzoptimierung erläutert.

2.1. 3D-Modelle

Ein Modell stellt ein Objekt aus der realen Welt vereinfacht dar. 3D-Modelle können vereinfacht als Gruppe von Punkten definiert werden. Um im dreidimensionalen Raum Objekte visualisieren zu können sind mindestens drei Punkte notwendig. Punkte von 3D-Modellen werden im folgenden als Vertex (Eckpunkte) bezeichnet und können als Vektor definiert werden. So können wir einen Vertex am Ursprung eines Koordinatensystems definieren als:

$$V = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Eine Sammlung von drei *Vertices* bildet ein *Triangle* (Dreieck). Hiermit kann ein *Triangle* generell wie folgt definiert werden:

$$T = \begin{bmatrix} V \\ V \\ V \end{bmatrix}$$

Um komplexere Formen wie *Quads* (Vierecke) zu bilden, werden jeweils mehrere *Triangle* kombiniert. Für eine Sammlung von Punkten wird generell der Term Polygon verwendet. Ein Modell besteht aus einer beliebigen Anzahl Polygone. Grundsätzlich gilt somit, dass ein beliebiger Polygon durch mehrere *Triangles* repräsentiert werden kann:

$$P = \begin{bmatrix} V \\ V \\ V \\ V \end{bmatrix} = \begin{bmatrix} T & T \end{bmatrix} = \begin{bmatrix} V & V \\ V & V \\ V & V \end{bmatrix}$$

Abbildung 2.1 zeigt die Vereinfachung eines *Quads* in zwei *Triangle*.



(a) Polygon

(b) Polygon als Sammlung von *Triangles*

Abbildung 2.1.: Vereinfachung von Polygone als Sammlung von *Triangles*

Die Verbindung zwischen zwei *Vertices* ist eine sogenannte *Edge* (Kante). Verbindet man die Punkte eines Polygons und füllt die Fläche, ergibt sich schlussendlich ein *Face* (Fläche).

Weitere Attribute

Neben den geometrischen Attributen verfügt ein Modell ebenfalls über visuelle Attribute. So wird für jeden Vertex ein *Normal* definiert. Ein *Normal* ist ein Vektor der im einfachsten Fall senkrecht zu den zwei an diesem Vertex verbundenen *Edges* verläuft. In der Geometrie ist dies auch bekannt als Normale. *Normals* werden häufig für das Berechnen von Reflexionen verwendet. *Normals* werden auch für gewisse Performanz Optimierungen eingesetzt, dazu mehr in Absatz 2.4.

Um die Oberfläche von Modellen zu definieren, wird häufig Texture Mapping ¹ durchgeführt. Hierfür sind zusätzliche Informationen für ein Modell notwendig. In der Praxis finden weitere Methoden Anwendung, auf diese wird jedoch hier nicht weiter eingegangen.

Abgrenzung

Für diese Arbeit sind nur polygonale Modelle relevant. Alternativen wie zum Beispiel *Point Clouds*, welche das Resultat von 3D-Scans sind, werden nicht weiter erläutert, da sie selten im Web eingesetzt werden. Auch andere Methoden wie *NURBS*, welche Modelle mithilfe von mathematisch definierten Flächen modelliert, werden deshalb aussen vor gelassen.

¹Verfahren, um mittels 2D-Bildern 3D-Objekte zu gestalten

Formate

Um ein 3D-Modell in einer Anwendung zu verwenden, muss ein entsprechendes Format verwendet werden. Hierfür steht eine Vielzahl von Optionen zur Auswahl. Aufgrund der Menge wird hier jedoch nur oberflächlich auf die bekanntesten Formate eingegangen.

OBJ Wavefront OBJ ist ein offenes Dateiformat das von Wavefront Technologies 1989 entwickelt wurde. Das Format ist jedoch speichertechnisch ineffizient und verfügt zudem nur über ein limitiertes Feature Set. Des Weiteren gibt es keine zentrale Instanz, welche eine Spezifikation liefert und Informationen sind deshalb schwerer zu finden. [13]

FBX FBX ist ein proprietäres Format, welches von Autodesk verwaltet wird. Das Verwenden von FBX Daten ist jedoch offiziell nur mit einer C++ FBX SDK möglich, welche für das Web nicht geeignet ist. Aufgrund der proprietären Natur gibt es keine Bestrebungen das Format offener zu gestalten.

glTF Seit 2015 gibt es ein offenes, modernes sowie auf optimale Speichernutzung fokussiertes Format: glTF. Dieses Format wird von der Khronos Gruppe entwickelt [14]. In dieser Arbeit wird es als Austauschformat verwendet, deshalb wird im folgenden genauer auf das Format eingegangen.

Die meisten 3D-Grafikprogramme wie Blender (.blend) verwenden ihre eigenen Dateiformate. Diese Dateiformate sind jedoch für den Einsatz in einer Applikation ungeeignet, da sie insbesondere nicht für die Laufzeitperformanz optimiert sind. Deswegen brauchte es für jedes Eingangsformat für jedes Ausgangsformat einen *converter*, wie in Abbildung 2.2 ersichtlich.

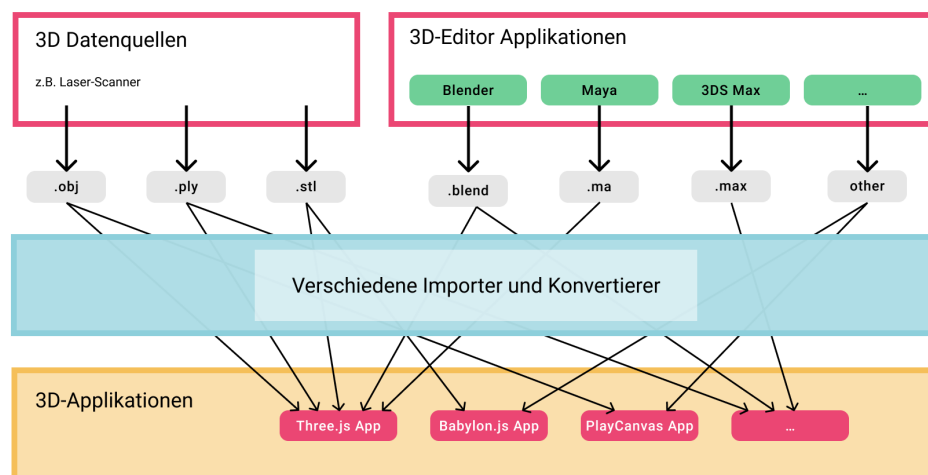


Abbildung 2.2.: Konvertierungspipeline ohne glTF [15]

Durch die immer breiter werdende Nachfrage nach 3D-Applikationen wurde ein Format benötigt, dass zum einen Applikationsunabhängig verwendet und zum anderen performant im Web eingesetzt werden kann. Durch diese Abstraktion kann eine Vielzahl *converter* vermieden werden und nur wenige, sehr spezifische Dateiformate benötigen noch einen einzigen *converter*. Diese neue Pipeline ist in Abbildung 2.3 dargestellt. [15]

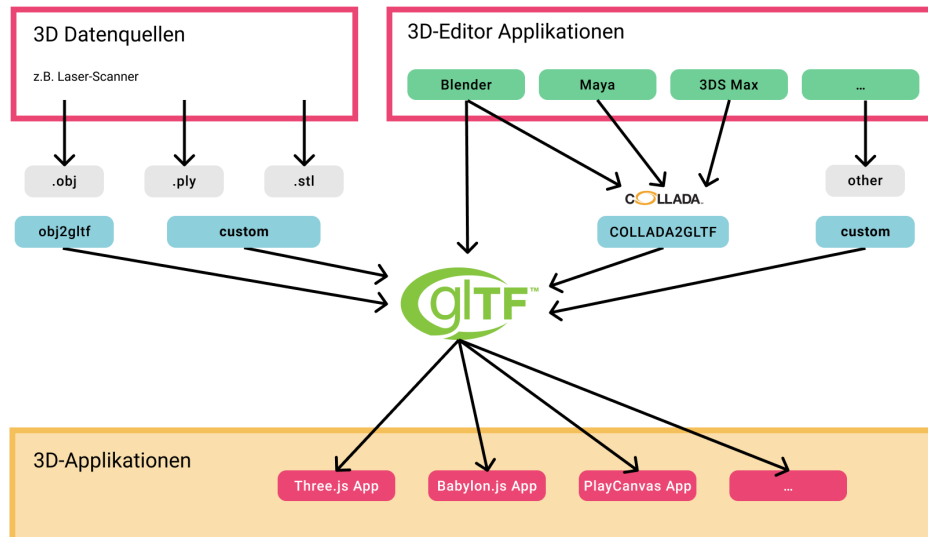


Abbildung 2.3.: Konvertierungspipeline mit glTF [15]

Die Basisstruktur von glTF basiert auf JSON. Darin ist die komplette Szenerie des Modells beschrieben und in Abbildung 2.4 aufgezeigt [16].

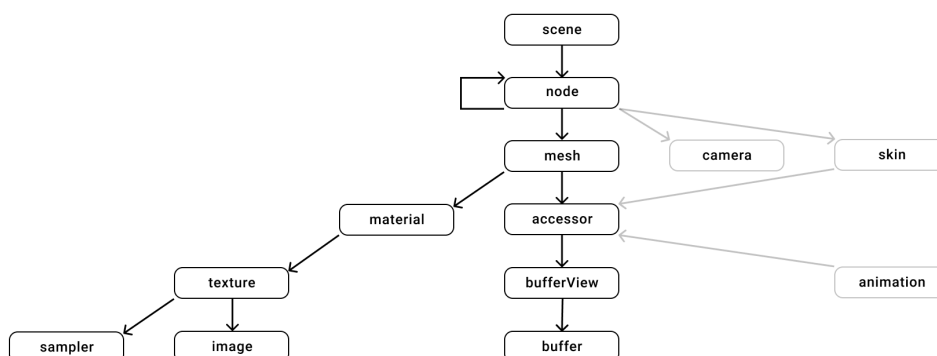


Abbildung 2.4.: glTF Datenstruktur [16]

Die wichtigsten Elemente dieser Struktur kurz erläutert:

- **Scene:** Einstiegspunkt der Szenerie und verweist auf alle Top-Level *nodes*.
- **Node:** Kann eine Transformation (Rotation, Translation oder Skalierung) beinhalten oder eine Kamera, *Skin*, Animation, ein *Mesh* oder *Childnodes* referenzieren.
- **Mesh:** Beschreibt ein geometrisches Objekt und verweist auf *accessor*, welcher die effektiven geometrischen Daten beinhaltet und auf *material*, das beschreibt, wie das Objekt beim Rendern aussehen soll.
- **Accessor:** Verweis auf die Binärdaten (*BufferView*), welche die effektiven Eigenschaften für *meshes*, *skins* und *animations* kompakt beinhaltet.
- **BufferView:** Definiert eine Ansicht (Länge, Ort, Typ) auf einen *Buffer*.
- **Buffer:** Verweist auf einen Block von Binärdaten, welchen die effektiven Daten des 3D-Modells platzsparend beinhaltet.
- **Material:** Definiert die visuellen Attribute eines *Mesh*.
- **Texture:** Definiert wie ein *Image* auf ein *Material* projiziert wird. Ein *Sampler* definiert wie das Bild platziert werden soll.
- Weitere Elemente, welche im Rahmen dieser Arbeit nicht relevant sind und nicht im Detail betrachtet werden, sind: *camera*, *skin* und *animation*.

2.2. Transformation von Modellen

Um ein Modell vereinfachen zu können, muss es verändert werden. Diese Veränderungen können in Basis Operationen vereinfacht erläutert werden.

Im Folgenden werden Transformationen mithilfe einer 2D Visualisierung erläutert. Das Modell kann jedoch ebenfalls im dreidimensionalen Raum sein – die Funktionsweise bleibt identisch.

Für die folgenden Beispiele wird jeweils das Modell aus Abbildung 2.5 verwendet.

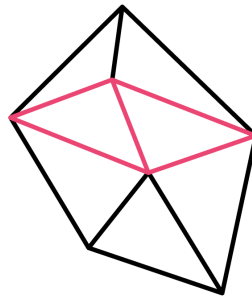
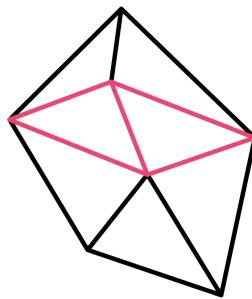


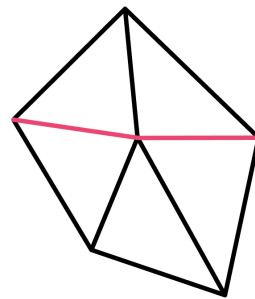
Abbildung 2.5.: Modell Basis

Edge Collapse

Hierbei werden zwei nebeneinanderliegende *Vertices* kombiniert. Durch diese Operation kann ein Vertex entfernt werden. In Abbildung 2.6 kann die Anzahl *Triangles* um zwei verringert werden. Beim *Edge Collapse* wird ein neuer Vertex definiert und zwei bestehende entfernt. Die Anzahl entfernter *Triangles* ist abhängig von der Situation. Unter Umständen kann so auch nur ein einzelner *Triangle* entfernt werden. Die Umkehrfunktion nennt man Vertex Split.



(a) Modell Ausgangslage



(b) *Edge Collapse*

Abbildung 2.6.: Transformation mittels *Edge Collapse*

Halfedge Collapse

Hierbei wird ein Vertex direkt entfernt und alle *Edges* auf einen danebenliegenden Vertex zusammengelegt. Bei dieser Operation muss kein neuer Vertex definiert werden, sondern

ein bereits bestehender Vertex kann wiederverwendet werden. Wie in Abbildung 2.7 ersichtlich, können auch in diesem Fall zwei *Triangles* entfernt werden.



(a) Modell Ausgangslage



(b) *Halfedge Collapse*

Abbildung 2.7.: Transformation mittels *Halfedge Collapse*

Vertex Removal

Hierbei wird ein Vertex entfernt und das Resultat neu trianguliert. Triangulation ist ein Verfahren, um ein Polygon in *Triangles* aufzuteilen. In Abbildung 2.8a wird der zentrale Vertex entfernt. Anschliessend werden alle *Triangles* an diesem Punkt entfernt. Das dabei entstehende Loch wird neu mit *Triangles* gefüllt. Der neu triangulierte Polygon ist in Abbildung 2.8b ersichtlich.



(a) Modell Ausgangslage



(b) Vertex Removal

Abbildung 2.8.: Transformation mittels Vertex Removal

2.3. Grafikpipeline

Die Grafikpipeline ist zuständig um eine definierte Szene auf einem Ausgabegerät zu visualisieren. Die verschiedenen Schritte werden im folgenden Abschnitt kurz erläutert. Es wird dabei ein simples 3D-Modell auf einem 2D-Display, wie zum Beispiel einem Monitor, visualisiert. Die verschiedenen Schritte sind in Abbildung 2.9 aufgezeigt. Die verschiedenen Schritte werden idealerweise 60 Mal pro Sekunde wiederholt. Dies findet in der sogenannten *Render Loop* statt.

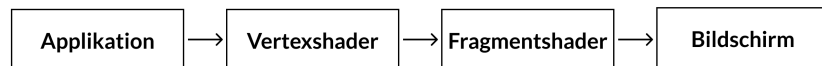


Abbildung 2.9.: Schritte einer Realtime Rendering Pipeline

Applikation

Im ersten Schritt wird die Szenerie aufbereitet, Modelle in den Arbeitsspeicher geladen und Positionen und Rotationen der Modelle definiert. Diese Schritte werden auf der CPU durchgeführt. Ein Beispiel für das Aufsetzen eines einfachen *Triangles* bestehend aus drei *Vertices* ist in Abbildung 2.10 ersichtlich.

```
const vertices = new Float32Array([
  -1, 1, 0, // vertex 0
  -1, -1, 0, // vertex 1
  1, -1, 0 // vertex 2
]);

const indices = new Uint16Array(
  [0, 1, 2] // triangle 0
);
```

Abbildung 2.10.: Definition eines Triangles

Vertexshader

Anschliessend werden die Daten an die GPU gesendet. Der erste Schritt ist der sogenannte Vertexshader. Auftrag des Vertexshaders ist es, die *Vertices* zu transformieren. Die Modelle werden ausgerichtet, die Beleuchtung bestimmt und die 2D-Projektion vorgenommen. Ein Beispiel für einen simplen Vertexhsader ist in Abbildung 2.11 ersichtlich.

```
attribute vec3 coordinates;

void main(void) {
    gl_Position = vec4(coordinates, 1.0);
}
```

Abbildung 2.11.: Vertexshader für die Definition eines Punktes

Fragmentsshader

Durch die Rasterisierung werden kontinuierliche Objekte zu diskreten Fragmenten verarbeitet. Der Fragmentshader generiert die Farbdefinitionen für die einzelnen diskreten Fragmente. Dies können einfache Farbwerte sein, es ist jedoch auch möglich Texturen und dergleichen auf Fragmente abzubilden. Ein einfaches Beispiel für einen Fragmentshader ist in Abbildung 2.12 ersichtlich.

```
void main(void) {
    gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Abbildung 2.12.: Fragmentshader für das definieren von Farbwerten

Bildschirm

Am Schluss wird für jedes vom Fragmentshader generierte Fragment ein Sichtbarkeits-test durchgeführt. So werden nur die Fragmente, welche von keinem anderen Fragment überdeckt werden, angezeigt. Das generierte Bild wird im Anschluss skaliert und auf dem Bildschirm dargestellt.

2.4. Performanzoptimierung

Visualisierungen können zu komplex werden, um jederzeit performant und interaktiv gerendert zu werden. Insbesondere wenn viele Objekte gleichzeitig sichtbar sind, lohnt es sich Performanzoptimierungen durchzuführen. Im Idealfall geschieht dies jedoch, ohne dass der Anwender dies bemerkt.

In diesem Abschnitt werden mögliche Ansätze erklärt, welche helfen sollen, die Render-Performanz zu erhöhen. Diese Arbeit konzentriert sich jedoch auf den Ansatz von Level of Detail; die anderen Ansätze werden nur kurz erläutert. Auf Level of Detail wird in Abschnitt 2.5 detailliert eingegangen.

Frustum culling

Ein Frustum ist eine geometrische Form, die einer Pyramide ähnelt, welcher die Spitze abgeschnitten wurde. Das Kamera-Frustum bezeichnet den Raum, welcher von der Ka-

mera aufgezeichnet wird. Dieser Bereich startet eine gewisse Distanz von der Kamera entfernt und endet weit hinten wieder, da nicht bis in die Unendlichkeit Objekte sichtbar sind. Wie in Abbildung 2.13 zusehen ist, beginnt das Kamera-Frustum in der Nähe der Kamera und dehnt sich ein wenig aus bis es gegen Ende des Spektrums beschränkt wird.

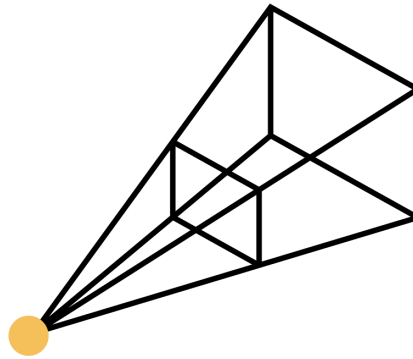


Abbildung 2.13.: Kamera Frustum

Polygone, welche nicht im Kamera Frustum enthalten sind, werden bei dieser Methode, wie in Abbildung 2.14 gezeigt, nicht weiter prozessiert. Dies reduziert die Anzahl Polygone drastisch.

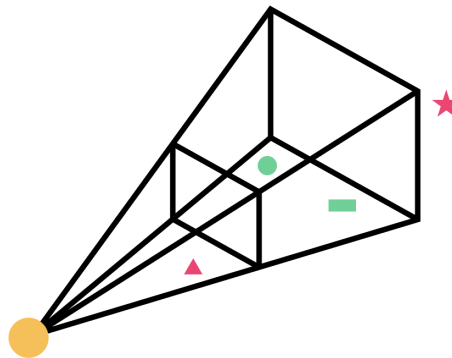


Abbildung 2.14.: Frustum culling visualisiert. Rote Elemente werden nicht prozessiert.

Occlusion culling

Polygone bzw. Objekte, welche komplett von anderen Objekten überdeckt werden, werden bei dieser Variante nicht prozessiert. Dieser Prozess analysiert die Szene mittels

einer virtuellen Kamera und merkt sich potenziell nicht sichtbare Objekte. Diese Daten werden dann zur Laufzeit von der Kamera verwendet, um zu bestimmen, ob ein Objekt prozessiert werden soll, oder nicht. Damit wird die Anzahl *Draw Calls*² reduziert und somit die Performanz erhöht.

Backface culling

Bei dieser Methode wird berechnet welche Polygone zur Kamera orientiert sind. Alle Polygone, welche in die andere Richtung zeigen werden nicht gezeichnet. Dies ist nicht immer gewünscht, für die meisten Anwendungen ist diese Optimierung jedoch aktiviert. Als Grundlage für die Berechnung werden die *Normals* der *Vertices* berücksichtigt.

Imposters

Vergleichbar zu LOD-Artefakten sind sogenannte *Imposters*. Dabei wird ein *Quad* mit verschiedenen Texturen verwendet. Für jede mögliche Position des Modells wird eine Textur generiert. Das *Quad* wird in Richtung Kamera ausgerichtet und abhängig von der Rotation des Modells die geeignete Textur verwendet. Geometrisch kann ein Modell so auf ein absolutes Minimum reduziert werden.

Weitere Ansätze

Weitere Ansätze zur Optimierung der Performanz für sehr spezifische Anwendungsfälle werden kurz erwähnt, jedoch nicht weiter erörtert da sie für viele Anwendungen nicht praktikabel sind.

Parallel rendering Auch bekannt unter Distributed Rendering ist der Einsatz von Techniken aus dem Parallel Programming in Visualisierungsanwendungen. So kann zum Beispiel die Bildfläche in Regionen aufgeteilt werden, welche alle individuell und gleichzeitig prozessiert werden. Es gibt aber auch andere Ansätze, wobei die *Triangles* anderweitig aufgeteilt und prozessiert werden und dann zum Schluss an den *Rasterizer* gesendet werden.

Image-based rendering In gewissen Fällen kann das Modellieren übersprungen werden. So kann anhand von Bildmaterial eine 3D-Illusion erzeugt werden. Da 2D-Bilder aber keine Tiefeninformationen darstellen können im 3-dimensionalen Raum, kann dies zu seltsamen Effekten führen, wenn sich die Kamera bewegt und kann deshalb nur in sehr spezifischen Anwendungsfällen eingesetzt werden.

²Ein Aufruf auf die GPU, um ein Objekt zu zeichnen. Ein *Draw Call* wird pro Material pro *Mesh* benötigt. Es ist besser, weniger, dafür umfangreiche *Draw Calls* abzusetzen, als viele Kleinere. Dies kommt daher, dass die GPU extrem schnell ist und sich die Unterbrechung durch einzelne Aufrufe viel negativer auswirkt.

2.5. Einführung Level of Detail

Als Level Of Detail (LOD) werden die verschiedenen Detailstufen bei der virtuellen Darstellung bezeichnet. Dies wird verwendet, um die Geschwindigkeit von Anwendungen zu steigern, indem Objekte im Nahbereich detailliert angezeigt werden; wohingegen Elemente im Fernbereich deutlich vereinfacht dargestellt werden.

Ein LOD-Algorithmus hat das Ziel für ein gegebenes Modell eine vereinfachte Darstellung zu finden, welche das Original ausreichend annähert. Um diese Approximationen zu generieren, kann eine Vielzahl von Algorithmen verwendet werden. Es gibt verschiedene Ansätze zur Generierung von LODs, welche in Unterabschnitt 2.5.2 im Detail erläutert werden.

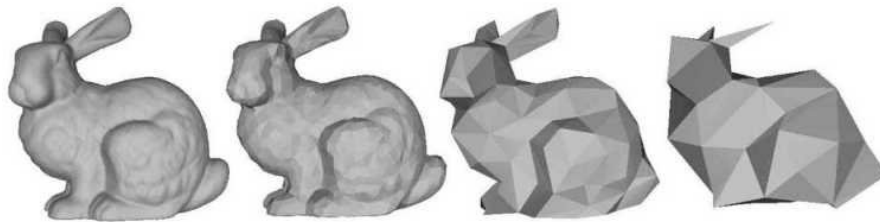


Abbildung 2.15.: Level Of Detail Visualisierung vier Hasen

Wie in der Abbildung 2.15 zu erkennen ist, wird von links nach rechts der Detailgrad und somit die Komplexität des Objektes reduziert. Sind es im Bild ganz links noch 69'451 Polygone, wird es bereits im ersten Schritt auf 2'502 Polygone reduziert. Dies ist eine enorme Reduktion von ca. 96.5%. Im dritten Schritt wird die Anzahl Polygone wiederum um ca. 90% auf 251 reduziert. Schlussendlich hat das letzte Objekt noch 76 Polygone was knapp 0.1% der ursprünglichen Anzahl entspricht.

2.5.1. Ansätze für LOD Artefakte

Es gibt verschiedene Ansätze, 3D-Modelle mittels LOD zu vereinfachen. In diesem Abschnitt werden einige davon detaillierter erläutert so wie ihre Vor- und Nachteile aufgezeigt.

Diskrete LOD (DL0D)

Bei diskreten LOD werden für ein detailliertes Modell mehrere weniger detaillierte Modelle erstellt. Abhängig von der Distanz zum Betrachter wird das optimale Modell gewählt. Es sind hierfür nur minimale Anpassungen am Scene Graph ³ notwendig da ausschliesslich ganze Modelle ausgetauscht werden. Ein Nachteil sind jedoch merkbare harte Grenzen. Der Benutzer stellt beim umhergehen in der Szene fest, wenn das Modell mit einer einfacheren Version ausgetauscht wird. Es ist zudem häufig nicht möglich

³Objektorientierte Datenstruktur zur Beschreibung von 2D- oder 3D-Szenarien

grosse Modelle sinnvoll zu vereinfachen. So ist der Ansatz für Terrain ungeeignet. Ähnlicherweise ist es auch nicht möglich viele sehr kleine Modelle zu kombinieren da jedes Modell unabhängig ist.

Kontinuierliche LOD (CLOD)

Im Gegensatz zu DLOD wird bei CLOD vereinfachende Veränderungen an einem Modell gespeichert.

Der Hauptunterschied zu DLOD besteht in den weichen Grenzen, so ist es signifikant weniger auffällig da zwischen den verschiedenen Auflösungen interpoliert werden kann. Der Hauptnachteil ist jedoch, dass diese Interpolation insbesondere Auswirkungen auf die Laufzeit Performanz der Applikation hat. Das Problem des Clusterings ist auch für diese Variante nicht gelöst.

Hierarchische LOD (HLOD)

Bei HLOD werden mehrere Objekte in einen Cluster gruppiert. Diese Methode erlaubt es somit zum Beispiel Terrains optimal abzubilden. So werden Ausschnitte, welche nahe beim Benutzer sind mit hoher Auflösung dargestellt während weiter entfernte Teile der Umgebung mit weniger Details ausgestattet sind. Zudem ist es möglich viele kleinere Objekte bei entsprechenden Distanzen zu kombinieren. Der Hauptnachteil liegt darin, dass hierfür signifikante Anpassungen am *Scene Graph* notwendig sind und für den Entwickler ein spürbarer Unterschied entstehen kann.

2.5.2. Vergleich Algorithmen

Das Ziel des Algorithmus ist es, eine gegebene geometrische Struktur mit möglichst wenig *Triangles* so genau wie möglich zu approximieren. Diese Art von Problem ist in der Literatur unter anderem als *Surface Simplification*, polygonale Simplifizierung, geometrische Simplifizierung oder Mesh Reduzierung bekannt. Grundsätzlich kann man die verschiedenen Algorithmen in Kategorien einteilen. Im folgenden Abschnitt wird auf die Grundidee der verschiedenen Kategorien eingegangen.

Vertex Dezimierung

Algorithmen dieser Kategorie entfernen jeweils einen Vertex und triangulieren das entstehende Loch neu. Eine weitere Möglichkeit ist das *Vertex Clustering*, hierbei wird ein Modell in verschiedene Gitterzellen aufgeteilt. Die Punkte innerhalb einer Gitterzelle werden dann in einen einzigen Punkt zusammengelegt. Die visuelle Annäherung dieser Algorithmen bei drastischen Vereinfachungen stellt jedoch häufig ein Problem dar für polygonale Modelle. Sogenannte *Point Clouds* ⁴ (in Abbildung 2.16 exemplarisch zu sehen), wie sie bei 3D-Scannern eingesetzt werden, können mit solchen Algorithmen schnell vereinfacht werden.

⁴Vertices im dreidimensionalen Raum ohne zugehörige *Triangle*

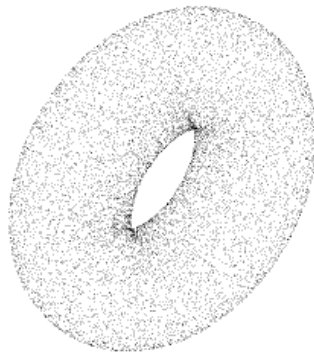


Abbildung 2.16.: Beispiel *Point-Cloud* [17]

Edge Collapse

Hierbei werden iterativ *Edges* entfernt und die beiden *Vertices* zu einem Punkt zusammengelegt. Die verschiedenen Algorithmen unterscheiden sich primär in der Selektion der *Edges*. Grundsätzlich wird hierfür eine Heuristik verwendet um den Fehler quantifizieren zu können. Anschliessend werden iterativ die *Edges* entfernt, welche zu einem minimalen Fehler führen. Beim Zusammenführen von *Edges* sind abhängig vom gewählten LOD-System unterschiedliche Strategien gefordert. Bei diskreten LOD-Systemen soll nach dem Zusammenführen ein optimaler neuer Punkt gefunden werden, bei kontinuierlichen LOD-Systemen wird häufig einer der beiden Punkte gewählt um Speicherplatz einsparen zu können.

Triangle Collapse

Bei dieser Variante werden jeweils ganze Flächen entfernt und die umliegenden *Triangles* zusammengelegt. Eine Möglichkeit ist der von Hamann definierte Algorithmus zur Vereinfachung von triangulierten Flächen [18]. Dabei wird die Krümmung eines *Triangles* definiert und iterativ derjenige *Triangle* mit der geringsten Krümmung entfernt. An die Stelle der drei *Vertices* tritt ein neuer *Vertex* und die umliegenden Flächen werden neu trianguliert.

3. Vorgehen

3.1. Vergleich LOD Systeme

Die verschiedenen LOD Systeme sowie ihre Vor- und Nachteile werden in Unterabschnitt 2.5.1 erläutert. In diesem Schritt wird erläutert, welche Art LOD-System in dieser Arbeit entwickelt werden soll.

Die verschiedenen Ansätze unterscheiden sich in einigen Kriterien. Für den Einsatz in einer Webanwendungen ist die Gewichtung der Kriterien anders als für Desktopanwendungen, Konsolenspiele oder dergleichen. So hat die Dateigrösse der Modelle bei Webanwendungen einen grösseren Einfluss auf die gefühlte Performanz beim Endanwender. Insofern ist die Auswirkung der gewählten Strategie auf die Downloadgrösse ein wichtiges Kriterium. Für diskrete LOD muss für jedes Level ein separates Modell geladen werden. Dies kann die Downloadgrösse gegebenenfalls merkbar erhöhen. Bei kontinuierlichen LOD-Systemen muss das Modell die Unterschiede zwischen verschiedenen Levels persistieren. Mit einem für kontinuierliche LOD ausgelegten Algorithmus kann der benötigte Speicher reduziert werden, indem beim *Edge Collapse* nicht auf einen neuen optimalen *Vertex* reduziert wird, sondern von den beiden bestehenden *Vertices* der passendere gewählt wird. Trotzdem entsteht für jeden *Edge Collapse* ein zusätzlicher Speicheraufwand. Insbesondere unter Berücksichtigung der Tatsache, dass Laufzeitperformanz essenziell für Webanwendungen ist, wurde entschieden nicht auf kontinuierliche LOD-Systeme zu setzen. Zudem wurde entschieden kein hierarchisches LOD-System zu entwickeln, um die Integration des Tools für die Entwickler einer Webanwendung möglichst einfach zu gestalten. Das diskrete LOD-System ist für Grafiker und Entwickler am einfachsten zu verwenden und deckt somit auch die breiteste Benutzerbasis und die meisten Anwendungsfälle ab. Aufgrund dessen, wird im Rahmen dieser Arbeit ein Algorithmus dieses Systems entwickelt und in den Entwicklungsprozess integriert. Eine Übersicht über die verschiedenen Kriterien kann in der Tabelle 3.1 gefunden werden.

| Technische Auswirkungen | DLOD | CLOD | HLOD |
|--|---------|---------|-------------|
| Auswirkung auf Downloadgrösse | mittel | mittel | mittel |
| Auswirkung auf Laufzeitverhalten | klein | mittel | mittel |
| Auswirkung auf <i>Scene Graph</i> | lokal | lokal | global |
| "Visual Pop" vermeidbar | nein | ja | ja |
| Integration in bestehende Arbeitsabläufe | einfach | einfach | umständlich |
| Möglichkeiten | | | |
| Drastische Reduktion von Polygonen | ja | ja | ja |
| Clustering möglich | nein | nein | ja |

Tabelle 3.1.: Übersicht Merkmale der verschiedenen LOD Systeme

Visual Pop bezeichnet das merkbare Austauschen zweier LOD Artefakte. Im optimalen Fall ist der Austausch zweier Artefakte nicht bemerkbar.

3.1.1. Bestehende Systeme

Unity sowie *Unreal* verfügen über umfangreiche LOD-Systeme. Eine Analyse dieser Systeme bildet somit die Grundlage für die Entwicklung eines neuen Systems.

Unreal

Unreal verfügt sowohl über ein DLOD-System als auch über ein HLOD-System. Das HLOD-Tool zeichnet sich insbesondere dadurch aus, dass es die verschiedenen HLOD-Artefakte automatisch generieren kann. Dies ermöglicht es, komplexe Strukturen direkt innerhalb von *Unreal* zu kombinieren [19]. Der manuelle Aufwand für die Integration des Systems in eine Applikation kann so signifikant reduziert werden und macht den Einsatz eines LOD-Systems attraktiver.

Unity

Unity verfügt über ein DLOD-System, welches es erlaubt LOD-Artefakte manuell zu definieren. Innerhalb der *Unity* Community gibt es eine Bibliothek für das automatische Generieren von Artefakten, welche für LOD verwendet werden können [20]. Ansätze für ein HLOD-System sind vorhanden, wurden jedoch nicht offiziell in *Unity* integriert [21]. In Abbildung 3.1 ist zusehen, wie innerhalb von *Unity* der Übergang zwischen zwei LOD-Artefakten konfiguriert werden können. Hervorzuheben sind hier zum einen den Fade Mode, der das *Visual Popping* vermindert und zum anderen die Triggerpunkte, bei welchen das Level gewechselt werden soll.



Abbildung 3.1.: 'LOD Group'-Komponente in Unity für die Konfiguration der Übergänge zwischen LOD-Artefakten

3.2. Surface Simplification Algorithmus

Verschiedene Algorithmen wurden für die Implementierung in Betracht gezogen. Basierend auf den Erkenntnissen von D. Luebke wurde als Algorithmus *Surface Simplification using Quadric Error Metrics* gewählt [22]. Der Algorithmus wurde von M. Garland 1997 definiert und basiert auf *Edge Collapses*. Der Ansatz überzeugt durch die Balance zwischen Performanz und Qualität [23, 24].

3.2.1. Grobablauf

Der Algorithmus kann wie folgt zusammengefasst werden:

1. Fehlermetrik für alle *Vertices* berechnen.
2. Optionen für *Edge Collapse* markieren.
3. *Edge Collapses* mit geringstem Fehler, solange durchführen bis gewünschte Anzahl *Vertices* erreicht ist.

3.2.2. Fehlermetrik

Die Fehlermetrik dient dazu, eine Heuristik zu definieren, welche den geometrischen Unterschied zwischen dem vereinfachten und dem originalen Modell beschreibt. Mithilfe der Fehlermetrik können anschliessend die Transformationen am Modell gemäss den kleinsten Kosten optimiert werden. Der Algorithmus funktioniert grundsätzlich unabhängig von der gewählten Fehlermetrik. So könnten verschiedene Metriken eingesetzt werden. Die hier gewählte Metrik definiert den Fehler mithilfe einer symmetrischen 4×4 Matrix. Die Matrix repräsentiert die Distanz eines *Vertices* zu einer zugehörigen *Face*. Für die Kombination mehrerer *Faces* können die Matrizen addiert werden. Beim Entfernen einer *Edge* werden die Fehlermetriken der zugehörigen *Vertices* addiert.

3.2.3. Referenzimplementation

Der gewählte Algorithmus wurde bereits mehrfach implementiert. Eine wiederverwendbare Implementation basierend auf einer formatunabhängigen Datenstruktur ist jedoch nicht vorhanden. Eine oft referenzierte Implementation ist diejenige von S. Forstmann, welche einige Optimierungen am ursprünglichen Algorithmus vornimmt, um optimale Performanz zu erhalten. Diese Optimierungen eignen sich hervorragend für eine webbasierte Implementation [25].

3.3. Pipeline Integration

Die Lösung soll in eine wiederverwendbare und konfigurierbare Pipeline integriert werden. In der Webentwicklung ist es üblich, über `CLI`¹ Arbeitsschritte zu automatisieren. Es soll demnach ein `CLI` erstellt und auf npm öffentlich zugänglich gemacht werden, welches nahtlos in den Entwicklungsablauf integriert werden kann. Über eine Konfiguration soll angegeben werden können, wo nach den `.glTF` Dateien gesucht, wohin die Output-Dateien gespeichert und in welchem Modus das Tool gestartet werden soll. Ebenso werden Algorithmus-Einstellungen in dieser Konfiguration gesetzt werden können. Es soll einfach zu bedienen, gut dokumentiert und gut gewählte Standardeinstellungen haben. Zudem soll das Tool im einmal Modus laufen können oder kontinuierlich sich ändernde Dateien neu optimieren.

3.4. Nutzen LOD

Um den Nutzen von LOD quantifizieren zu können, müssen verschiedene Aspekte berücksichtigt werden. Auf der einen Seite ist die Entwicklungszeit zu berücksichtigen, welche sich auf die Kosten der Applikation auswirkt. Auf der anderen Seite sind Aspekte wie die Downloadzeit, die visuellen Unterschiede und die Laufzeit-Performanz der Applikation wichtig, da sie die Qualität der Software beeinflussen.

3.4.1. Aspekte für Nutzen

Der Nutzen eines Systems definiert sich durch eine Vielzahl von Aspekte, welche in verschiedene Phasen aufgeteilt werden können. Die verschiedenen für die Entwicklung von 3D-Webanwendungen relevanten Aspekte werden im folgenden Abschnitt erläutert. Es werden hierbei lediglich die Relevantesten berücksichtigt.

Entwicklungszeit

Die Entwicklungszeit beziffert den Aufwand für die Entwicklung einer Anwendung und kann in Arbeitsstunden beziffert werden. Die Auswirkungen eines Systems auf die Entwicklungszeit ist entscheidend dafür, ob etwas eingesetzt werden kann. Umso grösser der

¹Command Line Interface – Kommandozeile

Nutzen eines Systems, desto grösser kann der Einfluss auf die Entwicklungszeit sein. Manuelle Prozesse sind wo möglich zu vermeiden. Ein System kann insofern in der Praxis nur relevant sein, wenn die Entwicklungszeit im Verhältnis zum Nutzen steht. Das Ziel muss also sein, den Aufwand und somit die Entwicklungszeit für die Nutzung von LOD möglichst gering zu halten, um die Hürde der Benutzung zu minimieren.

Downloadzeit

Die Downloadzeit ist grundsätzlich linear abhängig von der Dateigrösse. Je grösser die Dateien, desto mehr Bandbreite wird verwendet. Dies hat direkte Auswirkungen auf die Zeit, bis eine Anwendung benutzt werden kann. Somit sollte die Downloadgrösse möglichst tief gehalten werden. Dank der heutigen Bandbreiten in der Schweiz ist dies für viele Anwendungsbereiche ein kleiner werdendes Problem.

Quelle suchen

Zudem können Probleme in diesem Bereich teilweise kaschiert werden, indem man das Ladeverhalten optimiert. Häufig sind nicht alle Artefakte notwendig, um eine verwendbare Anwendung darzustellen und der Rest kann phasenweise nachgeladen werden. Ausserdem ist es auch möglich, Artefakte bereits vorzuladen.

Visuelle Auswirkungen

Die negativen visuellen Auswirkungen auf eine Anwendung sollen so gering wie möglich ausfallen. Es gilt auch in den visuellen Bereichen die richtige Balance zwischen Laufzeitverhalten und möglichst hohem Realismus zu finden. Ein Beispiel für negative visuelle Auswirkungen ist die Reduktion der Auflösung - eine tiefe Auflösung wird von vielen Benutzern als störend empfunden. Im Zusammenhang mit LOD Artefakten ist insbesondere das sogenannte *Visual Popping* erwähnenswert. Beim *Visual Popping* handelt es sich um den Moment, in dem die Artefakte ausgetauscht wird und somit kurz aufblitzt. Je subtiler die Änderungen an den Artefakten und je grösser die Distanz von der Kamera ist, desto weniger ist das Austauschen bemerkbar.

Laufzeitverhalten

Für jede Applikation ist das Laufzeitverhalten entscheidend. So ist eine schlechte *User Experience* inakzeptabel. *User Experience* ist häufig Teil der nicht funktionalen Anforderungen und besteht aus mehr als nur dem Laufzeitverhalten. Das Laufzeitverhalten wird durch eine Vielzahl von Aspekten beeinflusst. Für flüssige Animationen sind die *Frames per Second* (FPS) entscheidend. Bei 10 *FPS*² oder weniger, sind flüssige Bewegungen nicht mehr möglich und das menschliche Auge kann ein Ruckeln wahrnehmen. Diese Zahl ist abhängig vom Individuum, als Faustregel gilt das Ziel von 60 *FPS* [26].

²Bilder pro Sekunde. Die Anzahl der Render-Durchläufe der Rendering Engine, die sich darauf auswirkt, wie flüssig eine Applikation läuft.

Berücksichtigung der Aspekte

In dieser Arbeit wird ein hoher Wert darauf gelegt, dass die Entwicklungszeit möglichst geringfügig erhöht wird. Zudem werden, wo möglich, Empfehlungen getätigt, um die Downloadgrösse nur marginal zu verändern. Das Minimieren der visuellen Auswirkungen kann durch visuelle Vergleiche sichergestellt werden. Die somit primären Aspekte für die Beurteilung sind die Auswirkungen auf das Laufzeitverhalten.

Um diese Beurteilung tätigen zu können, wird im Folgenden ein Benchmark definiert, mit dem Ziel, das Laufzeitverhalten zu analysieren und somit den maximal möglichen Einfluss von LOD auf die Leistung klassifizieren zu können. Hierbei werden verschiedene Faktoren betrachtet.

3.4.2. Vergleichbare Arbeiten

Eine unabhängige Arbeit definierte 2017 einen vergleichbaren Benchmark, um die Performanz von Three.js und Babylon.js in einem spezifischen Anwendungsgebiet zu vergleichen. Der Benchmark entwickelte vergleichbare Szenarien und vergleicht die Ressourcennutzung der beiden Varianten [27].

3.4.3. Mögliche Ansätze

Verschiedene Varianten können gewählt werden, um den Nutzen aufzuzeigen. Die verschiedenen in Betracht gezogenen Ansätze werden in diesem Abschnitt erläutert.

Modellbetrachtung

Grundsätzlich kann der Nutzen beziffert werden, indem die Kosten zum Rendering eines Modells berechnet werden. So können die Kosten der benutzten GPU Ressourcen durch das Messen des Aufwandes eines spezifischen Draw Calls beziffert werden. Die Implementation ist simpel und liefert genaue Resultate. Die Auswirkungen von Nebeneffekten können beinahe vollständig negiert werden. Sinnvoll wäre es, das Modell in konstanter Distanz zur Kamera um zwei Achsen drehen zu lassen und die Messung so lange durchzuführen, bis eine ganze Umdrehung stattgefunden hat. Das originale Modell kann mit den verschiedenen Stufen verglichen und die Performanzverbesserung beziffert werden. Der primäre Nachteil bei dieser Variante liegt an der Aussagekraft. Die einzige belegte Aussage ist, dass es möglich ist, Performanzeinsparungen durch Vereinfachung zu erhalten. Dies dient jedoch nicht als Beleg, dass der Einsatz von LOD-Artefakten Performanzengpässe lösen kann, da beim Einsatz von LOD weitere Aspekte berücksichtigt werden müssen. Einige Beispiele für diese Aspekte sind: erhöhte Memory Auslastung durch die zusätzlichen Artefakte oder zusätzlich erforderliche Rechenleistung für das Berechnen des zu wählenden Artefakts.

Szeneriebetrachtung

Bei diesem Ansatz wird eine gegebene Szenerie möglichst geringfügig angepasst und die Auswirkungen auf die Performanz verglichen. Wenn die unoptimierte Variante 60 *FPS*

erreicht oder gar übersteigt, ist eine Optimierung nicht notwendig. Somit muss eine Szenerie gewählt werden, welche diesen Wert unterschreitet. Das Ziel ist es dann, die LOD-Artefakte in dieselbe Szenerie zu integrieren und die *FPS* zu steigern. Ziel der Arbeit ist es, ein System zu entwickeln, das für Szenarien unter 60 *FPS* diese erhöht und im Idealfall auf ebendiesen Wert hebt. Eine solche Szenerie soll grundsätzlich Modelle sowohl nah als auch in grosser Distanz zeigen. Ein Beispiel für eine solche Szenerie ist eine 3D-Welt, in der man sich durch offene Gelände bewegt. Im Vergleich zur Modellbetrachtung ist die Aussagekraft eines solchen Tests ausreichend, um einen Beleg für den positiven Nutzen des Systems zu haben. Sollte es nicht möglich sein, die *FPS* zu steigern, so liefert der Einsatz von LOD keinen nachweisbaren Nutzen.

4. Resultate

Das Resultat besteht wie beschrieben aus verschiedenen Artefakten. In diesem Abschnitt wird konkret auf die Resultate der verschiedenen Schritte eingegangen und der Zusammenhang hergestellt.

4.1. LOD Pipeline

Der Kern der Arbeit stellt die LOD-Pipeline, *lode* genannt, dar. Dieses Tool liefert eine für das Web optimierte Möglichkeit, LOD-Artefakte für eine Vielzahl von Anwendungsgebieten zu generieren.

4.1.1. Aktueller Workflow

Der übliche Ablauf für das Generieren von LOD-Artefakten ist wie folgt: Für ein gegebenes Modell werden innerhalb vom Modellierungstool wie zum Beispiel Blender bestimmte LOD-Stufen von Hand generiert. Anschliessend werden die verschiedenen Stufen exportiert und manuell in die Applikation integriert. *Fine Tuning* erfordert sowohl Anpassungen an den Modellen als auch im Code. Erst muss im Modellierungstool die Anpassung manuell vorgenommen, dann neu in die Applikation geladen und dort neu geprüft werden. Diese Schritte können sich mehrfach wiederholen, was mühselig und teuer ist.

4.1.2. lode Pipeline

Das Ziel von *lode* ist es, für ein möglichst breites Spektrum von Anwendungsfällen eine einfache Lösung anzubieten und somit die Hemmschwelle für den Einsatz von LOD-Artefakten zu reduzieren.

Deshalb setzt *lode* konsequent auf moderne Entwicklungsprozesse, die nahtlos in die bestehenden Prozesse integriert werden können. Manuelle Schritte sollen auf ein Minimum reduziert und Finetuning so einfach wie möglich werden.

4.1.3. lode Ablauf

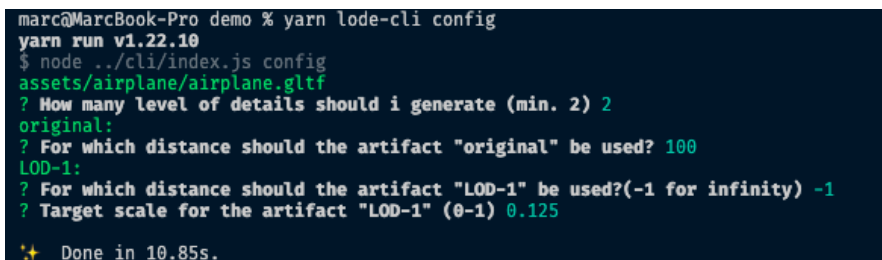
Der Ablauf der *lode*-Pipeline kann in drei Schritte unterteilt werden.

Setup und Konfiguration

3D-Modelle werden erstellt und als glTF-Dateien innerhalb des Projekts gespeichert. Eine zu den Modellen gehörige Konfigurationsdatei (Abbildung 4.1) kann mittels *CLI* angelegt werden (wie in Abbildung 4.2 dargestellt).

```
{
  "levels": [
    {
      "threshold": 300
    },
    {
      "threshold": -1,
      "configuration": {
        "targetScale": 0.0625
      }
    }
  ]
}
```

Abbildung 4.1.: Durch das CLI generierte Konfigurationsdatei



```
marc@MarcBook-Pro demo % yarn lode-cli config
yarn run v1.22.10
$ node ../cli/index.js config
assets/airplane/airplane.gltf
? How many level of details should i generate (min. 2) 2
original:
? For which distance should the artifact "original" be used? 100
LOD-1:
? For which distance should the artifact "LOD-1" be used?(-1 for infinity) -1
? Target scale for the artifact "LOD-1" (0-1) 0.125
🌟 Done in 10.85s.
```

Abbildung 4.2.: lode CLI config – Konfiguriert die gewünschten LOD-Artefakten

Erstellen der Artefakten

Danach generiert *lode* die in der Konfigurationsdatei beschriebenen LOD-Artefakte (in Abbildung 4.3 ersichtlich). Wenn sich ein Modell oder eine Konfiguration ändert, werden die notwendigen Schritte automatisch erneut durchgeführt. So ist es möglich, schnell Anpassungen vorzunehmen und die Änderungen im Projekt ohne manuelle Arbeit zu sehen.

```
marc@MarcBook-Pro demo % yarn lode-cli -w
yarn run v1.22.10
$ node ../cli/index.js -w

LODE

Preparing output folder:
done
Running initial LOD transformation:
performing LOD algorithm on file assets/airplane/airplane.gltf
done
performing LOD algorithm on file assets/duck/duck.gltf
done
watching files and running lode api server on port 3001...
performing LOD algorithm on file assets/airplane/airplane.gltf
done
```

Abbildung 4.3.: lode CLI run – Generiert die gewünschten LOD-Artefakte

Das zugehörige *lode-ui* (in Abbildung 4.4 gezeigt) bietet zudem die Möglichkeit, die verschiedenen Detailstufen miteinander zu vergleichen und die Konfiguration möglichst einfach zu justieren. Dies ermöglicht es, die Änderungen in Echtzeit zu sehen und die Distanzen, bei welchen Level aktiviert werden, einzustellen. So ist kein manuelles hin und her wechseln zwischen Applikationen mehr notwendig.

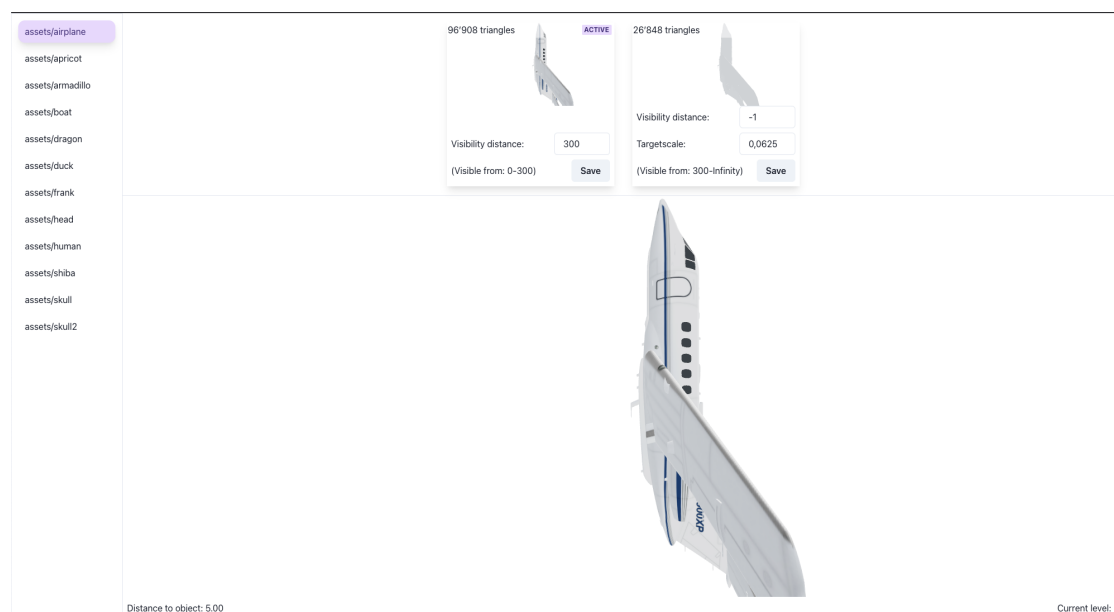


Abbildung 4.4.: lode-ui

Verwendung der Artefakte in der Applikation

Die Artefakte können innerhalb der Applikation mithilfe einer Bibliothek geladen werden. Dabei werden die Definitionen in der Konfigurationsdatei geladen und das Modell entsprechend in der Szenerie dargestellt, wie in Beispielcode 4.5 zu sehen ist.

```
1  import * as lodeLoader from "lode-three";
2  import manifest from "../lode-build/lode-manifest.json";
3
4  const lodeContext = lodeLoader.createContext({
5    manifest,
6    basePath: "../lode-build",
7  });
8
9  const elephantModel = await lodeLoader.loadModel({
10    lodeContext,
11    artifactName: 'assets/elephant',
12  })
13
14  scene.add(elephantModel)
15  clone.position.set(0, 0, 0);
```

Abbildung 4.5.: Beispielcode zur Benutzung der LOD-Artefakte mittels *lode-three* in *Three.js*

4.2. LOD Generierung

Für das Erstellen von LOD-Artefakten wurde eine für LOD-Artefakte optimierte Version des in Abschnitt 3.2 erklärten *Surface Simplification Algorithmus* entwickelt.

4.2.1. Implementation

Die Implementation des Algorithmus wurde in JavaScript, basierend auf Node.js¹, umgesetzt. Grund dafür ist das Einbinden von *lode* in bestehende Entwicklungsabläufe in der Webentwicklung. So ist es einfach möglich, das Paket kostenlos mithilfe von npm zur Verfügung zu stellen.

mehr details über Implementation: boundary normalization, optimal point, matrices...

4.2.2. Artefakte

Die generierten Artefakte sind unabhängig voneinander. So ist es nicht notwendig alle Artefakte zu jeder Zeit zu laden. Dies hat den primären Vorteil, dass es möglich ist, progressives Laden als Erweiterung zuzulassen. Progressives laden bedeutet, dass zuerst die LOD-Artefakte mit den wenigsten Details geladen werden und angezeigt werden bis die detaillierteren Level visualisiert werden. Es ist sogar möglich, auf gewissen Geräten die detaillierten Level überhaupt nicht zu laden und so zum Beispiel Bandbreite zu sparen.

¹JavaScript Ausführungsumgebung, welche ohne einen Browser verwendet werden kann

4.2.3. Fehlermetrik

Die Fehlermetrik wurde vergleichbar mit der Referenzimplementation justiert. Die Referenzimplementation setzt auf absolute Fehlerwerte. Dies hat zur Konsequenz, dass abhängig von der Skalierung des Modells potenziell zu viel oder zu wenige *Edge Collapses* durchgeführt werden. Ein Modell mit geringer Skalierung generiert tiefere Werte für die Fehlermetrik und umgekehrt. Um diesem Umstand gerecht zu werden, wird vorab die Skalierung der Modelle normalisiert. Die *Vertices* werden dabei um einen modellabhängigen Faktor angepasst. So haben alle Modelle eine vergleichbare Basis für die Fehlermetrik und starke Unterschiede zwischen verschiedenen Modellen können vermieden werden.

4.2.4. Texturen

Bei vielen Modellen sind der grösste Teil der Speichergrösse die Texturen. Das Optimieren der geometrischen Struktur reduziert die zugehörigen Texturen jedoch nicht. Um trotzdem möglichst grosse Einsparungen für die Texturen zu ermöglichen wurde eine Methode entwickelt, welche versucht die prominenteste Farbe eines Modells zu extrahieren und diese für die Vereinfachungen zu verwenden. In Abbildung 4.6 ist dieser Unterschied visualisiert. Diese Technik führt zu signifikanten Detailverlusten bei der Nahbetrachtung und ist somit ungeeignet für das Verwenden bei der ersten LOD-Stufe. Für alle weiteren Stufen überwiegt jedoch der Vorteil der Vereinfachung im Vergleich zum Detailverlust.

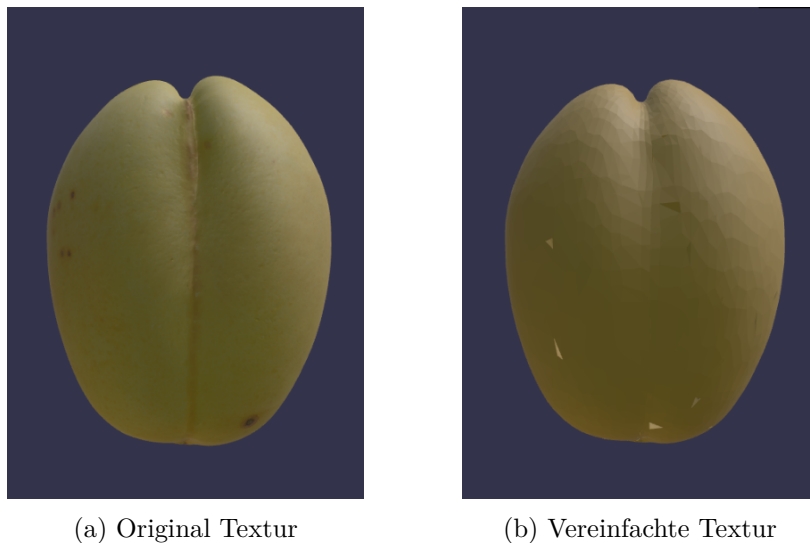


Abbildung 4.6.: Vergleich der Texturen

4.2.5. Vergleich

Unter Berücksichtigung der Auswirkungen auf die Downloadgrösse wird der Einsatz von zwei Stufen empfohlen: das Original und das Vereinfachte.

In Abbildung 4.7 ist eine solche Vereinfachung neben dem zugehörigen Original ersichtlich. Beim Original handelt es sich um ein Modell mit 4212 *Triangles*. Die Vereinfachung hat lediglich noch deren 269. Während das Original eine Dateigrösse von 122 KB aufweist, hat die Vereinfachung lediglich 6 KB. Vergleichbare Relationen treten auch mit komplexeren Modellen auf.

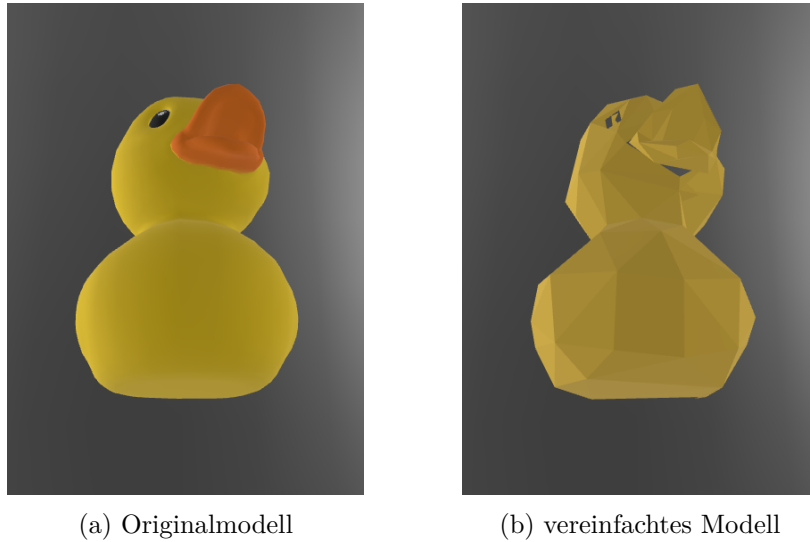


Abbildung 4.7.: Vergleich Originalmodell und vereinfachtes Modell

Bei einer Nahbetrachtung der Modelle sind die Unterschiede klar ersichtlich. Bei einer gewissen Distanz sind die Differenzen der beiden Modelle unauffälliger. In Abbildung 4.8 werden dieselben Modelle auf eine Distanz von 72 (m) dargestellt. Die Standardeinstellung für die maximale Distanz des Kamera Frustum liegt bei Babylon.js zum Beispiel bei 10'000 (m) [28]. Auf diese Distanz kann der orange Schnabel beim Originalmodell noch knapp erkannt werden - andere Details sind auf diese Distanz jedoch nicht ersichtlich.



Abbildung 4.8.: Vergleich Originalmodell und vereinfachtes Modell auf Distanz

4.3. Benchmark

Das Ziel des Benchmarks ist der Vergleich einer Version ohne Einsatz von LOD und einer Umgebung, welche LODs einsetzt. Um eine möglichst praxisnahe Aussage treffen zu können, wird hierfür eine Demo-Szenerie verwendet, welche einer echten Anwendung so nah wie möglich kommt. Somit wird gewährleistet, dass es nicht nur in der Theorie einen Nutzen für LODs gibt, sondern dieser in der Praxis vorhanden ist.

Browser Umgebung

Um den Umfang des Benchmarks überschaubar zu halten, wurde ausschliesslich ein Benchmark für Google Chrome entwickelt. Google Chrome basiert auf Chromium², dieselbe Engine, welche auch Microsoft Edge oder Opera verwenden. Einen Benchmark basierend auf Google Chrome liefert somit auch Indizien für diese beiden Browser, auch wenn gewisse Abweichungen möglich sind. Neben dem Marktführer Chrome sind Mozilla Firefox oder Safari von Apple ebenfalls Optionen. Jedoch wurde primär aufgrund des Marktanteils von total rund 70% [29] zugunsten von Google Chrome entschieden. Die getroffenen Aussagen bezüglich Laufzeitverhalten behalten ihre Gültigkeit auch für andere Browser.

Automation

Um die Tests durchzuführen, wird ein Testautomationstool benötigt; unter anderem der Einsatz von Selenium wurde in Erwägung gezogen. Der Vorteil von Selenium ist insbesondere, dass der Benchmark für weitere Browser ausgeweitet werden könnte. Da jedoch das

²Open Source Browser-Projekt, welches von Google entwickelt wird

Analysieren der GPU Daten stark vom System abhängig ist und dafür zusätzliche Komplexität notwendig wäre, wird in diesem Benchmark die im Google Chrome integrierten *Chrome DevTools* eingesetzt. Selenium bietet zurzeit eine suboptimale Integration für das *Chrome DevTools Protocol*. Um mögliche Diskrepanzen zwischen Systemen möglichst gering zu halten, wurde jedoch entschieden, auf die bewährte Lösung von Google Chrome zu setzen. Puppeteer³, eine weitere Option für die Automation, ist eine Bibliothek, die eine vereinfachte Schnittstelle zu einer Chromium Instanz bietet. Sie wird zudem direkt von Google entwickelt und bietet somit eine stabile Grundlage zur Kommunikation mit den *Chrome DevTools*.

Profiling

Die *Chrome DevTools* erlauben es, ein detailliertes Laufzeitprofil einer Applikation anzulegen. Im Profil befinden sich Informationen zu CPU- und GPU-Auslastung, aber auch generelle Informationen bzgl. der Rendering Engine werden gesammelt. Die Analyse dieser Daten ermöglicht es, eine Aussage zum Laufzeitverhalten einer Applikation zu tätigen.

Testaufbau

Derselbe Testablauf wird sowohl für die optimierte als auch für die unoptimierte Version verwendet. Bei einem Testablauf werden folgende Schritte durchlaufen:

1. Öffne die Applikation in einer *Chromium* Instanz.
2. Warte bis die Seite geladen ist.
3. Starte das *Profiling*.
4. Warte n Sekunden.
5. Stoppe das *Profiling*.
6. Werte die Daten aus.

Der Test erfasst die in Tabelle 4.1 aufgeführten Kennzahlen.

³Node.js Library, die eine API anbietet zum Steuern von Chromium oder Chrome über das Chrome DevTools Protocol

| Kennzahl | Beschreibung |
|---------------------------------------|--|
| Median <i>FPS</i> | Die <i>FPS</i> werden kontinuierlich berechnet. Um starke Abweichungen zu verwerfen wird der Median verwendet. |
| Dauer für das Laden der Modelle | Totale Zeit für das Laden aller Modelle der Szenerie. Dieser Wert ist relativ zu betrachten, da die Modelle lokal geladen werden und die Zeit für das Laden von einem Server signifikant höher sein kann. Grundsätzlich besteht eine ausreichende Korrelation zwischen Dateigrösse und Zeit für das Laden der Modelle. Ein zusätzlicher Faktor ist jedoch die Anzahl an Dateien, welche geladen werden müssen. |
| Median <i>Render Loop</i> Dauer | Der Median aller Laufzeiten der <i>Render Loop</i> . |
| Anzahl <i>Render Loop</i> Iterationen | Wie oft wurde ein neues Bild gezeichnet. Umso höher die Zahl, desto mehr verschiedene Frames konnten gerendert werden. |
| Totale GPU Zeit | Die totale Zeit, welche die GPU für Berechnungen benötigt. |
| Anzahl GPU Events | Anzahl der Events an die GPU. Dieser Wert soll lediglich dazu dienen um die gemessenen <i>FPS</i> und die Anzahl <i>Render Loop</i> Iterationen besser einschätzen zu können. Eine höhere Anzahl an GPU Events steht innerhalb der Demoszenerie im Zusammenhang mit mehr <i>Render Loop</i> Iterationen. |

Tabelle 4.1.: Kennzahlen für Benchmark

Aufbau Testapplikation

Die Testapplikation stellt eine komplexe Szenerie dar. Der Betrachter fliegt während dem Ablauf kontinuierlich über die Szenerie. Dies stellt die optimalen Bedingungen für den Einsatz von LOD-Artefakten dar. Abhängig von einem URL-Parameter wird entschieden ob LOD-Artefakte verwendet werden sollen oder nicht. Die Anwendung wurde mit Three.js implementiert. Bei der unoptimierten Version wird direkt das Originalmodell verwendet. Bei der optimierten Version werden mithilfe der LOD-Hilfsklasse die verschiedenen Artefakte geladen [30].

In Abbildung 4.9 ist ein Screenshot der Testapplikation ersichtlich. Als Basis für die Applikation wurde Three.js eingesetzt. Die Kamera wird kontinuierlich innerhalb der Szene bewegt, die Position wird abhängig von der Systemzeit definiert. So ist es möglich, dass beide Applikationen – unabhängig von den *FPS* – jeweils die gleichen Aspekte innerhalb der Applikation visualisieren. Die Modelle werden zudem bei jedem Durchlauf identisch positioniert. Dies gewährleistet, dass das Laufzeitverhalten verlässlich ist.

Jedes Modell wird dabei mehrfach angezeigt, die Modelle werden einmal geladen und anschliessend mittels der *clone* Methode von Three.js geklont [31]. Dies stellt sicher, dass die Datenstruktur der Modelle nicht mehrfach in den Arbeitsspeicher geladen werden muss. Wichtig ist zu erwähnen, dass die Modelle bei dieser Methode mit mehreren *Draw Calls* gerendert werden. Der Einsatz von *InstancedMesh* wurde in Erwägung gezogen, zeigte jedoch keine nennenswerte Unterschiede zwischen der optimierten und unoptimierten Version [32].

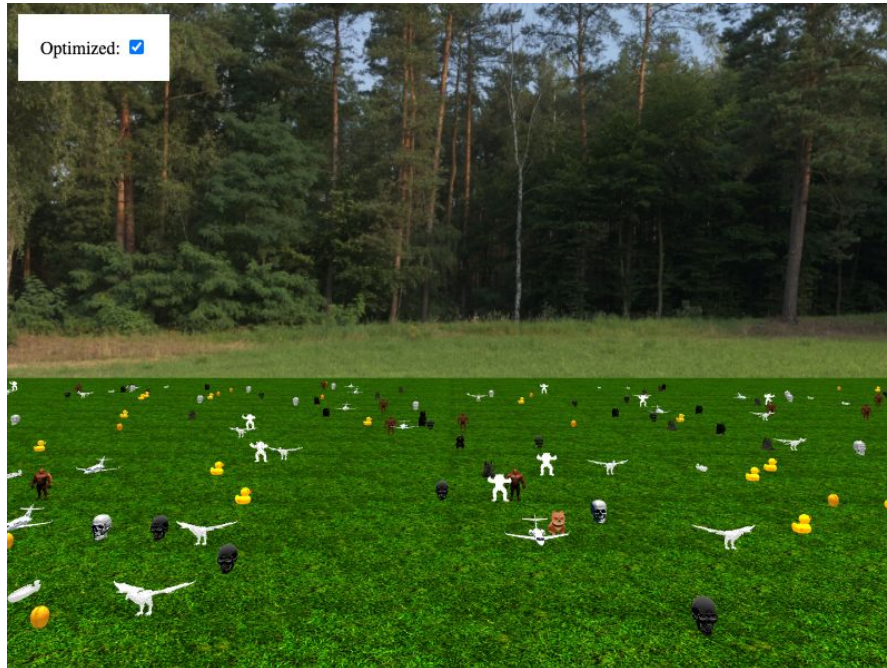


Abbildung 4.9.: Testapplikation

Testumgebung

Um während den Tests möglichst faire Bedingungen zu gewährleisten, wird die Maschine zuvor wie bei anderen Benchmarks vorbereitet. Ziel ist es, Seiteneffekte zu minimieren. Für diesen Benchmark wurden deshalb die Instruktionen von *Tracer Bench* zur Behebung von Rauschen befolgt [33]. Die beiden Versionen werden jeweils abwechselungsweise ausgeführt, so wird sichergestellt, dass keine der beiden Varianten einen starken Nachteil durch Nebeneffekte erleidet. Ausserdem werden mehrere Durchläufe direkt nacheinander ausgeführt. Starke Abweichungen zwischen den Durchläufen würden bei der Analyse der Daten auffallen.

Analyse der Daten

Für jeden Durchlauf wird der Median der *FPS* Daten berechnet. Anschliessend wird die Standardabweichung der *FPS* für die unoptimierten respektive optimierten Werte

berechnet. Die Standardabweichung dient als Kennzahl, um eine Signifikanz der Daten nachweisen zu können.

Zudem werden die weiteren erfassten Kennzahlen ausgewertet, um das Resultat der *FPS* verifizieren zu können.

Konfidenzintervall Die Signifikanz wird mithilfe eines statistischen Konfidenzintervalls nachgewiesen. Hierfür wird der Durchschnitt aller Mediane verwendet, zusätzlich wird ein Konfidenzintervall von 95% gewählt. Es wird gewährleistet, dass das Resultat des Benchmarks verlässlich ist und für einen Vergleich verwendet werden kann. Überschneiden sich die Intervalle für die optimierte und unoptimierte Version kann keine statistische Signifikanz nachgewiesen werden. Grundsätzlich gilt: umso kleiner die Intervalle desto besser. Zudem sollten die beiden Intervalle möglichst weit entfernt voneinander liegen.

4.3.1. Auswertung

Der Benchmark wurde auf einer Maschine mit moderner Grafikkarte ausgeführt. Die Resultate zeigen einen klaren Unterschied zwischen der unoptimierten und optimierten Version. So ist die Zeit für das Laden der Modelle zwar geringfügig länger, die Zahl der *Render Loop* Iterationen und insbesondere der *FPS* geben eine klare Auskunft. Unter Berücksichtigung des berechneten Konfidenzintervalls wird das Resultat als signifikant erachtet. Diese These wird auch durch eine einfache Kontrolle mit dem Auge belegt - die Anzahl *FPS* unterscheidet sich merkbar.

Somit ist es möglich einen Nutzen für bestimmte anspruchsvolle 3D-Visualisierungen zu generieren.

```
optimized fps: 59.9 (0.316 standard deviation)
the value is with a confidence of 95% between 59.704 and
60.096
baseline fps: 48.3 (0.483 standard deviation)
the value is with a confidence of 95% between 48.001 and
48.599

further information for interpreting data:

gpuTotalTime:
optimized: 108.406 (5.667 standard deviation)
baseline: 336.718 (398.204 standard deviation)
medianRenderLoopDuration:
optimized: 2.067 (0.064 standard deviation)
baseline: 1.394 (0.024 standard deviation)
totalGpuEvents:
optimized: 524.7 (26.081 standard deviation)
baseline: 518.6 (17.89 standard deviation)
totalModelLoadDuration:
optimized: 1350.611 (42.677 standard deviation)
baseline: 1233.288 (53.571 standard deviation)
totalRenders:
optimized: 300.4 (5.602 standard deviation)
baseline: 232.3 (5.658 standard deviation)
```

Abbildung 4.10.: Durchlauf eines Benchmarktests

Durch visuelle diagramme ersetzen.

Wie in Abbildung 4.10 zu sehen, konnten die FPS auf 60 FPS gehoben werden im Vergleich zu den 48.2 FPS in der unoptimierten Version. Direkt damit verbunden ist die Anzahl *Render Loop* Iterationen, welche genauso um ca 30% gesteigert werden konnte. Auf der Umkehrseite wurde die Dauer für das Laden der Modelle etwas erhöht, da mehr Modelle geladen werden müssen. Diese wuchs aber nur um rund 10%.

5. Diskussion und Ausblick

Das entwickelte Tool, *lode*, liefert einen ersten Schritt in die Richtung einer LOD-Pipeline, welche für das Web optimiert wurde. Das Tool erreicht das Ziel, für eine Vielzahl von Anwendungsgebieten einsetzbar zu sein. Wichtig ist es jedoch festzuhalten, dass es nicht für alle Anwendungen sinnvoll ist. Zum einen soll die Pipeline weiter entwickelt werden und in ihren Kernkompetenzen gestärkt werden, zum anderen gibt es jedoch auch spezifische Gebiete, wie zum Beispiel Terrain, für welche *lode* ungeeignet ist.

5.1. LOD System

Der grösste Nutzen entsteht in Anwendungen mit komplexen Modellen - für diese Situationen bietet *lode* einen nachweisbaren Mehrwert. Die Zahlen aus Unterabschnitt 4.3.1 zeigen den möglichen Nutzen in einem Anwendungsbeispiel.

5.1.1. Allgemeingültigkeit

Innerhalb der Testapplikation, wie in Absatz 4.3 beschrieben, werden dieselben Modelle mehrfach angezeigt. Dies wird in der Praxis insbesondere bei repetitiven Elementen, wie zum Beispiel Bäumen für Landschaftsgestaltung, verwendet. In solchen Situationen stellen die Anzahl Polygone das primäre Problem für eine schnelle Laufzeitperformanz dar und der Einsatz von LOD-Artefakten kann erwägt werden. Die Allgemeingültigkeit ist für Anwendungsfälle mit vergleichbaren Anforderungen gegeben.

5.1.2. Anwendbarkeit

Der Algorithmus kann für eine Vielzahl von verschiedenen Modellen angewendet werden und liefert solide Resultate. Die geometrische Approximation ist ausreichend für ein LOD-System. Für Spezialfälle wie Randbehandlung sind entsprechende Kontrollen implementiert worden. Als Erweiterung ist es möglich, verschiedene *Meshes* eines glTF zu kombinieren und als ein einziges *Mesh* zu behandeln. Vorteil ist, dass noch mehr Vereinfachungen vorgenommen werden können. Dies kann man als simples HLOD-System bezeichnen.

5.1.3. Abgrenzung

Ein LOD-System kann sinnvoll sein, in einigen Anwendungsgebieten bringt es jedoch nicht den gewünschten Zweck. Wird ein Modell nur auf einer Detailstufe angezeigt, so ist der Einsatz von LOD abzuraten. Sind die Modelle bereits sehr simpel, wie bei *Low-Poly*-Modellen – in Abbildung 5.1 zu sehen – sind die zusätzlichen Vereinfachungen nur

von geringem Nutzen. So muss der Einsatz von LOD-Artefakten für jeden Anwendungsfall separat abgewogen werden. Ein Vorteil von *lode* ist die Einfachheit der Integration. So ist es möglich, mit wenig Aufwand den groben Nutzen für ein spezifisches Anwendungsbeispiel abschätzen zu können, ohne dass viel Entwicklungszeit dafür aufgewendet werden muss.

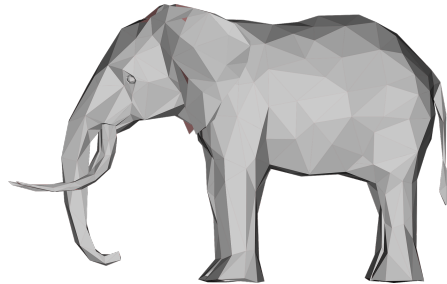


Abbildung 5.1.: Beispiel *Low-Poly*-Modell

5.2. Mögliche Erweiterungen

Die Möglichkeiten für Erweiterungen zu *lode* sind sehr umfangreich. In diesem Abschnitt wird auf einige Möglichkeiten eingegangen.

Akurate Texturinformationen

Zurzeit wird die Textur komplett entfernt und durch eine einfache Farbinformation ersetzt. Dies stellt beim aktuellen Einsatz der LOD-Artefakten kein Problem dar. Eine interessante Erweiterung ist so jedoch nicht möglich: das Vereinfachen der ersten Stufe. Der Algorithmus kann so erweitert werden, dass die Texturinformationen nicht vereinfacht, sondern als Teil der *Vertices* verwendet werden. Dies würde es ermöglichen, ein Modell, welches über mehrere Millionen Polygone verfügt, und somit ungeeignet für das Web ist, zu vereinfachen und diese erste Vereinfachung als detailreichste Stufe zu verwenden.

glTF LOD Format

Es gibt einen Vorschlag, um die Definition von LOD-Stufen innerhalb von glTF zu definieren. Dies würde es ermöglichen, verschiedene LOD-Artefakte in einer Datei zu speichern [34]. Was wiederum erlauben würde, Texturinformationen einfacher wiederzuverwenden. Die Pipeline kann grundsätzlich für diesen Anwendungsfall erweitert werden. Zurzeit hat Three.js jedoch noch keinen Support für diese Erweiterung. Auch Babylon.js unterstützt lediglich das progressive Laden von Modellen. [35]

Progressives Laden

Aktuell werden initial alle Level of Details geladen. Dies hat zur Konsequenz, dass im Vergleich zur unoptimierten Variante eine schlechtere initiale Ladezeit resultiert. Um

dieses Verhalten zu verbessern beziehungsweise sogar in einen Vorteil zu verwandeln, könnte das System so erweitert werden, dass die Modelle progressiv geladen werden. Es würde also zuerst die kleinste Datei geladen und bereits angezeigt werden, während in der Zwischenzeit die detailreicheren Modelle nachgeladen werden und die detailärmeren ersetzen. Somit reduziert sich die Zeit bis zum ersten visuellen Feedback und die *User Experience* kann damit verbessert werden.

Levelwahl während Laufzeit

Die optimale Wahl des Levels kann stark von der Laufzeitumgebung abhängig sein. So sollte zum Beispiel auf mobilen Geräten früher ein einfacheres Modell angezeigt werden als auf leistungsstarken Geräten. Ein weiterer Faktor ist die Auflösung: Bei Geräten mit niedriger Auflösung kann früher gewechselt werden, da die visuellen Unterschiede zum Original weniger auffällig sind. Deshalb eignet sich eine Wahl der Levels zur Laufzeit, um ein optimales Erlebnis auf allen Geräten zu ermöglichen. Neben dem *CLI* soll folglich auch ein Javascript-Modul bereitgestellt werden, welches das Laden von glTF Dateien übernimmt und basierend auf der erkannten Performanz das beste Level of Detail einspielt.

Fading

Aktuell finden die Modellwechsel an harten Grenzen statt. Dies hat zur Konsequenz, dass *Visual Popping* auftauchen kann. Eine Möglichkeit zur Linderung dieses Problems ist der Einsatz von *Fading*. Hierbei werden die Übergänge über grössere Intervalle durchgeführt und Teile der verschiedenen Artefakte dargestellt. Diese Methode wird zum Beispiel bei *Unity* angeboten.

Konsequenter Einsatz von glb Dateien

Aktuell werden in der Demoszenerie glTF Dateien eingesetzt. Dies kommt daher, dass die Basisinformationen menschenlesbar abgespeichert werden. Das binäre Format glb ist jedoch grundsätzlich glTF vorzuziehen und sollte als primäres Austauschformat verwendet werden. Für das Laufzeitverhalten entsteht kein signifikanter Unterschied durch den Einsatz von glb. Der primäre Vorteil vom glb Format ist, dass alles in einer Datei gespeichert ist und somit nicht zwei Netzwerkabfragen stattfinden müssen – die Downloadzeit wird somit reduziert. Das CLI könnte erweitert werden, auch glb Dateien zu unterstützen.

6. Verzeichnisse

Glossar

Chromium Open Source Browser-Projekt, welches von Google entwickelt wird. 39, 40, 49

CLI Command Line Interface – Kommandozeile. 29, 47

Draw Call Ein Aufruf auf die GPU, um ein Objekt zu zeichnen. Ein *Draw Call* wird pro Material pro *Mesh* benötigt. Es ist besser, weniger, dafür umfangreiche *Draw Calls* abzusetzen, als viele Kleinere. Dies kommt daher, dass die GPU extrem schnell ist und sich die Unterbrechung durch einzelne Aufrufe viel negativer auswirkt.. 22, 31

FPS Bilder pro Sekunde. Die Anzahl der Render-Durchläufe der Rendering Engine, die sich darauf auswirkt, wie flüssig eine Applikation läuft.. 30–32, 41–43

Game Engines Framework, das für den Spielverlauf und dessen Darstellung verantwortlich ist. 8

Node.js JavaScript Ausführungsumgebung, welche ohne einen Browser verwendet werden kann. 36

npm Node Package Manager. 9, 29, 36

OpenGL Spezifikation einer Programmierschnittstelle zur Entwicklung von 2D- und 3D-Grafikanwendungen. 8

Point Cloud Vertices im dreidimensionalen Raum ohne zugehörige *Triangle*. 24

Puppeteer Node.js Library, die eine API anbietet zum Steuern von Chromium oder Chrome über das Chrome DevTools Protocol. 40

Rendering Engine Teilprogramm, das zuständig für die Darstellung von Grafiken ist. 8, 30, 40, 49

Scene Graph Objektorientierte Datenstruktur zur Beschreibung von 2D- oder 3D-Szenarien. 23

Semantic Versioning Jede Version wird mit einer aus drei Zahlen bestehenden Versionsbezeichnung (z.B. v1.0.5) versehen. Wobei jede dieser drei Zahlen eine gewisse Gewichtung hat. So wird die vorderste Zahl als *Major* bezeichnet. Diese wird erhöht, wenn grössere Änderungen hinzukommen, die möglicherweise nicht rückwärtskompatibel sind. Die mittlere Zahl – *Minor* genannt – wird erhöht, wenn neue Funktionen hinzugefügt werden, welche rückwärtskompatibel sind. Die hinterste Zahl ist für *Patches*, welche Probleme beheben, ohne die Kompatibilität zu beeinträchtigen.. 9

Texture Mapping Verfahren, um mittels 2D-Bildern 3D-Objekte zu gestalten. 13

Literaturverzeichnis

- [1] World Bank. (2017) *Individuals using the Internet*. [Online]. URL: <https://data.worldbank.org/indicator/IT.NET.USER.ZS> [Stand: 14.05.2021].
- [2] D. Jackson und J. Gilbert. (2021) *WebGL Specification*. [Online]. URL: <https://www.khronos.org/registry/webgl/specs/latest/1.0/> [Stand: 26.03.2021].
- [3] GPU for the Web Community Group. (2021) *GPU for the Web Community Group Charter*. [Online]. URL: <https://gpuweb.github.io/admin/cg-charter.html> [Stand: 26.03.2021].
- [4] M. Segal, K. Akeley, C. Frazier, J. Leech und P. Brown. (2008) *The OpenGL® Graphics System: A Specification*. [Online]. URL: <https://www.khronos.org/registry/OpenGL/specs/gl/glspec30.pdf> [Stand: 26.03.2021].
- [5] npm Inc. (2021) *npm package three*. [Online]. URL: <https://www.npmjs.com/package/three> [Stand: 14.05.2021].
- [6] react-three-fiber contributors. (2021) *react-three-fiber on Github*. [Online]. URL: <https://github.com/pmndrs/react-three-fiber> [Stand: 14.05.2021].
- [7] npm Inc. (2021) *npm package react*. [Online]. URL: <https://www.npmjs.com/package/react> [Stand: 14.05.2021].
- [8] npm Inc. (2021) *npm package babylonjs*. [Online]. URL: <https://www.npmjs.com/package/babylonjs> [Stand: 14.05.2021].
- [9] npm Inc. (2021) *npm package playcanvas*. [Online]. URL: <https://www.npmjs.com/package/playcanvas> [Stand: 14.05.2021].
- [10] Unity Technologies. (2020) *Building and running a WebGL project*. [Online]. URL: <https://docs.unity3d.com/Manual/webgl-building.html> [Stand: 14.05.2021].
- [11] Unity Technologies. (2020) *Unity 2020.2.0a13 C# reference source code*. [Online]. URL: <https://github.com/Unity-Technologies/UnityCsReference> [Stand: 15.05.2021].
- [12] Babylon.js contributors. (2021) *Simplifying Meshes With Auto-LOD*. [Online]. URL: <https://doc.babylonjs.com/divingDeeper/mesh/simplifyingMeshes> [Stand: 04.05.2021].
- [13] unknown. (2007) *Object Files*. [Online]. URL: <http://www.martinreddy.net/gfx/3d/OBJ.spec> [Stand: 26.03.2021].

- [14] P. Cozzi und T. Parisi. (2016) *glTF 1.0 Specification*. [Online]. URL: <https://github.com/KhronosGroup/glTF/tree/master/specification/1.0> [Stand: 26.03.2021].
- [15] The Khronos Group Inc. (2021) *Introduction to glTF using WebGL*. [Online]. URL: https://github.com/KhronosGroup/glTF-Tutorials/blob/master/gltfTutorial/gltfTutorial_001_Introduction.md [Stand: 16.05.2021].
- [16] The Khronos Group Inc. (2021) *The Basic Structure of glTF*. [Online]. URL: https://github.com/KhronosGroup/glTF-Tutorials/blob/master/gltfTutorial/gltfTutorial_002_BasicGltfStructure.md [Stand: 16.05.2021].
- [17] L. Vieira. (2012) *Point cloud torus*. [Online]. URL: https://commons.wikimedia.org/wiki/File:Point_cloud_torus.gif [Stand: 24.05.2021].
- [18] B. Hamann. (1994) *A Data Reduction Scheme for Triangulated Surfaces*. [Online]. URL: <https://escholarship.org/uc/item/9x4542q1> [Stand: 24.05.2021].
- [19] S. Deiter. (2018) *An In-Depth Look at Unreal Engine 4.20's Proxy LOD Tool*. [Online]. URL: <https://www.unrealengine.com/en-US/tech-blog/an-in-depth-look-at-unreal-engine-4-20-s-proxy-lod-tool> [Stand: 03.05.2021].
- [20] M. Edlund. (2021) *Mesh simplification for Unity*. [Online]. URL: <https://github.com/Whinarn/UnityMeshSimplifier> [Stand: 03.05.2021].
- [21] A. Ebrahimi. (2018) *Unity Labs: AutoLOD – Experimenting with automatic performance improvements*. [Online]. URL: <https://blogs.unity3d.com/2018/01/12/unity-labs-autolod-experimenting-with-automatic-performance-improvements/> [Stand: 03.05.2021].
- [22] D. Luebke, „A developer’s survey of polygonal simplification algorithms”, *IEEE Computer Graphics and Applications*, Bd. 21, Nr. 3, S. 24–35, 2001.
- [23] M. Garland und P. S. Heckbert. (1997) *Surface Simplification Using Quadric Error Metrics*. [Online]. URL: <https://www.cs.cmu.edu/~garland/Papers/quadrics.pdf> [Stand: 17.04.2021].
- [24] M. Garland und P. S. Heckbert. (1998) *Simplifying surfaces with color and texture using quadric error metrics*. [Online]. URL: <https://ieeexplore.ieee.org/abstract/document/745312> [Stand: 14.05.2021].
- [25] S. Forstmann. (2020) *Fast Quadric Mesh Simplification*. [Online]. URL: <https://github.com/sp4cerat/Fast-Quadric-Mesh-Simplification> [Stand: 18.05.2021].
- [26] M. F. Deering. (2005) *The Limits of Human Vision*. [Online]. URL: <http://michaelfrankdeering.org/Projects/EyeModel/limits.pdf> [Stand: 24.05.2021].

- [27] O. Nordquist und A. Karlsson. (2017) *BabylonJS and Three.js - Comparing performance when it comes to rendering Voronoi height maps in 3D*. [Online]. URL: <http://www.diva-portal.org/smash/get/diva2:1228221/FULLTEXT01.pdf> [Stand: 17.04.2021].
- [28] Babylon.js contributors. (2021) *Babylon.js Source Code - camera.ts*. [Online]. URL: <https://github.com/BabylonJS/Babylon.js/blob/5e6321d887637877d8b28b417410abbb651c6e/src/Cameras/camera.ts#L185> [Stand: 22.05.2021].
- [29] StatCounter. (2021) *Browser Market Share Worldwide / StatCounter Global Stats*. [Online]. URL: <https://gs.statcounter.com/browser-market-share#monthly-202003-202103> [Stand: 14.04.2021].
- [30] Three.js contributors. (2021) *Three.js - LOD documentation*. [Online]. URL: <https://threejs.org/docs/#api/en/objects/LOD> [Stand: 22.05.2021].
- [31] Three.js contributors. (2021) *Three.js - Object3D.clone documentation*. [Online]. URL: <https://threejs.org/docs/#api/en/core/Object3D.clone> [Stand: 22.05.2021].
- [32] Three.js contributors. (2021) *Three.js - InstancedMesh documentation*. [Online]. URL: <https://threejs.org/docs/#api/en/objects/InstancedMesh> [Stand: 22.05.2021].
- [33] M. Lynch. (2020) *TracerBench: Noise Mitigation Techniques*. [Online]. URL: https://github.com/TracerBench/tracerbench/blob/master/NOISE_MITIGATION.md [Stand: 17.04.2021].
- [34] S. Bhatia, G. Hsu, A. Gritt, J. Copic, M. Appelsmeier und D. Frommhold. (2018) *glTF Extension MSFT lod*. [Online]. URL: https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Vendor/MSFT_lod [Stand: 28.04.2021].
- [35] Babylon.js contributors. (2021) *Progressively Load .glTF Files in Babylon.js*. [Online]. URL: <https://doc.babylonjs.com/divingDeeper/importers/progressiveglTFLoad> [Stand: 28.04.2021].

Abbildungsverzeichnis

| | |
|---|----|
| 1.1. Vergleich vereinfachtes Modell (entfernt) Originalmodell (nah) | 8 |
| 2.1. Vereinfachung von Polygone als Sammlung von <i>Triangles</i> | 13 |
| 2.2. Konvertierungspipeline ohne glTF [15] | 14 |
| 2.3. Konvertierungspipeline mit glTF [15] | 15 |
| 2.4. glTF Datenstruktur [16] | 15 |
| 2.5. Modell Basis | 17 |
| 2.6. Transformation mittels <i>Edge Collapse</i> | 17 |
| 2.7. Transformation mittels <i>Halfedge Collapse</i> | 18 |
| 2.8. Transformation mittels Vertex Removal | 18 |
| 2.9. Schritte einer Realtime Rendering Pipeline | 19 |
| 2.10. Definition eines Triangles | 19 |
| 2.11. Vertexshader für die Definition eines Punktes | 20 |
| 2.12. Fragmentshader für das definieren von Farbwerten | 20 |
| 2.13. Kamera Frustum | 21 |
| 2.14. Frustum culling visualisiert. Rote Elemente werden nicht prozessiert. . . . | 21 |
| 2.15. Level Of Detail Visualisierung vier Hasen | 23 |
| 2.16. Beispiel <i>Point-Cloud</i> [17] | 25 |
| 3.1. 'LOD Group'-Komponente in Unity für die Konfiguration der Übergänge zwischen LOD-Artefakten | 28 |
| 4.1. Durch das CLI generierte Konfigurationsdatei | 34 |
| 4.2. lode <i>CLI</i> config – Konfiguriert die gewünschten LOD-Artefakten | 34 |
| 4.3. lode <i>CLI</i> run – Generiert die gewünschten LOD-Artefakten | 35 |
| 4.4. lode-ui | 35 |
| 4.5. Beispielcode zur Benutzung der LOD-Artefakte mittels <i>lode-three</i> in <i>Three.js</i> | 36 |
| 4.6. Vergleich der Texturen | 37 |
| 4.7. Vergleich Originalmodell und vereinfachtes Modell | 38 |
| 4.8. Vergleich Originalmodell und vereinfachtes Modell auf Distanz | 39 |
| 4.9. Testapplikation | 42 |
| 4.10. Durchlauf eines Benchmarktests | 44 |
| 5.1. Beispiel <i>Low-Poly</i> -Modell | 46 |

Tabellenverzeichnis

| | |
|---|----|
| 3.1. Übersicht Merkmale der verschiedenen LOD Systeme | 27 |
| 4.1. Kennzahlen für Benchmark | 41 |

A. Anhang

A.1. Aufgabenstellung



Bachelorarbeit 2021

Marc Berli, Simon Stucki

Level-of-Detail Generierungs-System für 3D- Webapplikationen

Ausgangslage

WebGL wird mittlerweile in allen modernen Browsern unterstützt und bietet die Möglichkeit, interaktive 3D Applikationen im Web zu entwickeln. Das Toolset ist jedoch im Vergleich zu anderen Engines wie Unity oder Unreal merkbar weniger ausgereift. Unter anderem gibt es zurzeit kein solides System zur automatischen Generierung von «Level-Of-Detail» (im Folgenden LOD) Artefakten.

Ziel der Arbeit

Im Rahmen dieser Arbeit soll ein Werkzeug entwickelt werden, welches für ein gegebenes 3D-Modell automatisch LODs generiert.

Aufgaben

1. Zusammenstellen der Grundlagen zum Thema «Level-Of-Detail» in 3D-Webapplikationen.
2. Analyse und Recherche zum Stand der Entwicklung in diesem Bereich.
3. Definition eines Benchmarks für LOD unter Berücksichtigung verschiedener Aspekte (Download-Grösse, Laufzeitperformanz).
4. Entwicklung eines oder mehrerer Algorithmen mit Behandlung von Spezialfällen (Loose oder Thin Shapes).
5. Definition eines LOD-Formats.
6. Entwicklung geeigneter Werkzeuge, z.B. für Babylon.js, three.js, Webpack, oder Kommandozeilen-Tools.

Abgabetermin

Abgabetermin für die Arbeit ist Freitag, 11. Juni 2021.

Bewertungskriterien

Bei der Bewertung werden die folgenden Kriterien – zu je einem Drittel gewichtet – zugrunde gelegt:

- Projektverlauf, Leistung, Arbeitsverhalten
- Qualität der Ergebnisse
- Form und Inhalt des Berichts und der Präsentation

Detailliertere Angaben sind im Dokument *Bewertungsraster zur Projekt- und Bachelorarbeit an der SoE* zu finden.

Winterthur, 15. Februar 2021
Gerrit Burkert