



School of Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit (Informatik)

Level of Detail Toolset für 3D- Web-Applikationen

Autoren

Marc Berli
Simon Stucki

Hauptbetreuung

Dr. Gerrit Burkert

Datum

11.06.2021

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

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbstständig 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 Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

Schlieren, 11. Juni 2021

Name Studierende:

Simon Stucki _____

Schlieren, 11. Juni 2021

Marc Berli _____

Zusammenfassung

Die Web-Plattform 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 Laufzeitleistung aufgezeigt, indem mehrere Detailstufen, sogenannte *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.

In dieser Arbeit werden verschiedene Ansätze zum Einsatz von *LOD*-Artefakten, sowie ihre Vor- und Nachteile aufgezeigt. Des Weiteren werden Algorithmen für die automatisierte Vereinfachung von polygonalen Modellen analysiert und der gewählte Algorithmus implementiert.

Das Resultat ist ein Toolset, welches es ermöglicht, mit geringem manuellen Aufwand *LOD*-Artefakte zu generieren und in 3D-Web-Applikationen einzusetzen. Es fügt 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 Laufzeitleistung von gewissen 3D-Web-Applikationen verbessern kann.

Abstract

The web platform allows 3D visualizations to be made available to a wide audience. Due to a large number of devices on which such an application can be used, performance optimization is indispensable. In this work, an option to improve runtime performance is shown by using multiple levels of detail (*LOD*) of a model which are generated and displayed depending on the distance of the virtual camera. Currently, considerable manual effort is required to generate and integrate such artifacts on the web. In this paper, different approaches for generating *LOD* artifacts, as well as their advantages and disadvantages, are highlighted. Furthermore, algorithms for automated simplification of polygonal models are analyzed and the chosen algorithm is implemented. The result is a toolset that allows to generate *LOD* artifacts with little manual effort and to use them in 3D web applications. It seamlessly integrates into the developer's workflow and provides a simple interactive configuration option. Utilizing a benchmark, it is proven that the use of *LOD* can improve the runtime performance of certain 3D web applications.

Vorwort

Die Autoren der Bachelorarbeit sind seit mehreren Jahren in der Web-Entwicklung tätig. Der Fortschritt in der Web-Entwicklung ist hauptsächlich dem Engagement der vielen unabhängigen Entwickler zu verdanken. Insbesondere für 3D-Visualisierungen sind in den vergangenen Jahren viele neue Möglichkeiten entstanden. Die Autoren teilen die Begeisterung für 3D-Visualisierungen. Aufgrund der Neuartigkeit sind die Tools noch nicht auf dem Level der anderen Plattformen. Diese Arbeit stellt insofern einen Versuch dar, soweit dies in einem Semester möglich ist, etwas an die Community zurückzugeben und die Möglichkeiten im Web in Bezug auf 3D-Applikationen zu erweitern und zu erleichtern.

Die Performanz in Web-Applikationen hinkt derer nativer Applikationen hinterher. Jede Verbesserung in dieser Hinsicht ist wertvoll. Die sogenannten *Level of Details (LOD)* sind eine Methode, um die Performanz zu verbessern. Es gibt bereits unterschiedliche Ansätze, diese sind jedoch noch nicht ausreichend ausgereift, um in einer produktiven Applikation eingesetzt werden zu können.

Für die Autoren der Arbeit ist die Reduktion der manuellen Arbeitsschritte, welche bei der Entwicklung von 3D-Applikationen notwendig sind, ein besonderes Anliegen. Ziel der Arbeit ist es somit, Prozesse, welche sich in anderen Teilen der Web-Entwicklung 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 mit wertvollen Hinweisen unterstützt hat.

Inhalt

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	11
2. Theoretische Grundlagen	12
2.1. 3D-Modelle	12
2.2. Transformation von Modellen	17
2.3. Grafik-Pipeline	19
2.4. Performanzoptimierung	21
2.5. Einführung Level of Detail	23
2.5.1. Ansätze für <i>LOD</i> Artefakte	23
2.5.2. Vergleich Algorithmen	25
3. Vorgehen	27
3.1. Vergleich LOD-Systeme	27
3.1.1. Bestehende Systeme	27
3.1.2. Kriterien	29
3.1.3. Vergleich Downloadgrösse	29
3.1.4. Auswirkung auf Laufzeitverhalten	29
3.1.5. Auswirkung auf Scene Graph	29
3.1.6. Integration in bestehende Arbeitsabläufe	29
3.1.7. Visual Pop	30
3.1.8. Entscheidung	30
3.2. Surface Simplification Algorithmus	30
3.2.1. Grobablauf	31
3.2.2. Fehlermetrik	31
3.2.3. Optionen für Edge Collapse	31
3.2.4. Edge Collapses durchführen	31
3.2.5. Referenzimplementation	32
3.3. Toolset	32
3.4. Nutzen LOD	32
3.4.1. Aspekte für Nutzen	32
3.4.2. Vergleichbare Arbeiten	34

3.4.3. Mögliche Ansätze	34
4. Resultate	36
4.1. LOD-Toolset	36
4.1.1. Aktueller Workflow	36
4.1.2. lode-Toolset	37
4.1.3. Ablauf von lode	38
4.2. LOD-Generierung	41
4.2.1. Implementation	41
4.2.2. Artefakte	41
4.2.3. Fehlermetrik	41
4.2.4. Texturen	41
4.2.5. Vergleich	42
4.2.6. Generierung von glTF	43
4.2.7. Iterativer Ansatz	44
4.2.8. Edge Collapse	45
4.3. Benchmark	45
4.3.1. Browser-Umgebung	45
4.3.2. Testaufbau	46
4.3.3. Auswertung	49
5. Diskussion und Ausblick	51
5.1. LOD System	51
5.1.1. Allgemeingültigkeit	51
5.1.2. Anwendbarkeit	51
5.1.3. Abgrenzung	52
5.2. Mögliche Erweiterungen	52
6. Verzeichnisse	55
6.1. Glossar	55
6.2. Literaturverzeichnis	57
6.3. Abbildungsverzeichnis	61
6.4. Tabellenverzeichnis	62
A. Anhang	63
A.1. Aufgabenstellung	63
A.2. Packages	64
A.3. Benchmark Daten	74
A.4. Projektdokumentation	76
A.4.1. Zeitplan	76
A.4.2. Meeting-Protokolle	77

1. Einleitung

3D-Visualisierungen werden in vielen Branchen verwendet. Virtual Reality, CAD-Programme oder Computerspiele sind einige bekannte Anwendungsbereiche. Dank leistungsfähigerer Geräten sind realistischere Visualisierungen möglich.

Anwendungen können auf spezifische Hardware, wie zum Beispiel die Spielkonsole PlayStation, ausgerichtet sein. Es gibt Unterschiede zwischen den Konsolen, grundsätzlich sind die Variationen der Leistungsfähigkeit innerhalb einer Konsolengeneration klein. In anderen Anwendungsfällen kann eine Applikation sogar für nur eine spezifische Hardware entwickelt werden. Dies erlaubt es, eine Applikation auf die Anforderungen dieser Hardware zuzuschneiden. Der primäre Nachteil ist 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 Anwendungsbereiche ist dies jedoch ein sekundäres Problem.

Für hardwareunabhängige Anwendungen eignet sich die Web-Plattform hervorragend. Immer mehr Benutzer haben Zugang zu einem Desktop, Tablet, Mobiltelefon oder zu einem anderen 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 dieser Arbeit wird der Fokus auf die Web-Plattform gelegt. Die Grundlagen sind jedoch auch in anderen Entwicklungsumgebungen 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 zu implementieren. Da diese Applikationen rechenintensiv sind und gerade mobile Geräte in ihrer Rechenleistung beschränkt sind, ist Performanzoptimierung 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 zum Beispiel ein Modell in grosser Distanz vereinfacht dargestellt werden (Abbildung 1.1a), solange bei genauer Betrachtung mehr Details sichtbar werden (Abbildung 1.1b).

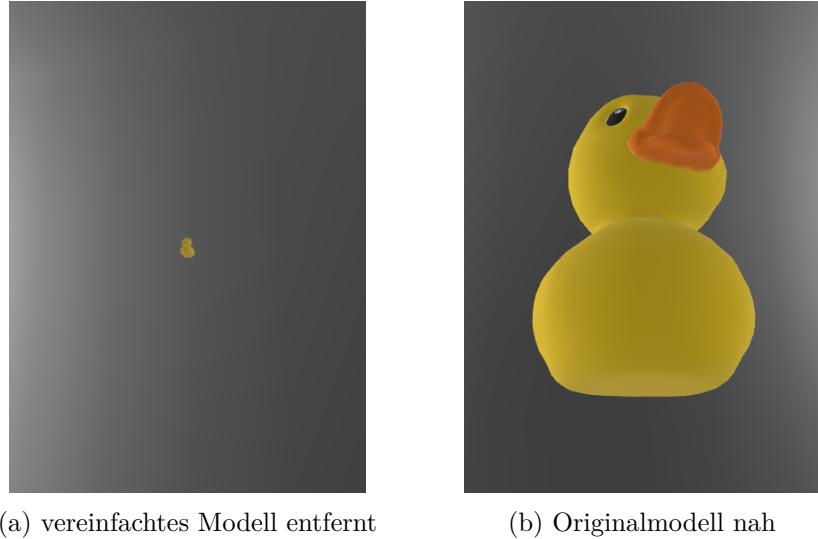


Abbildung 1.1.: Vergleich vereinfachtes Modell (entfernt) Originalmodell (nah)

In diversen *Rendering Engines*¹ gibt es deshalb Optionen für das Verwenden von so genannten *Level of Detail (LOD)* Artefakten. Für *Game Engines*² wie *Unreal Engine* oder *Unity* gibt es bewährte Tools, um den Einsatz von *LOD*-Artefakten zu vereinfachen. Zurzeit gibt es in der Web-Entwicklung 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]. Als Alternative wird zurzeit ein weiterer Standard entwickelt: *WebGPU*. Dieser ist aktuell noch in Entwicklung und wird deshalb nicht weiter berücksichtigt, auch wenn ein grosses Potenzial vorhanden ist [3]. Die Unabhängigkeit der Hardware bedeutet, dass Optimierung der Performanz in Web-Anwendungen notwendig ist, um allen Benutzern ein optimales Erlebnis zu ermöglichen. Im Vergleich zu fixen Hardwareanwendungen ist es realistisch, dass eine Web-Anwendung sowohl auf einem leistungsfähigen Desktop-Computer als auch auf einem günstigen Mobilgerät verwendet wird.

¹Programm, das zuständig für die Darstellung von 3D-Grafiken ist

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

Zudem ist *WebGL* eine junge Technologie und wurde erst 2011 veröffentlicht [2] – verglichen mit dem initialen Veröffentlichungsdatum von *OpenGL*³, welches im Jahre 1992 publiziert wurde [4]. Nicht nur das Alter, sondern auch die Natur der Web-Plattform hat dazu beigetragen, dass *WebGL* ein langsames Wachstum verspürt hat. Um einen Web-standard 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 hatte zum Beispiel der *Internet Explorer 10* keinen Support und es wurde deshalb 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 *High Level JavaScript* Bibliotheken. Die Bekanntesten werden hier kurz erwähnt. Als Indikator für die Populärität wurden die wöchentlichen Downloads auf *npm*⁴ verwendet. Wichtig ist, dass dies alleine keinen verlässlichen Indikator darstellt – für eine kurze Übersicht jedoch ausreichend geeignet ist.

Three.js

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

Babylon.js

Eine weitere offene Bibliothek ist *Babylon.js*, welche die Entwicklung von 3D-Applikationen vereinfacht. Sie zeichnet sich vor allem durch einen grossen Funktionsumfang aus [9]. So verfügt sie zum Beispiel über integrierte Erweiterungen für verschiedene *Physics Engines* und erleichtert somit den Einsatz von anspruchsvollen Features. Ausserdem setzt es auf *Semantic Versioning*. Im Vergleich zu *Three.js* verfügt *Babylon.js* über eine kleine Community. Aufgrund dessen werden Erweiterungen für populäre Bibliotheken tendenziell weniger aktiv entwickelt.

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

⁴*Node Package Manager*, Datenbank von *JavaScript* Paketen und Paketmanager.

⁵Konvention für Veröffentlichung von Software Paketen, siehe: semver.org [8]

PlayCanvas

PlayCanvas ist eine offene 3D-Engine, welche primär für das Web entwickelt wurde. Als Erweiterung wird zudem ein proprietärer *Cloud Service* angeboten [10]. Die Community ist – im Vergleich zur direkten Konkurrenz, wie zum Beispiel *Three.js* – klein. *PlayCanvas* verfolgt jedoch einen anderen Ansatz und liefert zum Beispiel eine standardmässige Integration für eine *Physics 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 [11]. *Unity* bietet zwar den *Sourcecode* für Teile der *Engine* offen zur Verfügung, die Lizenz verbietet es jedoch den Code weiterzuverwenden [12]. Zudem bietet *Unity* keine zu *Three.js*, *Babylon.js* oder *PlayCanvas* vergleichbare Integration für die Web-Entwicklung an. *Unity* Projekte müssen direkt im *Unity Editor* entwickelt werden. Dies erschwert den Einsatz von *Best-Practices*, deshalb wird *Unity* 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 im 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 Web-Applikationen 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 leistungsstarken Geräten mehrere Sekunden [13]. *Three.js* verfügt über eine vergleichbare Möglichkeit, welche dieselben Limitationen aufweist [14]. Möchte man die Artefakte vorzeitig generieren, so ist es notwendig, diese Schritte manuell durchzuführen. Bei Modellanpassungen entsteht somit ein wiederkehrender manueller Aufwand. Ein Tool für die Konfiguration der Artefakte, wie dies bei *Unreal* oder *Unity* der Fall ist, steht nicht zur Verfügung.

Des Weiteren sind die Systeme nicht untereinander kompatibel. So kann das System zur Modellgenerierung von *Babylon.js* nicht in *Three.js* verwendet werden, obwohl die Problemstellung dies erlaubt. Folglich divergieren die Systeme und Weiterentwicklungen müssen für jedes System neu implementiert werden.

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 signifikant 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, indem ein Konfigurationstool bereitgestellt wird, mit dem man Konfigurationseinstellungen für die generierten Modelle 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. Zudem muss 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 Grundlagen für die Entwicklung eines *LOD*-Systems aufgezeigt. Zuerst werden die Grundlagen der polygonalen 3D-Modelle und verschiedene Dateiformate erläutert. Danach wird auf die Theorie der Transformationen und die Grafik-Pipeline eingegangen. Zum Schluss werden noch alternative, beziehungsweise ergänzende, Optionen zur Performanzoptimierung beschrieben.

2.1. 3D-Modelle

Ein Modell stellt ein physisches Objekt, häufig aus der realen Welt, vereinfacht dar. 3D-Modelle können 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 kann man einen *Vertex* am Ursprung eines Koordinatensystems definieren als:

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

Eine Sammlung von drei *Vertices* bildet ein *Triangle*¹ (Dreieck). Ein *Triangle* wird somit wie folgt definiert:

$$T = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

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:

$$P = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} \cong T_1 \cup T_2 = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} \cup \begin{bmatrix} V_1 \\ V_2 \\ V_4 \end{bmatrix}$$

¹Wir verwenden die in englischer Sprache verfassten Fachliteratur zu 3D-Anwendungen gebräuchlichen Ausdrücke.

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



(a) Polygon



(b) Polygon als Sammlung von *Triangles*

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

Weitere Attribute

Neben den geometrischen Attributen verfügt ein Modell über weitere Attribute, welche zum Beispiel die visuellen Aspekte definieren.

Normals 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 diesem Fall ist er identisch zur aus der Geometrie bekannten Normale. *Normals* werden häufig für das Berechnen von Reflexionen verwendet. *Normals* werden auch für gewisse Performanzoptimierungen eingesetzt, dazu mehr in Absatz 2.4.

Texturen Um die Oberfläche von Modellen zu definieren, wird häufig ein *Texture Mapping*, auch *UV-Mapping* genannt, durchgeführt. Bei diesem Verfahren wird definiert, wie eine 2D-Grafik auf ein 3D-Modell abzubilden ist. Dafür werden für jeden *Vertex* zwei Koordinaten, u und v , definiert. Diese Koordinaten bestimmen die Position in der 2D-Grafik. In der Praxis finden weitere Methoden Anwendung, auf diese wird hier im Sinne der Übersichtlichkeit 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.

Formate

Um ein 3D-Modell in einer Anwendung einzusetzen, 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 einen limitierten Funktionsumfang. Des Weiteren gibt es keine zentrale Instanz, welche eine Spezifikation liefert. Deshalb sind Informationen schwerer zu finden. [15]

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 wird von der *Khronos Group* entwickelt [16]. In dieser Arbeit wird es als Austauschformat verwendet, deshalb wird im Folgenden genauer auf *glTF* 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 Laufzeitumgebung optimiert sind. Deswegen braucht jedes Ausgangsformat jeweils einen Konverter pro Eingangsformat, wie in Abbildung 2.2 ersichtlich.

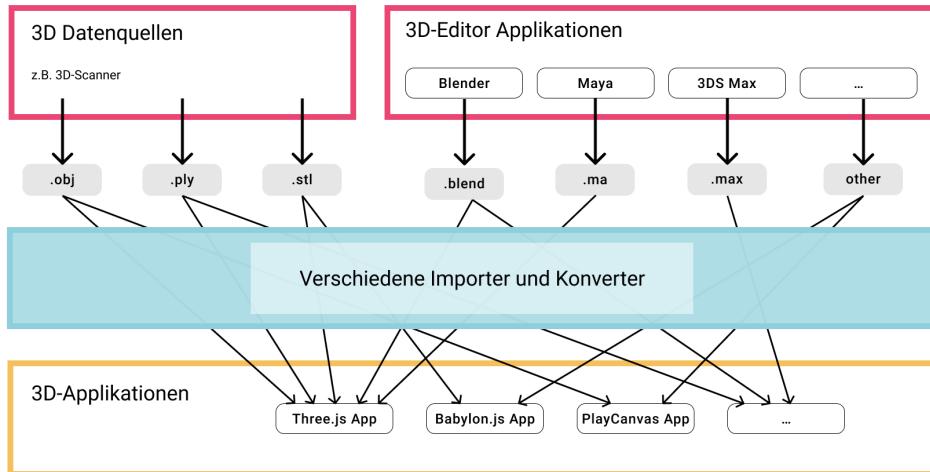


Abbildung 2.2.: Konvertierungspipeline ohne *glTF* [17]

2. Theoretische Grundlagen

Durch die immer grösser 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 Konverter vermieden werden und nur wenige, sehr spezifische Dateiformate benötigen noch einen einzigen Konverter. Diese neue Pipeline ist in Abbildung 2.3 dargestellt. [17]



Abbildung 2.3.: Konvertierungspipeline mit *glTF* [17]

Die Basisstruktur von *glTF* basiert auf *JSON*². Darin ist die komplette Szenerie des Modells beschrieben. Der Aufbau dieser Struktur ist in Abbildung 2.4 aufgezeigt [18].



Abbildung 2.4.: *glTF* Datenstruktur [18]

Die folgende Auflistung erläutert die Elemente dieser Struktur:

- **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 weitere *Child Nodes* referenzieren.
- **Mesh:** Beschreibt ein geometrisches Objekt, bestehend aus mehreren *Primitives*. Ein *Primitive* 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 *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 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*.

²JavaScript Object Notation, Austauschformat, welches häufig in der Web-Entwicklung eingesetzt wird

2.2. Transformation von Modellen

Um ein Modell vereinfachen zu können, muss es verändert werden. Diese Veränderungen können in 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.

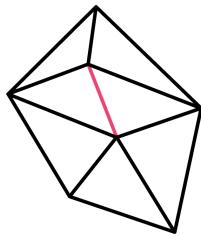
Für die folgenden Beispiele wird jeweils das Modell aus Abbildung 2.5 als Ausgangslage verwendet.



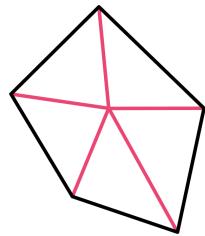
Abbildung 2.5.: Originalmodell

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) Originalmodell mit für *Edge Collapse* gewählter *Edge* in Rot

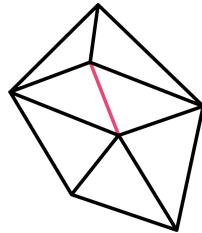


(b) Modell nach *Edge Collapse* mit veränderten *Edges* in Rot

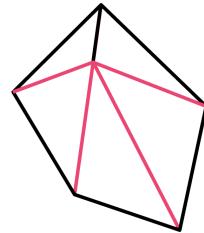
Abbildung 2.6.: Transformation mittels *Edge Collapse*

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) Originalmodell mit für *Halfedge Collapse* gewählter *Edge* in Rot

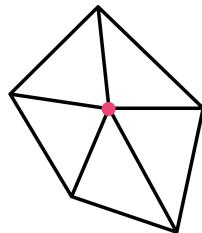


(b) Modell nach *Halfedge Collapse* mit veränderten *Edges* in Rot

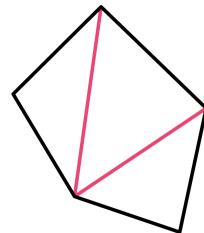
Abbildung 2.7.: Transformation mittels *Halfedge Collapse*

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. Eine mögliche Triangulation des Polygons ist in Abbildung 2.8b ersichtlich.



(a) Originalmodell mit für *Vertex Removal* gewähltem *Vertex* in Rot



(b) Modell nach *Vertex Removal* mit veränderten *Vertices* in Rot

Abbildung 2.8.: Transformation mittels *Vertex Removal*

2.3. Grafik-Pipeline

Die Grafik-Pipeline ist zuständig um eine definierte Szene auf einem Ausgabegerät zu visualisieren. Die Schritte werden im folgenden Abschnitt nur oberflächlich erläutert, um ein grobes Verständnis zu vermitteln. Es wird dabei ein simples 3D-Modell auf einem 2D-Display, wie zum Beispiel einem Monitor, visualisiert. Die Schritte sind in Abbildung 2.9 aufgezeigt. Der Prozess wird 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. Das *Triangle* wird anschliessend mithilfe dreier *Indices* definiert. Diese *Indices* verweisen auf die entsprechenden *Vertices*. So entspricht der Eintrag 0 dem *Vertex* an der Position V_0 . Wobei V_0 definiert ist als:

$$V_0 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$$

```

1 const vertices = new Float32Array([
2   -1, 1, 0, // vertex 0
3   -1, -1, 0, // vertex 1
4   1, -1, 0 // vertex 2
5 ]);
6
7 const indices = new Uint16Array(
8   [0, 1, 2] // triangle 0
9 );
  
```

Abbildung 2.10.: Definition eines Triangles

Anschliessend werden die Daten in einem *Draw Call* an die *GPU* gesendet. 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* grosse Datenmengen parallel prozessieren kann und sich die Unterbrechung durch einzelne Aufrufe negativer auswirkt.

Vertexshader

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 *Vertexshader* ist in Abbildung 2.11 ersichtlich. Der Code, welcher in *GLSL*³ geschrieben ist, wird für jeden *Vertex* ausgeführt. Der aktuelle *Vertex* ist dabei in der Variable *coordinates* abgespeichert. *gl_Position* ist eine spezielle Variable, welche von *GLSL* zur Verfügung gestellt wird und den Endpunkt beinhalten soll. Das gegebene Beispiel ist trivial, in der Praxis werden hier häufig die Matrixtransformationen unter anderem für die Kameraperspektive durchgeführt.

```
1 attribute vec3 coordinates;
2
3 void main(void) {
4     gl_Position = vec4(coordinates, 1.0);
5 }
```

Abbildung 2.11.: *Vertexshader* für die Definition eines Punktes

Fragmentshader

Durch die Rasterisierung werden kontinuierliche Objekte zu diskreten Fragmenten verarbeitet. Ein Fragment entspricht grundsätzlich einem Pixel. 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. Bei *gl_FragColor* handelt es sich wiederum um eine Variable von *GLSL*. Der zugewiesene Wert entspricht der Farbe des Pixels. In diesem Fall ist die Farbe Schwarz. Die Reihenfolge ist *rgba*: rot, grün, blau, alpha (Transparenz).

```
1 void main(void) {
2     gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
3 }
```

Abbildung 2.12.: *Fragmentshader* für das definieren von Farbwerten

³*OpenGL Shading Language*, C-basierte Programmiersprache für das Programmieren von *GPU*-Prozessen

Bildschirm

Zum 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 Laufzeitleistung 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 Kamera 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 2.13a zu sehen ist, beginnt das Kamera *Frustum* in der Nähe der Kamera und dehnt sich ein wenig aus bis es gegen Ende des Spektrums beschränkt wird. Polygone, welche nicht im Kamera *Frustum* enthalten sind, werden bei dieser Methode, wie in Abbildung 2.13b gezeigt, nicht weiter prozessiert. Dies reduziert die Anzahl Polygone drastisch.



(a) Kamera *Frustum*



(b) *Frustum Culling* visualisiert. Rote Elemente werden nicht prozessiert.

Abbildung 2.13.: *Frustum Culling*

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 Polygone, welche gerendert werden müssen, reduziert.

Backface Culling

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

Impostors

Vergleichbar zu *LOD*-Artefakten sind sogenannte *Impostors*. 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. Schwierigkeiten dieser Methode sind Modelle, welche spezielle Materialien aufweisen. So ist zum Beispiel das Abbilden von Reflexionen, wie dies bei metallischen Oberflächen der Fall ist, eine Herausforderung [19].

Parallel Rendering

Auch bekannt unter *Distributed Rendering* ist der Einsatz von Techniken aus dem *Parallel Programming* in Visualisierungsanwendungen. So kann die Aufgabe auf verschiedene *GPUs* verteilt parallel verarbeitet werden. Eine Problemstellung hierbei ist jedoch, dass das sinnvolle Aufteilen der Arbeit nicht trivial ist. So eignet sich dieser Ansatz jedoch gut für Anwendungen im Bereich von *Virtual Reality*, da dort zwei separate Bilder generiert werden müssen und somit leicht zu separieren sind [20].

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 dreidimensionalen Raum, kann dies zu seltsamen Effekten führen, wenn sich die Kamera bewegt. Diese Technik kann deshalb nur in sehr spezifischen Anwendungsfällen eingesetzt werden. Die *Street View* Funktion von *Google Maps* ist ein bekanntes Tool, welches auf *Image-based Rendering* setzt. Beim Bewegen durch die virtuelle Welt in *Street View*, ist zu sehen, wie sich die Bilder austauschen [21].

2.5. Einführung Level of Detail

Als *Level of Detail* werden die verschiedenen Detailstufen bei der virtuellen Darstellung bezeichnet. Dies wird verwendet, um die Laufzeitleistung von Anwendungen zu steigern, indem Objekte im Nahbereich detailliert angezeigt werden; wohingegen Elemente im Fernbereich deutlich vereinfacht dargestellt werden. So muss der *Vertexshader* weniger *Vertices* prozessieren.

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 2.14.: *Level of Detail* Visualisierung vier Hasen [22]

Wie in der Abbildung 2.14 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 *Triangles*, wird es bereits im ersten Schritt auf 2'502 *Triangles* reduziert. Dies ist eine enorme Reduktion von ca. 96.5 %. Im dritten Schritt wird die Anzahl *Triangles* wiederum um ca. 90 % auf 251 reduziert. Schlussendlich hat das letzte Objekt noch 76 *Triangles*, 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 die verschiedenen Systeme detailliert erläutert, so wie ihre Vor- und Nachteile aufgezeigt.

Diskrete LOD (DLOD)

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*⁴ 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

⁴Datenstruktur, welche die räumliche Anordnung der Modelle einer 3D-Szenerie beschreibt

mit einer einfacheren Version ausgetauscht wird. Dieses Verhalten wird *Visual Popping* genannt. *Visual Pop* bezeichnet das merkbare Austauschen zweier *LOD*-Artefakte. Je subtiler die Änderungen an den Artefakten und je grösser die Distanz zur Kamera ist, desto weniger ist das Austauschen bemerkbar. Es ist zudem häufig nicht möglich grössere Modelle sinnvoll zu vereinfachen. Grosse Modelle zeichnen sich dadurch aus, dass Teile davon nah sind, während andere Teile fern sind. Die nahen Teile des Modells sollten detailliert angezeigt werden, wohingegen die entfernten Elemente vereinfacht dargestellt werden sollen. So ist der Ansatz zum Beispiel für Terrain ungeeignet. Gleichermaßen ist es auch nicht möglich, viele sehr kleine Modelle zu kombinieren, da jedes Modell unabhängig ist. Das Kombinieren von mehreren Modellen wird *Clustering* genannt.

Kontinuierliche LOD (CLOD)

Bei kontinuierlichen *LOD*-Systemen werden Transformationen 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 Laufzeitleistung der Applikation hat. Die verschiedenen Vereinfachungen können zwar vorgängig berechnet werden, das Applizieren der Vereinfachungen und die damit verbundene Interpolation muss jedoch zur Laufzeit durchgeführt werden. Das Problem des *Clusterings* ist auch für diese Variante nicht gelöst.

Hierarchische LOD (HLOD)

Bei *HLOD* werden mehrere Objekte in einen *Cluster* gruppiert. Diese *Cluster* können mithilfe des *Scene Graphs* definiert werden. Hierarchische Systeme erlauben 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. Im folgenden Abschnitt wird auf die Grundidee einiger Kategorien eingegangen, um einen groben Überblick zu verschaffen. Für mehr Informationen zu weiteren Ansätzen, siehe *Quadric-Based Polygonal Surface Simplification* Sektion 2.4 [23].

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 *Vertices* innerhalb einer Gitterzelle werden dann in einen einzigen *Vertex* zusammengelegt. Die Qualität der visuellen Annäherung dieser Algorithmen stellt bei drastischen Vereinfachungen von polygonalen Modellen jedoch häufig ein Problem dar. Sogenannte *Point Clouds* können mit solchen Algorithmen schnell vereinfacht werden. *Point Clouds* sind *Vertices* im dreidimensionalen Raum ohne zugehörige *Triangle*, also ohne Fläche (in Abbildung 2.15 exemplarisch zu sehen). Diese Daten können das Resultat eines 3D-Scanners sein.

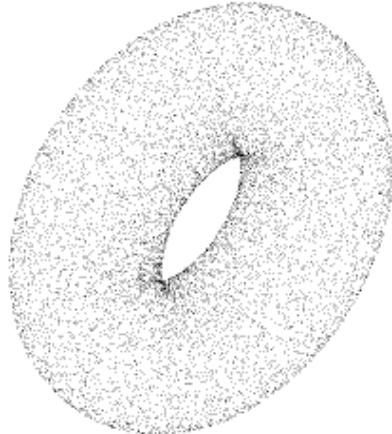


Abbildung 2.15.: Beispiel eines Torus visualisiert als *Point Cloud* [24]

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 ein *Halfedge Collapse* durchgeführt 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 [25]. 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 definiert. In diesem Schritt wird erläutert, welche Art *LOD*-System es bereits gibt und welches in dieser Arbeit entwickelt werden soll.

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 Artefakte automatisch generieren kann. Dies ermöglicht es, komplexe Strukturen direkt innerhalb von *Unreal* zu kombinieren [26]. So können einige Faktoren angepasst werden, um die *LOD* den Anforderungen entsprechend zu gestalten. Das Ergebnis kann direkt in einer Vorschau überprüft werden, wie in Abbildung 3.1. So wird es leicht gemacht, Feinjustierungen vorzunehmen. Der Moment für die Levelwahl kann eingestellt werden. *Unreal* bietet eine automatische Option, so wie auch eine manuelle, wobei für jedes Level der Triggerpunkt gewählt werden kann. *Unreal* lässt Benutzer den Triggerpunkt über die relative Grösse eines Modells zur Fenstergrösse einstellen. Dies hat zwei wesentliche Vorteile: Einerseits können so Modelle in verschiedenen Grössen verwendet werden, welche alle individuelle Triggerzeitpunkte haben. Andererseits ist diese Grösse einfach zu testen und benötigt somit für die Einstellung weniger manuellen Aufwand. Der Zeitaufwand für die Integration des Systems in eine Applikation wird so signifikant reduziert und macht den Einsatz eines *LOD*-Systems attraktiver.

3. Vorgehen

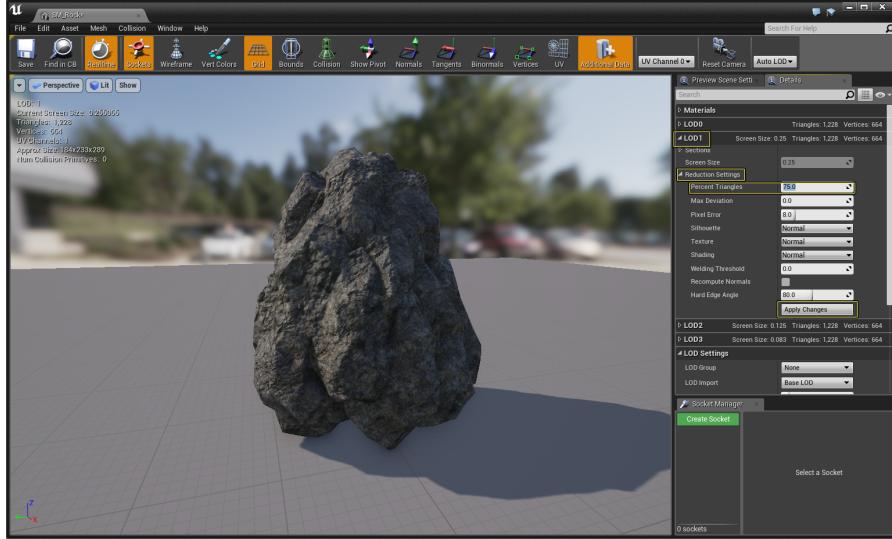


Abbildung 3.1.: Automatische *LOD*-Generierungsansicht in *Unreal* [27]

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 [28]. Ansätze für ein *HLOD*-System sind vorhanden, wurden jedoch nicht offiziell in *Unity* integriert [29]. In Abbildung 3.2 ist ersichtlich, wie innerhalb von *Unity* der Übergang zwischen zwei *LOD*-Artefakten konfiguriert werden kann. Hervorzuheben sind hier zum einen der *Fade Mode*, der das *Visual Popping* vermindert und zum anderen die Triggerpunkte, bei welchen das Level gewechselt werden soll. Dies wird in *Unity* identisch zu *Unreal* gelöst, in dem der Triggerpunkt relativ zur Fenstergrösse festgelegt wird.

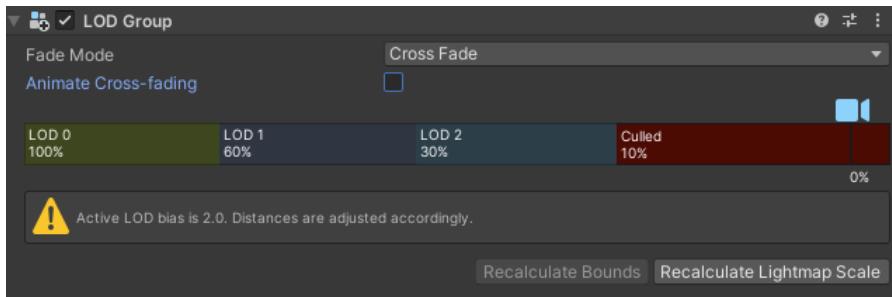


Abbildung 3.2.: 'LOD Group'-Komponente in *Unity* für die Konfiguration der Übergänge zwischen *LOD*-Artefakten

3.1.2. Kriterien

Die verschiedenen Ansätze unterscheiden sich in einigen Kriterien. Für den Einsatz in einer Web-Anwendung ist die Gewichtung der Kriterien anders als für Desktop-Anwendungen, Konsolenspiele oder dergleichen. So hat die Dateigröße der Modelle bei Web-Anwendungen einen grösseren Einfluss auf die gefühlte Performanz beim Endanwender. Insofern ist die Auswirkung der gewählten Strategie auf die Downloadgrösse ein wichtiges Kriterium.

3.1.3. Vergleich Downloadgrösse

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, also ein *Halfedge Collapse* durchgeführt wird. Trotzdem entsteht für jeden *Edge Collapse* ein zusätzlicher Speicheraufwand.

3.1.4. Auswirkung auf Laufzeitverhalten

Der Einsatz von *LOD*-Systemen hat Auswirkungen auf das Laufzeitverhalten, das Ausmass variiert jedoch. Das Laufzeitverhalten umschreibt hier primär die Arbeit, welche auf der *CPU* verrichtet werden muss, um die Szenerie adäquat vorzubereiten. Insbesondere unter Berücksichtigung der Tatsache, dass die Laufzeitleistung essenziell für Web-Anwendungen ist, wurde entschieden nicht auf kontinuierliche *LOD*-Systeme zu setzen.

3.1.5. Auswirkung auf Scene Graph

Beim Einsatz von *LOD*-Systemen sind Anpassungen am *Scene Graph* notwendig. Es wird grundsätzlich zwischen lokalen und globalen Anpassungen unterschieden. Bei lokalen Anpassungen wird lediglich der Graph des vereinfachten Modells angepasst. Der Rest der Szenerie bleibt unberührt. Bei globalen Anpassungen müssen weiter oben im *Scene Graph* Anpassungen vorgenommen werden.

3.1.6. Integration in bestehende Arbeitsabläufe

Bei diskreten und kontinuierlichen *LOD*-Systemen ist es möglich, lediglich marginal in die Arbeitsabläufe einzugreifen. Dies bedeutet, dass keine signifikanten Anpassungen an den Abläufen vorgenommen werden müssen, sondern dass es möglich ist, die Abläufe zu erweitern. Bei hierarchischen Systemen ist dies nicht gegeben, da es die Kombination mehrerer Modelle vorsieht und somit mehr Konfigurationsaufwand notwendig ist. Deshalb wurde entschieden kein hierarchisches *LOD*-System zu entwickeln, um die Integration des Tools für die Entwickler einer Web-Anwendung möglichst einfach zu gestalten.

3.2.1. Grobablauf

Der Algorithmus kann wie folgt zusammengefasst werden:

1. Initiale 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.

Die verschiedenen Schritte werden im Folgenden ausführlicher beschrieben.

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 mit dem geringsten Fehler iterativ durchgeführt werden. Ziel ist es, zuerst Transformationen durchzuführen, welche keine grossen Anpassungen an der Geometrie vornehmen.

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.

Umso weiter ein *Vertex* von der Idealposition entfernt ist, desto grösser wird der Fehler des *Vertices*. Die Idealposition ist hierbei immer die Ursprungposition.

Für mehr Informationen zu alternativen Fehlermetriken, siehe *Quadric-Based Polygonal Surface Simplification* Sektion 3.3 [23].

3.2.3. Optionen für Edge Collapse

Die offensichtlichen Optionen für einen *Edge Collapse* sind die *Edges* aller *Triangles*. Wird eine *Edge* entfernt, so wird die Fehlermetrik der beiden *Vertices* kombiniert. Es ist zudem möglich, *Vertices*, welche nahe zusammen sind ebenfalls als Optionen für *Edge Collapse* zu markieren. Hierdurch können unabhängige Teile des Modells kombiniert werden. Dies kann die Oberfläche jedoch stärker verändern und ist nicht in jedem Anwendungsfall gewünscht.

3.2.4. Edge Collapses durchführen

Die *Edge Collapses* mit dem geringsten Fehler werden iterativ durchgeführt. Die Fehlermetriken der beiden *Vertices* werden addiert und bilden die Fehlermetrik des neuen *Vertex*. Bei Garland werden die Optionen sortiert. So kann sichergestellt werden, dass immer die mathematisch korrekte Reihenfolge der Operationen durchgeführt wird.

Grundsätzlich ist es jedoch auch möglich die Optimierungen ohne sortieren durchzuführen, wie die Referenzimplementation aufzeigt. Der Vorteil ohne Sortierung liegt im Geschwindigkeitsgewinn und der Nachteil der Präzision ist vernachlässigbar.

3.2.5. Referenzimplementation

Der gewählte Algorithmus wurde bereits mehrfach implementiert. Eine wiederverwendbare und aktiv weiterentwickelte Implementation basierend auf einer formatunabhängigen Datenstruktur ist jedoch nicht vorhanden. Eine oft referenzierte Implementation ist diejenige von S. Forstmann [33], welche einige Optimierungen am ursprünglichen Algorithmus vornimmt, um optimale Performanz zu erhalten. Diese Optimierungen eignen sich hervorragend für eine webbasierte Implementation. Die Implementation ist jedoch in C++ geschrieben und unterstützt nur 3D-Modelle im *OBJ* Format. Somit kann sie nicht wiederverwendet werden.

3.3. Toolset

Das Toolset soll in wiederverwendbare Pakete aufgeteilt werden. Da es in der Web-Entwicklung üblich ist, über ein *CLI*¹ Arbeitsschritte zu automatisieren, soll ein *CLI* erstellt und auf *npm* öffentlich zugänglich gemacht werden. Dieses *CLI* soll nahtlos in den Entwicklungsablauf integriert werden können. 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 Einstellungen für den Algorithmus in dieser Konfiguration definiert. Wichtig ist der Fokus auf die Bedienbarkeit des Toolsets, hierfür ist eine ausreichende Dokumentation, sowie gut gewählte Standardeinstellungen entscheidend. 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 Laufzeitleistung der Applikation wichtig, da sie die Qualität der Software beeinflussen.

3.4.1. Aspekte für Nutzen

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

¹ *Command Line Interface* – Kommandozeile. Computer-Mensch-Schnittstelle, welche ermöglicht, durch Eingabe von Befehlwörter Anweisungen an die Maschine zu geben

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 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 benötigt. Dies hat direkte Auswirkungen auf die Zeit, bis eine Anwendung benutzt werden kann. Somit sollte die Downloadgrösse möglichst tief gehalten werden. Dank den heutigen Bandbreiten in der Schweiz ist dies für viele Anwendungsbereiche ein kleiner werdendes Problem. 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.

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* [34].

²Bilder pro Sekunde. Die Anzahl der Render-Durchläufe der *Rendering Engine*, die sich darauf auswirkt, wie flüssig eine Applikation läuft. Angestrebt werden in der Regel 60 *FPS*, was eine sehr flüssige Bewegung garantiert

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 [35].

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 eine Achse 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.

3. Vorgehen

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 ein offenes Gelände bewegt.

Auswahl

Im Vergleich zur Modellbetrachtung ist die Aussagekraft einer Szeneriebetrachtung ausreichend, um einen Beleg für den positiven Nutzen des Systems zu haben. Bei der Modellbetrachtung ist es nicht möglich den praktischen Nutzen unter Berücksichtigung verschiedener Aspekte zu beurteilen, sie dient ausschliesslich zum theoretischen Vergleich. Deshalb werden die Auswirkungen auf das Laufzeitverhalten mithilfe einer Szeneriebetrachtung gemessen. Sollte es nicht möglich sein, die *FPS* zu steigern, so liefert der Einsatz von *LOD*-Artefakten keinen nachweisbaren Nutzen.

4. Resultate

Das Resultat besteht aus unterschiedlichen, jedoch zusammenhängenden Artefakten. In diesem Abschnitt wird konkret auf die Resultate der verschiedenen Schritte eingegangen und der Zusammenhang hergestellt.

4.1. LOD-Toolset

Der Kern der Arbeit stellt das *LOD-Toolset*, *lode* genannt, dar. Dieses Tool liefert eine für das Web optimierte Möglichkeit, *LOD*-Artefakte für eine Vielzahl von Anwendungsbereichen zu generieren. Sämtliche Teile der Arbeit, insbesondere der Quellcode, stehen offen zur Verfügung und können online eingesehen werden [36].

4.1.1. Aktueller Workflow

Es gibt verschiedene, manuelle Möglichkeiten, *LOD*-Artefakte in Web-Applikationen zu integrieren. Ein möglicher 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 manuell generiert. Dies kann von Hand selber gemacht werden oder mit vom Modellierungstool zur Verfügung gestellten Funktionen. In Abbildung 4.1 wurde basierend auf dem Original 4.1a ein weiteres *LOD*-Artefakt mittels dem *Decimate Modifier* erstellt. Wie in Abbildung 4.1b zu erkennen ist, bleibt hier die Texturinformation beibehalten. Dies kann gewünscht sein, führt aber auch zu grösseren Dateigrössen, was im Web möglichst vermieden werden sollte.



Abbildung 4.1.: Manuelles erstellen von *LOD*-Artefakten in Blender

Anschliessend werden die verschiedenen Stufen exportiert und manuell in die Applikation integriert. Dafür werden diese direkt im Projekt gespeichert und ins Versionskontrollsysteem (zum Beispiel *Git*) hinzugefügt.

Diese Artefakte müssen nun im Code einzeln geladen und in die Szenerie integriert werden. Wie in Codeausschnitt 4.2 im Beispiel von *Three.js* zu sehen ist, muss jedes Level manuell geladen und konfiguriert werden.

```

1 import * as THREE from "three";
2 import { GLTFLoader } from "three/examples/jsm/loaders/GLTFLoader";
3
4 const loader = new GLTFLoader();
5 const lod = new THREE.LOD();
6 const scene = new THREE.Scene();
7
8 loader.load('path/to/duck-original.gltf', (artifact) => {
9   lod.addLevel(artifact.scene, 0);
10 });
11 loader.load('path/to/duck-lod1.gltf', (artifact) => {
12   lod.addLevel(artifact.scene, 150);
13 });
14 loader.load('path/to/duck-lod2.gltf', (artifact) => {
15   lod.addLevel(artifact.scene, 300);
16 });
17
18 scene.add(lod)

```

Abbildung 4.2.: Beispielcode zur manuellen Integration der *LOD*-Artefakte in *Three.js*

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 nochmals geprüft werden. Diese Schritte können sich mehrfach wiederholen, was mühselig und zeitintensiv ist.

4.1.2. *lode*-Toolset

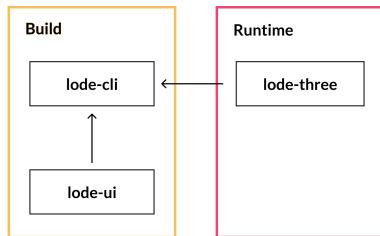
Das Ziel von *lode* ist es, für ein möglichst breites Spektrum von Anwendungsfällen eine einfache Lösung anzubieten und somit die Schwierigkeiten für den Einsatz von *LOD*-Artefakten zu reduzieren.

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.

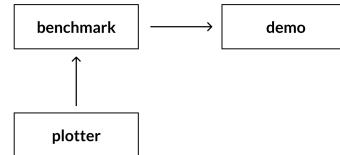
Die verschiedenen Teile sind in Abbildung 4.3 ersichtlich. Das Toolset (siehe Abbildung 4.3a) besteht aus wiederverwendbaren Paketen, welche dem Anwender zur Verfügung gestellt werden. Die Pakete werden in zwei Kategorien unterteilt: *Build*- und Laufzeitpakete. Um den Nutzen belegen zu können wurden weitere Pakete entwickelt, diese sind in Abbildung 4.3b ersichtlich. Der Benchmark ermittelt die Kennzahlen basierend auf

4. Resultate

dem Demoprojekt und im Anschluss generiert der *Plotter* Grafiken, um einen besseren Überblick zu erhalten.



(a) *lode*-Toolset, wiederverwendbare Pakete



(b) *lode*-Pakete, welche für den *Benchmark* entwickelt wurden

Abbildung 4.3.: Kategorisierung der verschiedenen *lode*-Pakete - Pfeile zeigen direkte Abhängigkeiten auf

4.1.3. Ablauf von *lode*

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 (siehe Abbildung 4.4) kann mittels *CLI*, welches auf *NPM* verfügbar ist, angelegt werden (siehe Abbildung 4.5).

```
1  {
2    "levels": [
3      {
4        "threshold": 300
5      },
6      {
7        "threshold": -1,
8        "configuration": {
9          "targetScale": 0.0625
10       }
11     }
12   ]
13 }
```

Abbildung 4.4.: Durch das CLI generierte Konfigurationsdatei

4. Resultate

```
npx @kreativwebdesign/lode-cli 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
```

Abbildung 4.5.: *lode CLI* config – Konfiguriert die gewünschten *LOD*-Artefakte

Erstellen der Artefakte

Danach generiert *lode* die in der Konfigurationsdatei beschriebenen *LOD*-Artefakte (siehe Abbildung 4.6). 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 zusätzliche manuelle Arbeit zu sehen.

```
npx @kreativwebdesign/lode-cli -w  
[L O D E]  
Preparing output folder:  
done  
Running initial LOD transformation:  
performing LOD algorithm on file assets/airplane/airplane.gltf  
done  
watching files and running lode server on port 3001...
```

Abbildung 4.6.: *lode CLI* run – Generiert die gewünschten *LOD*-Artefakte

Das zugehörige *lode-ui* (siehe Abbildung 4.7) 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. Das manuelle Wechseln zwischen Applikationen ist somit nicht notwendig. Um den Einsatz von *lode-ui* zu vereinfachen, wird das Paket direkt mit *lode-cli* zur Verfügung gestellt, es ist keine manuelle Installation des Pakets notwendig.

4. Resultate



Abbildung 4.7.: *lode-ui*, links befindet sich die Modellwahl, oben die *LOD*-Artefakte und im Hauptbereich die Vorschau

Verwendung der Artefakte in der Applikation

Die Artefakte können innerhalb der Applikation mithilfe von *lode-three* geladen werden. Dabei werden die Definitionen in der Konfigurationsdatei geladen und das Modell entsprechend in der Szenerie dargestellt, wie in Beispielcode 4.8 zu sehen ist.

```
1 import * as lodeLoader from "@kreativwebdesign/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 airplaneModel = await lodeLoader.loadModel({
10   lodeContext,
11   artifactName: 'assets/airplane',
12 })
13
14 scene.add(airplaneModel)
15 clone.position.set(0, 0, 0);
```

Abbildung 4.8.: 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.js*, umgesetzt. Grund dafür ist das Einbinden von *lode* in bestehende Entwicklungsabläufe in der Web-Entwicklung. So ist es einfach möglich, das Paket mithilfe von *npm* für viele Plattformen zur Verfügung zu stellen. Die mathematischen Grundlagen konnten dabei teilweise basierend auf der Referenzimplementation umgesetzt werden. Zudem wurde *gl-matrix* für das effiziente Arbeiten mit Vektoren eingesetzt [37].

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.

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 die Texturen für einen grossen Teil der Speichergrösse verantwortlich. 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 die prominenteste Farbe einer Textur extrahiert und diese für die Vereinfachungen verwendet. In Abbildung 4.9 ist dieser Unterschied visualisiert. Für die Wahl der Farbe wird das Modul *fast-average-color* verwendet [38]. Das Modul wählt den bestimmenden Farnton der Textur, welcher dann im *Material* des neuen Artefakts gespeichert wird. 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. Speichertchnisch ist es so nicht notwendig eine Textur zu verwenden, die Information kann auf einen einzigen Farbwert reduziert werden. Dies hat zudem den Vorteil, dass es nicht notwendig ist Informationen für das *Texture Mapping* abspeichern zu müssen.

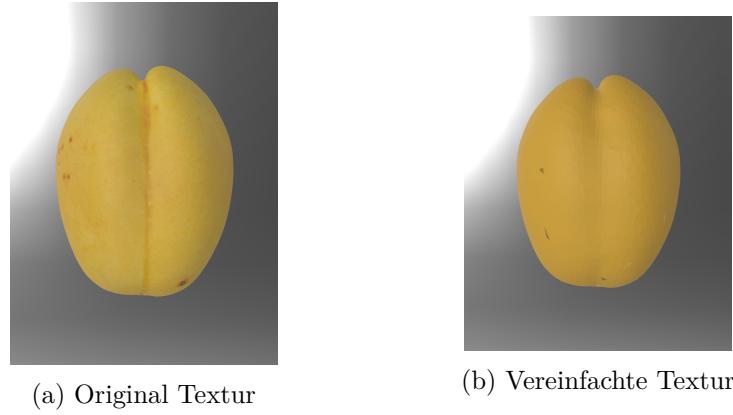


Abbildung 4.9.: Vergleich der Texturen

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 4.10 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 4.10.: Vergleich von Originalmodell mit vereinfachtem Modell

Bei einer Nahbetrachtung der Modelle sind die Unterschiede klar ersichtlich. Bei einer gewissen Distanz sind die Differenzen der beiden Modelle unauffälliger. In Abbildung 4.11 werden dieselben Modelle auf eine Distanz von 72 (m) dargestellt. Das m ist dabei in Klammern gesetzt, da es sich um keine reale Einheit, sondern um eine künstliche Einheit handelt. Die Standardeinstellung für die maximale Distanz des Kamera *Frustum* liegt bei Babylon.js zum Beispiel bei 10'000 (m) [39]. Auf diese Distanz kann der orangefarbene Schnabel beim Originalmodell noch knapp erkannt werden – andere Details sind auf diese Distanz jedoch nicht ersichtlich.



Abbildung 4.11.: Vergleich Originalmodell mit vereinfachtem Modell auf Distanz

4.2.6. Generierung von glTF

Für die *LOD*-Artefakte werden vom Originalmodell unabhängige *glTF* Dateien erstellt. Dafür wird das Tool *glTF-Transform* eingesetzt [40]. Das Erstellen eines solchen Basismodells ist in 4.12 ersichtlich. Hierbei wird die Skalierung und Rotation des Originalmodells verwendet (gespeichert in Variable *originalNode*).

```

1 const newDoc = new Document();
2 const scene = newDoc.createScene("scene");
3 const node = newDoc.createNode("node");
4 node.setRotation(originalNode.getRotation());
5 node.setScale(originalNode.getScale());
6
7 const mesh = newDoc.createMesh("mesh");

```

Abbildung 4.12.: Basis *glTF* Generierung

Anschliessend werden die neuen Informationen in *Buffer* gespeichert. Als Beispiel hierzu ist die Generierung des *Vertexbuffers* in Abbildung 4.13 ersichtlich. Ein *Mesh* kann mehrere *Primitives* beinhalten. Ein *Primitive* definiert einen Teil des Modells, dies beinhaltet unter anderem *Vertices* und *Material*-Informationen. Die Variable *primitiveIndex* speichert den Index des aktuellen *Primitives*. In der Variable *verticesAsFloat* werden die neuen *Vertices* als *Float32Array* gespeichert.

```

1 const newPrimitive = newDoc.createPrimitive();
2
3 const buffer = newDoc.createBuffer(`buffer_${primitiveIndex}_1`);
4
5 const positionAccessor = newDoc
6   .createAccessor(`data_${primitiveIndex}_1`)
7   .setArray(verticesAsFloat)
8   .setType(Accessor.Type.VEC3)
9   .setBuffer(buffer);
10
11 newPrimitive.setAttribute("POSITION", positionAccessor);

```

Abbildung 4.13.: Erstellen des *Buffers* für die *Vertices*

4.2.7. Iterativer Ansatz

Wie in der Referenzimplementation wird auf einen *Threshold* (Schwelle) gesetzt. Der *Threshold* ist ein Wert, welcher bestimmt, welche Fehlertoleranz gelten soll. Umso tiefer der *Threshold*, desto mehr *Edge Collapses* werden durchgeführt. Der *Threshold* wird kontinuierlich erhöht, so werden immer weitere *Triangles* entfernt. Der gewählte *Threshold* basiert direkt auf der Referenzimplementation [33]. Ein Ausschnitt aus dem Quellcode ist in Abbildung 4.14 ersichtlich. Gewisse Teile wurden für die Übersicht entfernt. Die Variable *triangles* beinhaltet alle *Triangles* eines *Primitives*. Jeder *Triangle* speichert die tiefste Fehlermetrik der drei zugehörigen *Vertices* in *error[3]*. Die Positionen 0, 1 und 2 entsprechen dabei den drei *Vertices*. Weist ein *Triangle* einen Fehlerwert, welcher tiefer ist als der *Threshold*, so wird ein *Edge Collapse* oder *Halfedge Collapse* in Betracht gezogen. In gewissen Fällen ist es möglich, dass eine ausreichende Vereinfachung nicht in absehbarer Zeit generiert werden kann. Dies ist abhängig von den *maxIterations*. Ein weiterer Faktor ist die Aggressivität, welche auf einen passenden Wert eingestellt wurde. Eine geeignete Berechnung des *Thresholds* hängt direkt mit der Normalisierung der Modellskalierung zusammen.

```

1  for (let iteration = 0; iteration < maxIterations; iteration++) {
2    if (initialTriangleCount - deletedTriangles <= targetTriangles) {
3      break; // check if targetTriangles are reached
4    }
5
6    const threshold = 0.000000001 * Math.pow(iteration + 3,
7        aggressiveness);
8    triangles.forEach((triangle) => {
9      if (triangle.error[3] > threshold) return;
10     // proceed with edge collapse operation
11   });
12 }
```

Abbildung 4.14.: Iteration für Vereinfachung

4.2.8. Edge Collapse

Bei der Durchführung einer *Edge Collapse* Operation muss ein neuer *Vertex* gefunden werden. Die bevorzugte Variante ist das Finden der optimalen Position. Für mehr Informationen zu den konkreten Implementationsdetails, siehe *Quadric-Based Polygonal Surface Simplification* Sektion 3.5 [23].

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.

4.3.1. Browser-Umgebung

Um den Umfang des Benchmarks überschaubar zu halten, wurde ausschliesslich ein Benchmark für *Google Chrome* entwickelt. *Google Chrome* basiert auf *Chromium*¹, die selbe 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 *Google Chrome* sind *Mozilla Firefox* oder *Safari* von *Apple* ebenfalls Optionen. Jedoch wurde primär aufgrund des Marktanteils von total rund 70 % [41] zugunsten von *Google Chrome* entschieden. Die getroffenen Aussagen bezüglich Laufzeitverhalten behalten ihre Gültigkeit auch für andere Browser.

¹Open Source Browser-Projekt, welches von *Google* entwickelt wird. Andere Browser basieren ebenfalls auf der Code-Basis von *Chromium*

Automation

Um die Tests durchzuführen, wird ein Testautomationstool benötigt; unter anderem wurde der Einsatz von *Selenium* in Erwägung gezogen. Der Vorteil von *Selenium* ist insbesondere, dass der Benchmark für weitere Browser ausgeweitet werden könnte. Da jedoch das 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. Um allfällige Diskrepanzen zwischen Systemen möglichst gering zu halten, wurde entschieden, auf die bewährte Lösung von *Google Chrome* zu setzen. *Selenium* bietet zurzeit eine suboptimale Integration für das *Chrome DevTools Protocol*. *Puppeteer*², 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.

4.3.2. 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.

²Node.js Bibliothek, die eine API anbietet zum Steuern von *Chromium* oder *Google 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 <i>GPU</i> -Zeit	Die totale Zeit, welche die <i>GPU</i> für Berechnungen benötigt.
Anzahl <i>GPU</i> -Events	Anzahl der Events an die <i>GPU</i> . 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 <i>GPU</i> -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 [42].

In Abbildung 4.15 ist ein Screenshot der Testapplikation ersichtlich. 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 [43]. 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. Es zeigte jedoch keinen nennenswerten Vorteil, weder für die optimierte, noch für die unoptimierte Version [44].

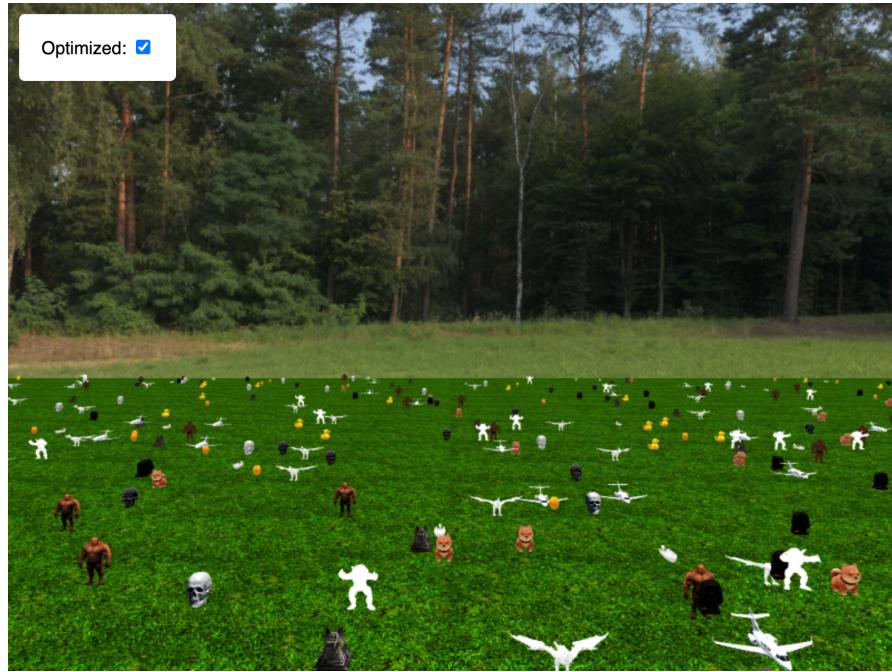


Abbildung 4.15.: Testapplikation mit einer Vielzahl an Modellen, welche von verschiedenen Quellen frei zur Verfügung gestellt wurden.

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 [45]. 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.

Analysen 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 unoptimisierte 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.

4.3.3. Auswertung

Der Benchmark wurde auf verschiedenen Geräten ausgeführt. Im Folgenden sind zwei Durchläufe mit jeweils 10 Proben erfasst.

Der erste Durchlauf wurde auf einem *MacBook Pro* mit einer dedizierten Grafikkarte durchgeführt. Wie in Abbildung 4.16a zu sehen, konnten die *FPS* auf 60 *FPS* gehoben werden im Vergleich zu den 48.2 *FPS* in der unoptimierten Version. Die Standardabweichung ist im Diagramm ebenfalls ersichtlich. 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, gemessen in Millisekunden, etwas erhöht, da mehr Modelle geladen werden müssen (siehe Abbildung 4.16b). Diese wuchs aber nur um rund 10 %. Zu beachten ist, dass die Modelle lediglich lokal geladen werden – über ein richtiges Netzwerk sind die Ladezeiten entsprechend länger.

Die detaillierten Informationen befinden sich in Anhang A.1.

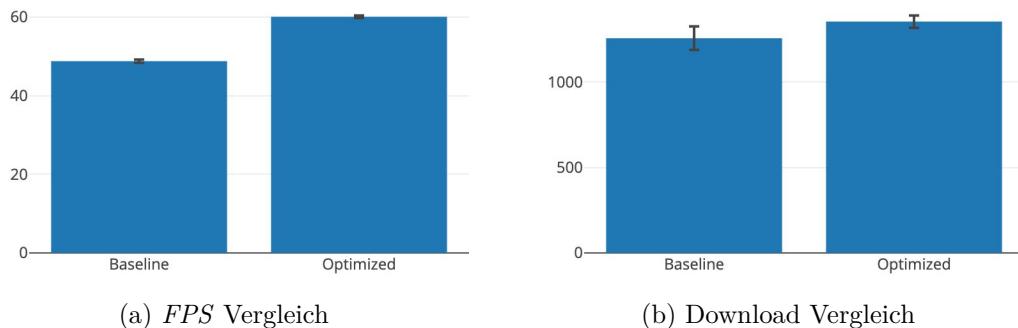


Abbildung 4.16.: Vergleich Kennzahlen auf Gerät mit dedizierter Grafikkarte

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 des *glb* Formats 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.

Rendering Engine Programm, das zuständig für die Darstellung von 3D-Grafiken ist. 8, 33, 46, 55

Scene Graph Datenstruktur, welche die räumliche Anordnung der Modelle einer 3D-Szenerie beschreibt. 23, 24, 29, 30

SDK *Software Development Kit.* 14

Semantic Versioning Konvention für Veröffentlichung von Software Paketen, siehe: semver.org [8]. 9

5.1. Vergleich zwischen <i>Low-Poly</i> und <i>LOD</i> -simplifiziertem Model	52
A.1. Durchlauf Benchmark auf <i>MacBook Pro 2018</i>	74
A.2. Hardwarespezifikation <i>MacBook Pro 2018</i>	74
A.3. Durchlauf Benchmark auf <i>Dell Precision 5540</i>	75
A.4. Hardwarespezifikation <i>Dell Precision 5540</i>	75

6.4. Tabellenverzeichnis

3.1. Übersicht Merkmale der verschiedenen <i>LOD</i> -Systeme	30
4.1. Kennzahlen für Benchmark	47

A. Anhang

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

A.2. Packages

lode Benchmark

The benchmark expects `../demo` project to be running.

The benchmark will automatically run a number of samples, to customize the amount of samples, pass `--iterations=n`. As the benchmark will finish after a given timeout to prevent endless runs, you can pass `--timeout=n` in ms which defaults to 20'000. The benchmark can be run in a headless environment. It's not advised to do so as the result is not reliable. If this is desired (e.g. for testing purposes on CI) you can pass `--headless`.

By default the benchmark will only show the report. In order to show more detailed information which is helpful for debugging pass `--logDetails`. This will log more information from the demo scene.

Dev

Run `yarn setup` to get up and running and then `yarn start` **Note:** The `../demo` project needs to be running as well.

Dev

Run `yarn setup` to get up and running

Publish

- Prepare versions and ensure clean setup state
- `yarn build`
- `npm publish --access public`

Currently publish workflow is only tested on Mac.

lode Demo

This is a demo showing the benefits of using LODs.

Dev

Run `yarn setup` to get up and running and then `yarn start`

LODE - CLI

Run `yarn lode-cli` to run the lode-cli to generate the simplified meshes used in the optimized version.

Info

There are currently two variants:

- optimized
- non optimized

In order to run the optimized version pass `?optimize`.

This demo will have to be expanded further, it's currently only intended to be used in order to define the base benchmark.

lode UI

lode UI provides a web interface for managing lode artifacts.

Key Features are:

- Live configuration of visibility trigger threshold
- Update target triangle configuration and see generated artifacts

Dev

Make sure a `lode-cli` instance is started in watch mode. `yarn start` will start the application on port 3000.

lode Benchmark Plotter

This module shows the data of the benchmark in a clean way. Make sure to have run the benchmark beforehand as the plotter will always visualize the last result. The plotter uses `plotly` to generate the diagrams.

Usage

Run `yarn start` to generate `fpsChart.jpg` and `downloadChart.jpg` which can be used in the report.

Dev

For local development run `yarn visualize` to start the visualizer on port 8081.

A.4.2. Meeting-Protokolle

21.05.2021

Besprochenes:

- Demo Szenerie finalisiert
- UI-Design für Lode-UI
- Bericht
- Basis Texturierung (Bsp. Ente)

Anmerkungen:

- -

Geklärte Fragestellungen:

- Titelanpassung möglich? -> Ja, "Pipeline" statt "Generierungs-System": Level of Detail Pipeline für 3D-Webapplikationen

Offene Fragestellungen:

- -

Nächste Schritte:

- Demoszenerie Fine-Tuning
- Fine-Tuning an Algorithmus
- Bericht

A. Anhang

30.04.2021

Besprochenes:

- Mehrere Levels (vom Benutzer einstellbar)
- *glTF* Artefakten mit mehreren Meshs unterstützt

Anmerkungen:

- -

Geklärte Fragestellungen:

- Generelles zur Präsentation?
 - Online
 - Vier Personen: (Simon, Marc, Herr Burkert und Experte)
 - Teilweise noch Studiengangleitung
 - Zielpublikum: IT-Leute

Offene Fragestellungen:

- -

Nächste Schritte:

- Level-Wahl pro Modell
- Fine-Tune Webversion
- Demo Projekt dokumentieren

A. Anhang

23.04.2021

Besprochenes:

- Demo
- Pipelines
- Bericht

Anmerkungen:

- Abstände um Fussnoten anschauen
- Ausgangslage noch breiter einsteigen (Inspiration von 2.1)
- Zwei Nomen nacheinander mit Bindestrich oder zusammenschreiben
- 2.4 könnte ausführlicher sein
- Graphics pipeline ausführlicher
- Generell: Möglichst viele Bilder, Beispiele und möglichst konkret

Geklärte Fragestellungen:

- Gibt es Seiten Maximum? -> Nein, muss einfach alles Wichtige drinstehen

Offene Fragestellungen:

- -

Nächste Schritte:

- Zusammenfassung & Vorwort schreiben
- Anzahl Level kann vom Benutzer angegeben werden
- CLI soll nur Modell generieren, die nicht bereits generiert sind
- LOD-Runtime als alleinstehendes Projekt zur Verfügung stellen

A. Anhang

16.04.2021

Besprochenes:

- Glossar mit Fussnoten
- Pipeline für Berichtsgenerierung
- Algorithmus Prototyp läuft

Anmerkungen:

- Glossar fehlt in Pipelinebuild

Geklärte Fragestellungen:

- -

Offene Fragestellungen:

- -

Nächste Schritte:

- CLI Integration finalisieren
- *OBJ* Modelle suchen, die mit Algorithmus gut funktionieren
- CLI in Demoprojekt verwenden
- Demo Szenerie aufbauen mit Flyover
- DevExperience (Lode-Runtime)
- Algorithmus weitertreiben & dokumentieren

A. Anhang

09.04.2021

Besprochenes:

- Algorithmus ausgesucht
- Datenformat auf *glTF* festgelegt
- Switch zu Three.js
- Setup deployment Pipeline

Anmerkungen:

- -

Geklärte Fragestellungen:

- Ähnliche bereits durchgeführte Berichte? -> Gibt es so nicht. Punktuelles Feedback statt Orientierungshilfe. Arbeit über functional GO [en] war gut – per Mail verschickt.

Offene Fragestellungen:

- -

Nächste Schritte:

- Bericht weitertreiben
- Pipeline fertig aufsetzen
- Algorithmus Prototyp zum Laufen bringen

A. Anhang

26.03.2021

Besprochenes:

- Bericht
 - Quellen
- Benchmark
- CLI

Anmerkungen:

- Benchmark
 - Rahmenbedingungen festhalten

Geklärte Fragestellungen:

- Glossareinträge jedes Mal referenzieren oder nur beim ersten Verwenden? -> Beim ersten Mal
- Glossareinträge beim ersten Verwenden direkt ebenfalls in die Fussnoten aufnehmen? -> Ja, guter Ansatz

Offene Fragestellungen:

- -

Nächste Schritte:

- Glossar / Emphasizing / Fussnoten
- Benchmark-Sektion in Report auf Akademisches-Level hieven
- Vergleich LOD System
- Anfang Auswahl LOD-Algorithmus
- Technischer Durchstich
- Switch von Babylon.js zu Three.js
- Github Pipeline und demo-host aufsetzen

A. Anhang

19.03.2021

Besprochenes:

- Bericht
- Zeitplan gezeigt und Abgrenzungen bestätigt

Anmerkungen:

- Feedback Bericht Einleitung:
 - Ausgangslage einstieg ein bisschen harsch
 - Gross- / Kleinschreibung
- Feedback Bericht Grundlagen
 - WebGL, WebGPU, OpenGL erklären
 - Eventuell mit Fussnote oder Quellen verweise auf weiterführende Informationen
 - Perpendikular -> Fussnote
 - Mehr Beispiele (Visuell / Code)
 - Transformations visuals ausgangs Bild jeweils links daneben anzeigen
 - Frustum? -> Erklären
- Feedback Bericht Vorgehen
 - Model erklären im Bericht

Geklärte Fragestellungen:

- Sind Anglizismen OK? -> Ja, aber eventuell kursiv drucken
- Glossar aufbauen
- Sollen wir vermehrt auf Quellen setzen? -> Ja

Offene Fragestellungen:

- -

Nächste Schritte:

- Benchmark erweitern
- Bericht weitertreiben
- CLI aufsetzen

A. Anhang

12.03.2021

Besprochenes:

- Demo Projekt mit Benchmark
- Benchmarks nur mit Chrome getestet
- Verwendung von Puppeteer für Benchmark-tests
- Einleitung und Vorgehen Kapitel des Berichts

Anmerkungen:

- Sieht so weit gut aus. Auf gutem Weg.
- Maximal 2/3 Seite Code-Ausschnitte

Geklärte Fragestellungen:

- Hat Herr Burkert technische Grundlagen, um Projekt lokal laufen zu lassen? -> Ja.
- Sind Google interne Dokumente als Quellen zulässig? -> Ja, da öffentlich zugänglich. Weitere wissenschaftliche Quellen wünschenswert.

Offene Fragestellungen:

- -

Nächste Schritte:

- Herr Burkert gibt Feedback zu Detailierungsgrad des Berichts bis nächste Woche
- Herr Burkert schaut sich das Benchmark Projekt an und gibt erstes Feedback
- Zeitplan aktualisieren
- Simon und Marc geben Herrn Burkert ein Signal, wenn Bericht Inhaltlich bereit ist
- Alle Dokumente im Teams in den General Kanal verschieben
- In Zukunft Termine für Besprechungen aufsetzen.

A. Anhang

05.03.2021

Besprochenes:

- Gantt Chart (<https://www.figma.com/file/8TcufkDndGDy10a1GoVVZP/Dokumentation-BA?node-id=1%3A745>)
- Report mit LaTeX: <https://github.com/kreativwebdesign/lode/blob/main/documentation/report/doc.pdf>
- Experiment zeigt Nutzen von LOD für mobile Geräte

Anmerkungen:

- Ausführliche Einführung ins Thema LOD erwünscht (15 Seiten)
- Meeting Protokolle im Teams erfassen
- Aufgabenstellung im Teams hochladen
- Demo im Bericht ausführlich beschreiben mit Bildern und Statistiken/Messungen

Geklärte Fragestellungen:

- Muss Aufgabenstellung verfeinert werden? -> Nein, Highlevel gewährt uns Flexibilität

Offene Fragestellungen:

- Gibt es einen einfachen weg, die GPU zu drosseln? -> Hardware abhängig, jedoch kann mit Chrome DevTools und Puppeteer im Headless Modus genügend detaillierte Informationen bzgl. GPU Leistung gefunden werden.

Nächste Schritte:

- Einführung zum Thema LOD im Bericht schreiben
- Meeting nächsten Freitag 12.03. 13:00 Uhr um die erste Fassung der Einführung zu besprechen

A. Anhang

26.02.2021

Besprochenes:

- -

Anmerkungen:

- -

Geklärte Fragestellungen:

- Sprache? -> nur Deutsch
- Code öffentlich? -> auf Github (<https://github.com/kreativwebdesign/lode>)
- Relevanz Projektmanagement? -> Anhang
- Berichtstruktur? -> Details im Intranet

Offene Fragestellungen:

- -

Nächste Schritte:

- -