Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

# Bachelorarbeit in Informatik

## Level of Detail Generierungs-System für 3D-Webapplikationen (max. 4 Zeilen)

| | |
|---|---|
| **Autoren** | Marc Berli<br>Simon Stucki |
| **Hauptbetreuung** | Dr. Gerrit Burkert |
| **Nebenbetreuung** | Titel Vorname Name |
| **Industriepartner** | |
| **Externe Betreuung** | |
| **Datum** | 17.04.2021 |

Bitte füllen Sie das Titelblatt aus und berücksichtigen Sie Folgendes:

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

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

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

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

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

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

Ort, Datum:

……

Unterschriften:

……...

……

……

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

## Zusammenfassung

Die rapide Entwicklung der Webtechnologien in den vergangenen Jahren ermöglicht es, immer komplexere Applikationen für ein breites Spektrum zugänglich zu machen. Unter anderem können 3D-Visualisierungen im Webbrowser umgesetzt werden. 3D-Applikationen haben durch die zunehmende Komplexität und Grösse erhöhte Anforderungen, welche durch das junge Alter des Entwicklungsprozess' noch mühsam sind, zu erfüllen. Durch die Vielzahl an Geräten, auf welchen ein Webbrowser auf eine solche Applikation zugreifen kann, ist das Optimieren der Performanz unabdingbar. Zum Einen ist die Downloadgrösse essenziell, um die Applikation in möglichst vielen Situationen zugänglich zu machen. Zum Anderen soll insbesondere die Rechenleistung der Geräte effizient genutzt werden, um die Applikation auf möglichst vielen Gerätearten verwenden zu können. Ersteres kann verbessert werden, indem ein hochauflösendes 3D-Modell vorgängig vereinfacht wird. So kann die Dateigrösse verringert und insbesondere auf mobilen Geräten Daten gespart werden. Für das zweite Problem gibt es eine Vielzahl von Lösungsansätzen. In dieser Arbeit wird eine Option aufgezeigt indem mehrere Detailstufen (*Level of Details – LOD*) eines Modells generiert und je nach Entfernung der virtuellen Kamera angezeigt werden. So muss ein Objekt, welches kaum noch sichtbar, ganz hinten im Bild angezeigt wird, nicht mehr hochauflösend sein. Zurzeit ist das Generieren und Einsetzen solcher Artefakte im Web nur bedingt automatisiert und beträchtlicher manueller Aufwand ist erforderlich. Ziel dieser Arbeit ist, ein solches Tool zu entwickeln und nahtlos in den Arbeitsprozess der Entwickler einzugliedern.

Hierfür wurden die verschiedenen Ansätze zur Generierung von solchen LOD-Artefakten verglichen und die zugehörigen Algorithmen analysiert. Darauf basierend wurde der gewählte Algorithmus implementiert und ein Tool entworfen, diesen in den Arbeitsprozess zu integrieren. Mittels eigenentwickelten Benchmarks wurde bewiesen, dass der Einsatz von LOD, die Laufzeitperformanz von 3D-Webapplikationen verbessert.

Das Resultat ist ein Tool, welches es erlaubt mit geringem manuellen Aufwand die Laufzeitperformanz von 3D-Webapplikation zu verbessern.

**Abstract**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

# Vorwort

Die Autoren der Arbeit sind seit mehreren Jahren in der Webentwicklung tätig. Der rapide Fortschritt in der Webentwicklung ist insbesondere dem Engagement der vielen Freiwilligen zu verdanken. Insbesondere für 3D-Visualisierungen sind in den vergangenen Jahren viele neue Möglichkeiten entstanden. Die Autoren teilen die Begeisterung für 3D-Visualisierungen. Aufgrund des noch jungen Alters sind die Tools noch nicht auf dem Level der anderen Plattformen. Diese Arbeit stellt insofern einen Versuch dar, etwas an die Community zurückzugeben und die Arbeit im Web im Bezug auf 3D-Applikationen zu erweitern und zu erleichtern.
Die Performanz in Webapplikationen hinkt derer nativer Applikationen hinterher. Dies bedeutet, dass jede mögliche Optimierung in dieser Hinsicht sehr wertvoll ist. Die sogenannten *Level of Details* sind eine Methode um die Performanz zu verbessern. Es gibt bereits Ansätze, diese sind jedoch noch nicht ausreichend ausgereift um in einer produktiven Applikation eingesetzt werden können.
Insbesondere die Anzahl der manuellen Arbeitsschritte, welche bei der Entwicklung von 3D-Applikationen notwendig sind, sind den Autoren der Arbeit ein Dorn im Auge. Ziel der Arbeit ist es somit Prozesse, welche sich in anderen Teilen der Webentwicklung bewährt haben auch für die Entwicklung von 3D-Applikationen einsetzen zu können und den manuellen Aufwand möglichst gering zu halten.
Ein besonderes Dankeschön geht an unseren Hauptbetreuer Gerrit Burkert, welcher uns während des Verfassens der Arbeit unterstützt hat.

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Kontext

In der Arbeit wird der Fokus auf die Web-Plattform gelegt. Die meisten Grundlagen sind jedoch auch unabhängig davon zutreffend. Für die Vereinfachung wird im Folgenden der Begriff "Das Web"für die Plattform von verschiedenen Web Technologien genutzt. Dies beinhaltet insbesondere vom W3C veröffentlichte Spezifikationen.

## 1.2. Ausgangslage

Dank der Rechenleistung auf modernen Geräten ist es möglich anspruchsvolle 3D-Visualisierungen in Echtzeit auf diversen Geräten anzuwenden. Da diese Applikationen sehr rechenintensiv sind und gerade mobile Geräte in ihrer Rechenleistung beschränkt sind, ist Performance Optimierung im 3D-Rendering unabdinglich. Insbesondere die Komplexität der Modelle hat einen signifikanten Einfluss auf die Leistung. Eine Möglichkeit zur Optimierung ist das Anzeigen von vereinfachten Modellen ab bestimmten Distanzen zum Betrachter. So kann z.B. ein Modell in grosser Distanz vereinfacht dargestellt werden, solange bei genauer Betrachtung mehr Details sichtbar werden. In diversen Rendering Engines [1] gibt es deshalb Möglichkeiten für das Verwenden von sogenannten Level Of Details (LOD) Artefakten. Für Game Engines [2] wie Unreal oder Unity gibt es bewährte Möglichkeiten, um den Einsatz von LOD Artefakten zu vereinfachen. Zurzeit gibt es im Web Bereich keine weit verbreitete Möglichkeit für das Generieren von LOD Artefakten.

### 1.2.1. 3D-Rendering im Web

3D-Visualisierungen werden in vielen Branchen verwendet. Virtual Reality, CAD Anwendungen oder Computerspiele sind bekannte Anwendungsgebiete, welche 3D-Visualisierungen einsetzen. Dank leistungsstärkeren Geräten sind in vielen weiteren Bereichen realere Visualisierungen möglich.
Anwendungen können auf spezifische Hardware wie zum Beispiel die Spielkonsolen PlayStation oder Xbox ausgerichtet sein. Dies hat den Nachteil, dass die Verteilung der Software aufwändiger ist, da spezifische Hardware notwendig ist. Für gewisse Anwendungsgebiete ist dies kein Problem.
Für hardwareunabhängige Anwendungen eignet sich das Web hervorragend. Viele Benutzer haben Zugang zu einem Desktop, Tablet oder Mobiltelefon. Somit ermöglicht das

[1] Teilprogramm, das zuständig für die Darstellung von Grafiken ist
[2] Framework, das für den Spielverlauf und dessen Darstellung verantwortlich ist

Webanwendungen mit weniger Aufwand einem grossen Zielpublikum zugänglich zu machen. Seit einigen Jahren ist es auch möglich, 3D-Visualisierungen im Web zu realisieren. Als Basis dafür 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. [1] Alternativ zu WebGL wird zurzeit ein weiterer Standard entwickelt: WebGPU. Dieser ist zur Zeit des Schreibens noch in Entwicklung und wird deshalb nicht weiter berücksichtigt, auch wenn ein grosses Potenzial vorhanden ist. [2]
Die Unabhängigkeit der Hardware bedeutet jedoch auch, dass Optimierung der Performanz in Webanwendungen unabdinglich ist, um allen Benutzern ein optimales Erlebnis zu ermöglichen. Im Vergleich zu fixen Hardware Anwendungen ist es realistisch, dass eine Webanwendung sowohl auf einem leistungsfähigen Desktop Computer als auch auf einem günstigen Mobilgerät verwendet wird.
Zudem ist WebGL eine junge Technologie und wurde erst 2011 veröffentlicht – verglichen mit dem initialen Release Date von OpenGL [3] welches im Jahre 1992 publiziert wurde. [1, 3] Nicht nur das Alter, sondern auch die Natur der Web-Plattform haben dazu beigetragen, dass WebGL ein langsames Wachstum verspürt hat. Um einen Webstandard wie WebGL einsetzen zu können müssen alle grossen Browser die Spezifikation implementieren. Ansonsten kann der grosse Vorteil des Webs – einfache Verteilung an alle Benutzer – nicht in vollem Umfang genutzt werden. So hat zum Beispiel Internet Explorer 10 keinen Support und es wurde erstmals Ende 2013 möglich im Internet Explorer 11 3D-Anwendungen für ein breites Publikum zu entwickeln.

### 1.2.2. JavaScript Bibliotheken

Für eine einfachere Arbeit mit WebGL gibt es verschiedene JavaScript Bibliotheken, welche eine Abstrahierungsschicht einfügen. Die bekanntesten werden hier kurz erläutert.

#### *Three.js*

Ist die wohl weitverbreiteste Bibliothek für 3D-Rendering im Web. Die Community hinter Three.js ist aktiv und das offene Produkt wird somit konstant weiterentwickelt. Insbesondere die Erweiterungen für *React*, eine populäre Bibliothek für die Entwicklung von Frontend Applikationen, zeigen den Innovationsdrang.

#### *Babylon.js*

Ein weiterer Kandidat ist Babylon.js, eine offene Bibliothek, welche die Entwicklung von 3D-Applikationen vereinfacht. Babylon.js zeichnet sich vor allem durch ein breites Featureset aus.

#### *PlayCanvas*

PlayCanvas ist eine offene 3D-Engine, welche auch einen proprietären Cloud Service anbietet.

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

***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.

### 1.2.3. Stand der Technik

Wie erwähnt gibt es bereits umfangreiche LOD Systeme für komplexe Anwendungsgebiete in anderen Umgebungen. Diese werden in der Sektion Unterabschnitt 3.1.1 detaillierter erläutert. Die erläuterten Bibliotheken bieten Funktionen für das Laden von LOD Artefakten. Teilweise gibt es die Möglichkeit Vereinfachungen im Browser zu generieren. Das Generieren von LOD Artefakten zur Laufzeit ist jedoch für Webapplikationen nicht geeignet, da dies signifikante Auswirkungen auf das Laufzeitverhalten hat. So dauert das Optimieren eines komplexen Modells wie zum Beispiel bei Babylon.js demonstriert auch auf rechnungsstarken Geräten mehrere Sekunden [4].
Des Weiteren sind die Systeme nicht kompatibel mit anderen Bibliotheken. So kann die Auto LOD Lösung von Babylon.js nicht in Three.js verwendet werden, obwohl die Problemstellung dies grundsätzlich erlauben würde.

## 1.3. Zielsetzung

Ziel der Arbeit ist es, ein Tool zu entwickeln, das den Umgang mit LOD Artefakten im Web vereinfacht. Hierfür muss vorab der Beleg erbracht werden, dass das Laufzeitverhalten der Applikation mit LOD Artfakten verbessert werden kann. Des Weiteren muss ein Algorithmus entwickelt werden, welcher es erlaubt Modelle drastisch zu vereinfachen ohne die grobe geometrische Form zu verlieren. Im Anschluss muss das Tool zur Verfügung gestellt werden, sodass es in der Praxis eingesetzt werden kann. Hierfür muss insbesondere berücksichtigt werden, dass die Artefakte nicht innerhalb des Browsers generiert werden sollen. Zudem soll der Einsatz des Tools einen möglichst geringen Zusatzaufwand bedeuten und für ein breites Spektrum an Modellen angewendet werden können. Des Weiteren soll das Tool soweit erweiterbar sein, dass es für verschiedene Bibliotheken eingesetzt werden kann, ohne den Kern neu entwickeln zu müssen.

# 2. Theoretische Grundlagen

Im folgenden Abschnitt werden die verschiedenen Grundlagen für die Entwicklung eines LOD Systems kurz erläutert. Die Themen sind soweit als möglich voneinander getrennt.

## 2.1. 3D-Modelle

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

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

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

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

Um komplexere Formen wie Vierecke (*Quads*) zu bilden, werden jeweils mehrere *Triangle* kombiniert. Abbildung 2.1 zeigt die Vereinfachung eines *Quads* in zwei *Triangle*.

(a) Polygon (b) Polygon als Sammlung von Triangles

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

Für eine Sammlung von Punkten wird generell der Term Polygon verwendet. Ein Modell besteht aus einer beliebigen Anzahl Polygone.
Die Verbindung zwischen zwei Vertices ist eine sogenannte Edge (Kante). Verbindet man die Punkte eines Polygons und füllt die Fläche, ergibt sich schlussendlich ein Face (Fläche).

#### *Weitere Attribute*

Neben den geometrischen Attributen verfügt ein Modell ebenfalls über visuelle Attribute. So wird für jeden Vertex ein sogenannter *Normal* definiert. Ein *Normal* ist ein Vektor der im einfachsten Fall senkrecht zu den zwei an diesem Vertex verbundenen Edges verläuft. In der Geometrie ist dies auch bekannt als Normale. *Normals* werden häufig für das Berechnen von Reflexionen verwendet. *Normals* werden auch für gewisse Performanz Optimierungen eingesetzt, dazu mehr in Absatz 2.4.
Um die Oberfläche von Modellen zu definieren wird häufig ein sogenanntes Texture Mapping [1] durchgeführt. Hierfür sind zusätzliche Informationen für ein Modell notwendig. In der Praxis finden viele weitere Methoden Anwendung, auf diese wird jedoch hier nicht weiter eingegangen.

#### *Abgrenzung*

Diese Sektion erläuterte die für diese Arbeit relevanten polygonale Modelle. Alternativen wie zum Beispiel *Point Clouds*, welche das Resultat von 3D-Scans sind, werden nicht weiter erläutert. Auch andere Methoden wie *NURBS*, welche Modelle mithilfe von mathematisch definierten Flächen modelliert, werden aussen vor gelassen.

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

### *Formate*

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

**OBJ** Wavefront OBJ ist ein offenes Dateiformat das von Wavefront Technologies 1989 entwickelt wurde. Das Format ist jedoch speichertechnisch ineffizient und verfügt zudem nur über ein limitiertes Feature Set. Des Weiteren gibt es keine zentrale Instanz, welche eine Spezifikation liefert und Informationen sind deshalb schwerer zu finden. [5]

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

**glTF** Seit 2015 gibt es ein offenes, modernes sowie auf optimale Speichernutzung fokussiertes Format: glTF. Dieses Format wird von der Khronos Gruppe entwickelt. [6] In dieser Arbeit wird es als Austauschformat verwendet, deshalb wird im folgenden genauer auf das Format eingegangen.
Die meisten 3D-Grafikprogramme wie Blender (.blend) verwenden ihre eigenen Dateiformate. Diese Dateiformate sind jedoch für den Einsatz in einer Applikation ungeeignet da sie insbesondere nicht für die Laufzeitperformanz optimiert sind. Deswegen brauchte es für jedes Eingangsformat für jedes Ausgangsformat einen *converter*, wie in Abbildung 2.2 ersichtlich.

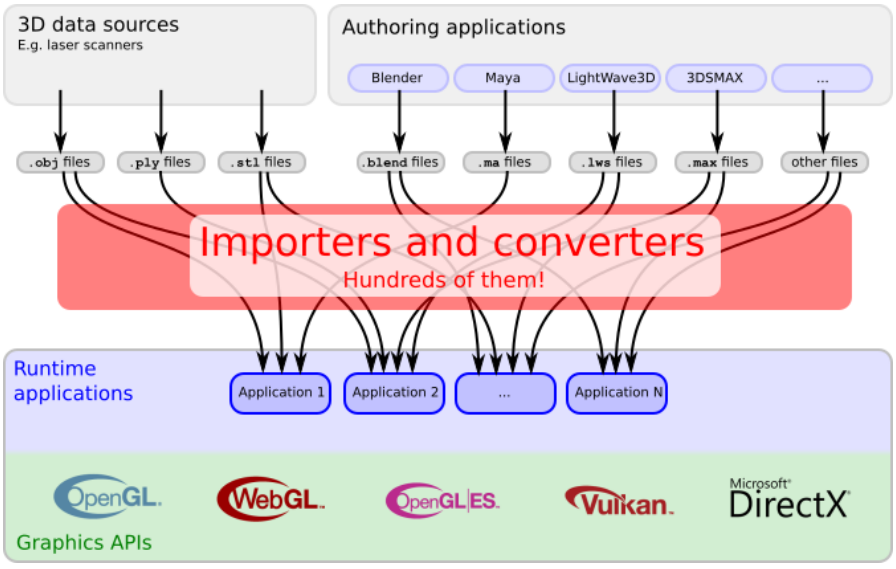


Abbildung 2.2.: Convertierungspipeline vor glTF [6]

Durch die immer breiter werdende Nachfrage nach 3D-Applikationen wurde ein Format benötigt, dass zum einen Applikationsunabhängig verwendet und zum anderen performant im Web eingesetzt werden kann. [6] Durch diese Abstraktion kann eine Vielzahl *converter* vermieden werden und nur wenige, sehr spezifische Dateiformate benötigen noch einen einzigen *converter*. Diese neue Pipeline ist in Abbildung 2.3 dargestellt. [6]

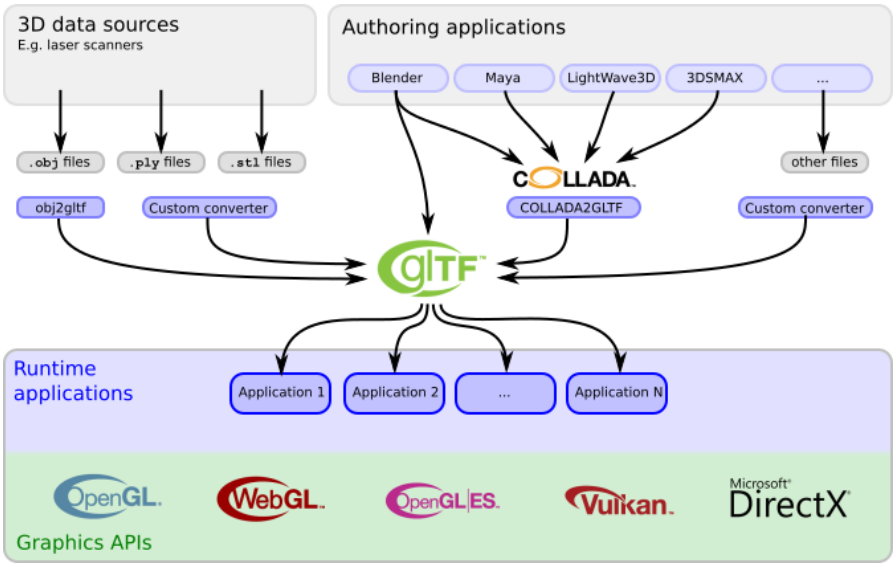


Abbildung 2.3.: Convertierungspipeline mit glTF [6]

Die Basisstruktur von glTF basiert auf JSON. Darin ist die komplette Szenerie des Modells beschrieben und in Abbildung 2.4 aufgezeigt.[6]

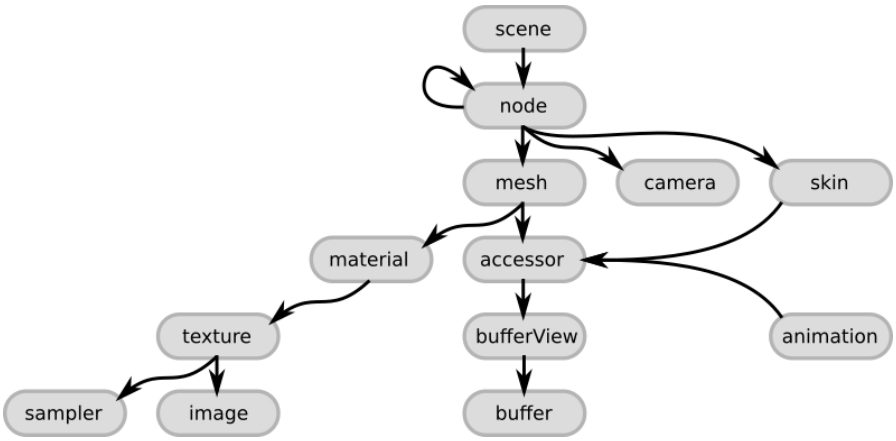


Abbildung 2.4.: glTF Datenstruktur [6]

Die wichtigsten Elemente dieser Struktur kurz erläutert:

- **Scene**: Einstiegspunkt der Szenerie und verweist auf alle Top-Level *nodes*.

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

## 2.2. Transformation von Modellen

Um ein Modell vereinfachen zu können, muss es verändert werden. Diese Veränderungen können in Basis Operationen vereinfacht erläutert werden.
Im Folgenden werden Transformationen mithilfe einer 2D Visualisierung erläutert. Das Modell kann jedoch ebenfalls im dreidimensionalen Raum sein – die Funktionsweise bleibt identisch.
Für die folgenden Beispiele wird jeweils das Modell aus Abbildung 2.5 verwendet.

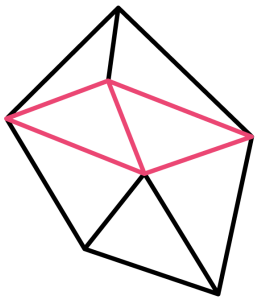

Abbildung 2.5.: Modell Basis

### *Edge Collapse*

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

(a) Modell Ausgangslage (b) Edge Collapse

Abbildung 2.6.: Transformation mittels Edge Collapse

### *Halfedge Collapse*

Hierbei wird ein Vertex direkt entfernt und alle Edges auf einen danebenliegenden Vertex zusammengelegt. Bei dieser Operation muss kein neuer *Vertex* definiert werden, sondern ein bereits bestehender *Vertex* kann wiederverwendet werden. Wie in Abbildung 2.7 ersichtlich, können auch in diesem Fall zwei *Triangles* entfernt werden.

(a) Modell Ausgangslage (b) Halfedge Collapse

Abbildung 2.7.: Transformation mittels Halfegde Collapse

***Vertex Removal***

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

(a) Modell Ausgangslage (b) Vertex Removal

Abbildung 2.8.: Transformation mittels Vertex Removal

## 2.3. Grafikpipeline

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

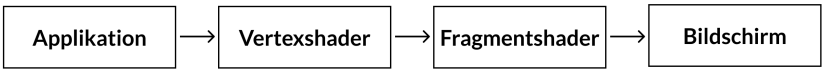


Abbildung 2.9.: Schritte einer Realtime Rendering Pipeline

***Applikation***

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

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

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

Abbildung 2.10.: Definition eines Triangles

***Vertexshader***

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

```
attribute vec3 coordinates;

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

Abbildung 2.11.: Vertexshader für die Definition eines Punktes

***Fragmentshader***

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

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

Abbildung 2.12.: Fragmentshader für das definieren von Farbwerten

***Bildschirm***

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

## 2.4. Performanzoptimierung

Visualisierungen können zu komplex werden, um jederzeit performant und interaktiv gerendert zu werden. Insbesondere wenn viele Objekte gleichzeitig sichtbar sind, lohnt es sich Performanzoptimierungen durchzuführen. Im Idealfall geschieht dies jedoch, ohne dass der Anwender dies bemerkt.
In diesem Abschnitt werden mögliche Ansätze erklärt, welche helfen sollen, die Render-Performanz zu erhöhen. Diese Arbeit konzentriert sich jedoch auf den Ansatz von Level of Detail; die anderen Ansätze werden nur kurz erläutert. Detailliert auf Level of Detail wird in Abschnitt 2.5 eingegangen.

***Frustum culling***

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

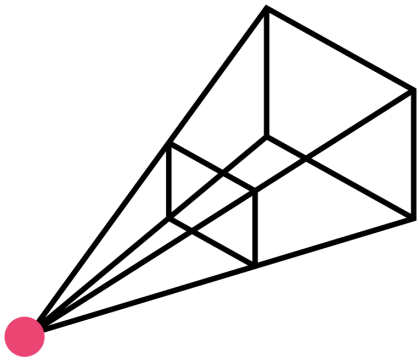

Abbildung 2.13.: Kamera Frustum

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

#### *Occlusion culling*

Polygone bzw. Objekte, welche komplett von anderen Objekten überdeckt werden, werden bei dieser Variante nicht prozessiert.

#### *Backface culling*

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

#### *Parallel rendering*

Auch bekannt unter Distributed Rendering ist der Einsatz von Techniken aus dem Parallel Programming in Visualisierungsanwendungen.

#### *Image-based rendering*

In gewissen Fällen kann das Modellieren übersprungen werden. So kann anhand von Bildmaterial eine 3D-Illusion erzeugt werden.

## 2.5. Einführung Level of Detail

Als Level Of Detail (LOD) werden die verschiedenen Detailstufen bei der virtuellen Darstellung bezeichnet. Dies wird verwendet, um die Geschwindigkeit von Anwendungen zu steigern, indem Objekte im Nahbereich detailiert angezeigt werden; wohingegen Elemente im Fernbereich deutlich vereinfacht dargestellt werden.
Ein LOD Algorithmus hat das Ziel für ein gegebenes Modell eine vereinfachte Darstellung zu finden, welche das Original ausreichend annähert. Um diese Approximationen zu generieren, kann eine Vielzahl von Algorithmen verwendet werden. Es gibt verschiedene Ansätze zur Generierung von LODs, welche in Unterabschnitt 2.5.2 im Detail erläutert werden.

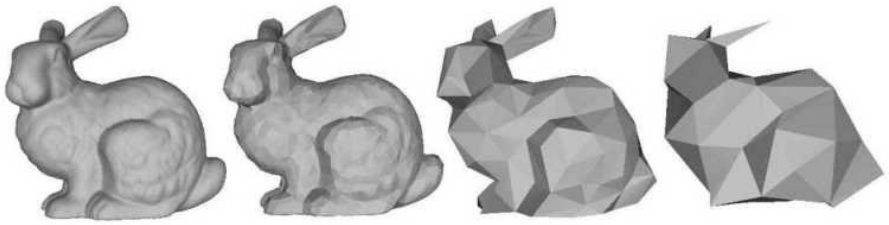

Abbildung 2.14.: Level Of Detail Visualisierung vier Hasen

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 Polygone, wird es bereits im ersten Schritt auf 2'502 Polygone reduziert. Dies ist eine enorme Reduktion von ca. 96.5%. Im dritten Schritt wird die Anzahl Polygone wiederum um ca. 90% auf 251 reduziert. Schlussendlich hat das letzte Objekt noch 76 Polygone was knapp 0.1% der ursprünglichen Anzahl entspricht.

### 2.5.1. Ansätze für LOD Artefakte

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

#### *Diskrete LOD (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 [2] notwendig da ausschliesslich ganze Modelle ausgetauscht werden. Ein Nachteil sind jedoch merkbare harte Grenzen. Der Benutzer stellt beim umhergehen in der Szene fest, wenn das Modell mit einer einfacheren Version ausgetauscht wird. Es ist zudem häufig nicht möglich

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

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

#### *Kontinuierliche LOD (CLOD)*

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

#### *Hierarchische LOD (HLOD)*

Bei HLOD werden mehrere Objekte in einen Cluster gruppiert. Diese Methode erlaubt es somit zum Beispiel Terrains optimal abzubilden. So werden Ausschnitte, welche nahe beim Benutzer sind mit hoher Auflösung dargestellt während weiter entferntere 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

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

#### *Vertex Dezimierung*

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

[3] Vertices im dreidimensionalen Raum ohne zugehörige *Triangle*

#### *Edge Collapse*

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

#### *Face Collapse*

Bei dieser Variante werden jeweils Flächen entfernt und die umliegenden Flächen zusammengelegt.

# 3. Vorgehen

## 3.1. Vergleich LOD Systeme

Die verschiedenen LOD Systeme sowie ihre Vor- und Nachteile werden in Unterabschnitt 2.5.1 erläutert. In diesem Schritt wird erläutert, welche Art LOD System in dieser Arbeit entwickelt werden soll.
Die verschiedenen Ansätze unterscheiden sich in einigen Kriterien. Für den Einsatz in einer Webanwendungen ist die Gewichtung der Kriterien anders als für Desktopanwendungen, Spiele oder dergleichen. So hat die Dateigrösse der Modelle bei Webanwendungen einen grösseren Einfluss auf die gefühlte Performanz beim Endanwender. Insofern ist die Auswirkung der gewählten Strategie auf die Downloadgrösse ein wichtiges Kriterium. Für diskrete LOD muss für jedes Level ein separates Modell geladen werden. Dies erhöht die Downloadgrösse merkbar. Bei kontinuierlichen LOD Systemen muss das Modell die Unterschiede zwischen verschiedenen Levels persistieren. Mit einem für kontinuierliche LOD ausgelegten Algorithmus kann der benötigte Speicher reduziert werden, indem beim *Edge Collapse* nicht auf einen neuen optimalen *Vertex* reduziert wird, sondern von den beiden bestehenden *Vertices* der passendere gewählt wird. Trotzdem entsteht für jeden *Edge Collapse* ein zusätzlicher Speicheraufwand. Insbesondere unter Berücksichtigung der Tatsache, dass Laufzeitperformanz essentiell für Webanwendungen ist, wurde entschieden nicht auf kontinuerliche LOD Systeme zu setzen. Zudem wurde entschieden kein hierarchisches LOD System zu entwickeln, um die Integration des Tools für die Entwickler einer Webanwendung möglichst einfach zu gestalten. Das diskrete LOD System ist für Grafiker und Entwickler die einfachste zu verwenden und deckt somit auch die breiteste Benutzerbasis und die meisten Anwendungsfälle ab. Aufgrund dessen, wird im Rahmen dieser Arbeit ein Algorithmus dieses Systems entwickelt und in den Entwicklungsprozess integriert. Eine Übersicht über die verschiedenen Kriterien kann in Tabelle 3.1 gefunden werden.

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

Tabelle 3.1.: Übersicht Merkmale der verschiedenen LOD Systeme

### 3.1.1. Bestehende Systeme

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

***Unreal***

Unreal verfügt sowohl über ein DLOD System als auch über ein HLOD System. Das HLOD Tool zeichnet sich insbesondere dadurch aus, dass es die verschiedenen HLOD Artefakte automatisch generieren kann. Dies ermöglicht es, komplexe Strukturen direkt innerhalb von Unreal zu kombinieren. [7]

***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. [8] Ansätze für ein HLOD System sind vorhanden, wurden jedoch nicht offiziell in Unity integriert. [9]

## 3.2. Surface Simplification Algorithmus

Die verschiedenen Klassen von Algorithmen werden in Unterabschnitt 2.5.2 kurz erläutert.
Verschiedene Algorithmen wurden für die Implementierung in Betracht gezogen. Die wohl weitverbreiteste Art ist der von Garland und Heckbert 1997 definierte *Surface Simplification using Quadric Error Metrics* welcher auf *Edge Collapses* basiert. [10]
Der Algorithmus kann wie folgt zusammengefasst werden:

1. Fehlermetrik für alle Vertices berechnen.
2. Optionen für *Edge Collapse* markieren.
3. Optionen mit geringstem Fehler optimieren.

### 3.2.1. Fehlermetrik

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

## 3.3. Pipeline Integration

Die Lösung soll in eine wiederverwendbare und konfigurierbare Pipeline integriert werden. In der Webentwicklung ist es üblich, über CLI [1] Arbeitsschritte zu automatisieren. Es soll demnach ein *CLI* erstellt und auf npm [2] öffentlich zugänglich gemacht werden, welches nahtlos in den Entwicklungsablauf integriert werden kann. Über eine Konfiguration soll angegeben werden können, wo nach den *.glTF* Dateien gesucht, wie die Output-Dateien benennt und in welchem Modus das Tool gestartet werden soll. Ebenso werden Algorithmus-Einstellungen in dieser Konfiguration gesetzt werden können. Es soll einfach zu bedienen, gut dokumentiert und gut gewählte Standardeinstellungen haben. Zudem soll das Tool im einmal Modus laufen können oder kontinuierlich sich ändernde Dateien neu optimieren.

## 3.4. Nutzen LOD

Um den Nutzen von LOD quantifizieren zu können, müssen verschiedene Aspekte berücksichtigt werden.

### 3.4.1. Aspekte für Nutzen

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

***Entwicklungszeit***

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

[1] Command Line Interface – Kommandozeile
[2] Node Package Manager

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.

#### *Downloadzeit*

Die Downloadzeit ist grundsätzlich linear abhängig von der Dateigrösse. Je grösser die Dateien, desto mehr Bandbreite wird verwendet. Dies hat direkte Auswirkungen auf die Zeit, bis eine Anwendung benutzt werden kann. Ziel ist es, die Downloadgrösse möglichst tief zu halten. Dank der heutigen Bandbreiten ist dies für viele Anwendungsbereiche ein kleinerwerdendes 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.

#### *Laufzeit*

Entscheidend ist, unabhängig von der Applikation, das Laufzeitverhalten. Eine Applikation, welche eine schlechte User Experience liefert, ist inakzeptabel. Das Laufzeitverhalten wird durch eine Vielzahl von Aspekten beeinflusst. Angestrebt werden grundsätzlich immer 60 FPS [3] . Wenn eine bestimmte Szenerie diese 60 *FPS* nicht erreicht, ist ein Performanzproblem vorhanden.

#### *Berücksichtigung der Aspekte*

In dieser Arbeit wird ein hoher Wert darauf gelegt, dass die Entwicklungszeit möglichst geringfügig verändert wird. Zudem werden, wo möglich, Empfehlungen getätigt, um die Downloadgrösse möglichst marginal zu verändern. 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. Ziel ist es, das Laufzeitverhalten zu analysieren und somit den maximal möglichen Einfluss von LOD auf die Leistung klassifizieren zu können. Hierbei werden verschiedene Aspekte 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 Szenerien und vergleicht die Ressourcennutzung der beiden Varianten [11].

### 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.

[3] Frames per Second. Bilder pro Sekunde. Gängige Art, die Performanz der Rendering Engine zu messen

#### *Modellbetrachtung*

Grundsätzlich kann der Nutzen beziffert werden, indem die Kosten zum Rendering eines Modelles berechnet werden. So können die Kosten der benutzten GPU Ressourcen durch das Messen des Aufwandes eines spezifischen *Draw Calls* beziffert werden. Die Implementation ist simpel und liefert genaue Resultate. Die Auswirkungen von Nebeneffekten können beinahe vollständig negiert werden. Sinnvoll wäre es, das Modell in konstanter Distanz zur Kamera um zwei Achsen drehen zu lassen und die Messung solange durchzuführen, bis eine ganze Umdrehung stattgefunden hat. Das originale Modell kann mit den verschiedenen Stufen verglichen werden 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. Beispiele für diese Aspekte sind: erhöhte Memory Auslastung durch die zusätzlichen Artefakte oder zusätzlich erforderliche Rechenleistung für berechnen des zu wählenden Artefakts.

#### *Szeneriebetrachtung*

Bei diesem Ansatz wird eine gegebene Szenerie möglichst geringfügig angepasst und die Auswirkungen auf die Performanz verglichen. Wenn die unoptimierte Variante 60 *FPS* erreicht oder gar übersteigt, ist eine Optimierung nicht notwendig. Somit muss eine Szenerie gewählt werden, welche diesen Wert unterschreitet. 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 Visualisierung aus der Vogelperspektive mit Zoomverhalten. Im Vergleich zur Modellbetrachtung ist die Aussagekraft eines solchen Tests ausreichend, um einen Beleg für den positiven Nutzen des Systems zu haben. Sollte es nicht möglich sein, die *FPS* zu steigern, so liefert der Einsatz von LOD keinen nachweisbaren Nutzen.

# 4. Resultate

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

## 4.1. LOD Pipeline

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

### 4.1.1. Workflow

Der übliche Ablauf für das generieren von LOD Artefakten ist wie folgt: Für ein gegebenes Modell werden innerhalb vom Modellierungstool wie z.B. Blender bestimmte LOD Stufen von Hand generiert. Anschliessend werden die verschiedenen Stufen exportiert und manuell in die Applikation integriert. Fine Tuning erfordert so sowohl Anpassungen an den Modellen als auch im Code.

### 4.1.2. lode Pipeline

Das Ziel von *lode* ist es, für ein möglichst breites Spektrum von Anwendungsfällen eine einfache Lösung anzubieten und somit die Hemmschwelle für den Einsatz von LOD Artfakten zu reduzieren.
Deshalb setzt *lode* konsequent auf moderne Entwicklungsprozesse, manuelle Schritte sollen auf ein Minimum reduziert werden.

### 4.1.3. lode Ablauf

Ein Modell wird erstellt und als glTF innerhalb des Projekts gespeichert. Eine zum Modell gehörige Konfigurationsdatei wird angelegt und *lode* generiert die darin beschriebenen LOD Artfakte. Die Artfakten können innerhalb der Applikation mithilfe einer Bibliothek geladen werden. Bei Anpassungen an der Konfigurationsdatei werden die neuen Detailstufen automatisch generiert. Das zugehörige *lode-ui* bietet zudem die Möglichkeit die verschiedenen Detailstufen miteinander vergleichen zu können und die Konfiguration möglichst einfach zu justieren.

## 4.2. LOD Generierung

Für das Erstellen von LOD Artefakten wurde ein 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 [1] , umgesetzt. Grund dafür ist das einbinden von *lode* in bestehende Entwicklungsabläufe in der Webentwicklung. So ist es einfach möglich das Paket kostenlos mithilfe von npm zur Verfügung zu stellen.

### 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. Texturen

Bei vielen Modellen sind der grösste Teil der Speichergrösse die Texturen. Das Optimieren der geometrischen Struktur reduziert die zugehörigen Texturen jedoch nicht. Um trotzdem möglichst grosse Einsparungen für die Texturen zu ermöglichen wurde eine Methode entwickelt, welche versucht die prominenteste Farbe eines Modelles zu extrahieren und diese für die Vereinfachungen zu verwenden. 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.

## 4.3. Benchmark

In diesem Abschnitt wird das Benchmarksetup erläutert. Die verschiedenen technischen Entscheide werden detailliert aufgezeigt. 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.

[1] JavaScript Ausführungsumgebung welche ohne einen Browser verwendet werden kann

### *Browser Umgebung*

Um den Umfang des Benchmarks überschaubar zu halten, wurde ausschliesslich ein Benchmark für Google Chrome entwickelt. Google Chrome basiert auf Chromium [2], dieselbe Engine, welche auch Microsoft Edge oder Opera verwenden. Einen Benchmark basierend auf Google Chrome liefert somit auch Indizien für diese beiden Browser, auch wenn gewisse Abweichungen möglich sind. Neben dem Marktführer Chrome sind Mozilla Firefox oder Safari von Apple ebenfalls Optionen. Jedoch wurde primär aufgrund des Marktanteils von total rund 70% [12] zugunsten von Google Chrome entschieden. Die getroffenen Aussagen bezüglich Laufzeitverhalten behalten ihre Gültigkeit auch für andere Browser.

### *Automation*

Um die Tests durchzuführen, wird ein Testautomationstool benötigt; unter anderem der Einsatz von Selenium wurde in Erwägung gezogen. Der Vorteil von Selenium ist insbesondere, dass der Benchmark für weitere Browser ausgeweitet werden könnte. Da jedoch das Analysieren der GPU Daten stark vom System abhängig ist und dafür zusätzliche Komplexität notwendig wäre, wird in diesem Benchmark die im Google Chrome integrierten *Chrome DevTools* eingesetzt. Selenium bietet zurzeit eine suboptimale Integration für das *Chrome DevTools Protocol*. Um mögliche Diskrepanzen zwischen Systemen möglichst gering zu halten, wurde jedoch entschieden, auf die bewährte Lösung von Google Chrome zu setzen. Puppeteer [3], eine weitere Option für die Automation, ist eine Bibliothek, die eine vereinfachte Schnittstelle zu einer Chromium Instanz bietet. Sie wird zudem direkt von Google entwickelt und bietet somit eine stabile Grundlage zur Kommunikation mit den *Chrome DevTools*.

### *Profiling*

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

### *Testaufbau*

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

1. Öffne die Applikation in einer *Chromium* Instanz.
2. Warte bis die Seite geladen ist.

[2] Open Source Browser-Projekt welches von Google entwickelt wird.
[3] Node.js Library, die eine API anbietet zum Steuern von Chromium oder Chrome über das Chrome DevTools Protocol

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.

| Kennzahl | Beschreibung |
|---|---|
| Median *FPS* | Die *FPS* 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 signifkant 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 Render Loop Dauer | Der Median aller Laufzeiten der Render Loop. |
| Anzahl *Render Loop* Iterationen | Wie oft wurde ein neues Bild gezeichnet. Umso höher die Zahl, desto mehr verschiedene Frames konnten gerendert werden. |
| Totale GPU Zeit | Die totale Zeit, welche die GPU für Berechungen benötigt. |
| Anzahl GPU Events | Anzahl der Events an die GPU. Dieser Wert soll lediglich dazu dienen um die gemessenen *FPS* und die Anzahl *Render Loop* Iterationen besser einschätzen zu können. Eine höhere Anzahl an GPU Events steht innerhalb der Demoszenerie im Zusammenhang mit mehr *Render Loop* 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.

In Abbildung 4.1 ist ein Screenshot der Testapplikation ersichtlich. Die Kamera wird kontinuierlich innerhalb der Szene bewegt. Die Kamera wird abhängig von der Systemzeit positioniert. 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.

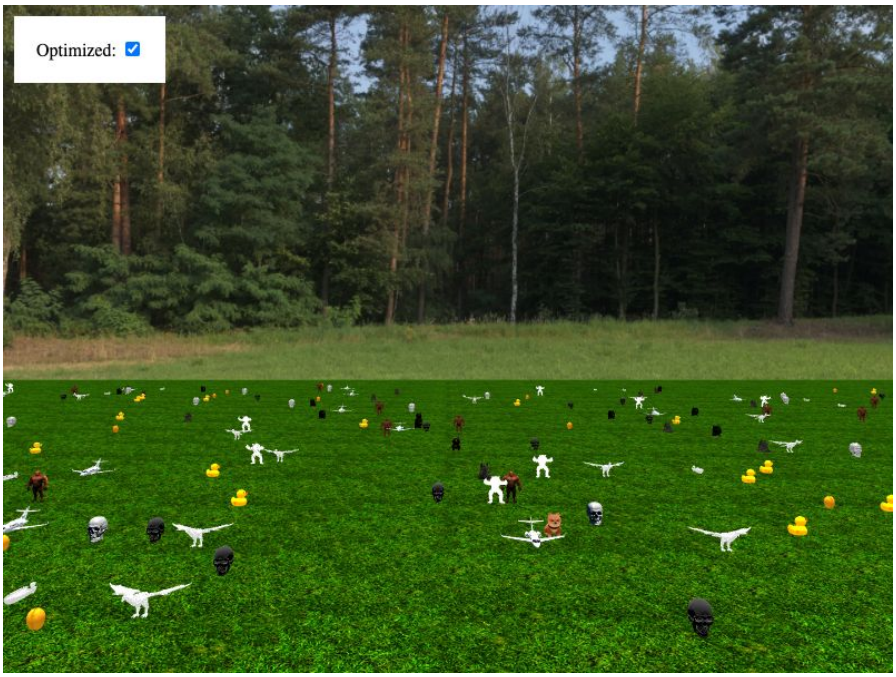


Abbildung 4.1.: Testapplikation

### *Testumgebung*

Um während den Tests möglichst faire Bedingungen zu gewährleisten, wird die Maschine zuvor wie bei anderen Benchmarks vorbereitet. Ziel ist es, Seiteneffekte zu minimieren. Für diesen Benchmark wurden deshalb die Instruktionen von *Tracer Bench* zur Behebung von Rauschen befolgt. [13]

### *Analyse der Daten*

Um eine zuverlässige Aussage treffen zu können, wird der Vorgang mehrfach wiederholt. Für jeden Durchlauf wird der Median der *FPS* Daten berechnet. Anschliessend wird die Varianz der *FPS* für die unoptimierten respektive optimierten Werte berechnet. Die Varianz dient als Kennzahl, um eine Signifikanz der Daten nachweisen 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. Somit wird gewährleistet, dass das Resultat des Benchmarks verlässlich ist und für einen Vergleich verwendet werden kann.

# 5. Diskussion und Ausblick

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

## 5.1. Mögliche Erweiterungen

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

#### *glTF LOD Format*

Es gibt einen Proposal für die Definition von LODs innerhalb von glTF zu definieren. [14] Zur Zeit hat Three.js jedoch noch keinen Support für diese Extension. Auch Babylon.js unterstützt lediglich das progressive Laden von Modellen. [15]

#### *Progressives Laden*

Aktuell werden initial alle Level of Details geladen. Dies hat zur Konsequenz, dass im Vergleich zur unoptimierten Variante eine schlechtere initiale Ladezeit resultiert. Um dieses Verhalten zu verbessern beziehungsweise sogar in einen Vorteil zu verwandeln könnte das System so erweiteret werden, dass die Modelle progressiv geladen werden.

#### *Levelwahl während Laufzeit*

Die optimale Levelwahl kann stark von der Laufzeitumgebung abhängig sein. So sollte zum Beispiel auf mobilen Geräten früher ein einfacheres Modell gezeigt werden als auf leistungsstarken Geräten. Deshalb eignet sich eine Wahl der Levels zur Laufzeit, um ein optimales Erlebnis auf allen Geräten zu ermöglichen. Nebst dem *CLI* soll folglich auch ein Javascript-Modul bereitgestellt werden, welches das Laden von glTF Dateien übernimmt und basierend auf der erkannten Performanz das beste Level of Detail einspielen.

# 6. Verzeichnisse

# Glossar

**Chromium** Open Source Browser-Projekt welches von Google entwickelt wird.. 30, 36

**CLI** Command Line Interface – Kommandozeile. 25, 34

**FPS** Frames per Second. Bilder pro Sekunde. Gängige Art, die Performanz der Rendering Engine zu messen. 26, 27, 31, 32

**Game Engines** Framework, das für den Spielverlauf und dessen Darstellung verantwortlich ist. 7

**Node.js** JavaScript Ausführungsumgebung welche ohne einen Browser verwendet werden kann. 29

**npm** Node Package Manager. 25, 29

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

**Point Cloud** Vertices im dreidimensionalen Raum ohne zugehörige *Triangle*. 21

**Puppeteer** Node.js Library, die eine API anbietet zum Steuern von Chromium oder Chrome über das Chrome DevTools Protocol. 30

**Rendering Engine** Teilprogramm, das zuständig für die Darstellung von Grafiken ist. 7, 26, 30, 36

**Scene Graph** Objektorientierte Datenstruktur zur Beschreibung von 2D- oder 3D-Szenarien. 20

**Texture Mapping** Verfahren, mittels 2D-Bildern 3D-Objekte zu gestalten. 11

# Literaturverzeichnis


[1] D. Jackson und J. Gilbert. (2021) *WebGL Specification.* [Online]. URL: https://www.khronos.org/registry/webgl/specs/latest/1.0/ [Stand: 26.03.2021].

[2] GPU for the Web Community Group. (2021) *GPU for the Web Community Group Charter.* [Online]. URL: https://gpuweb.github.io/admin/cg-charter.html [Stand: 26.03.2021].

[3] M. Segal, K. Akeley, C. Frazier, J. Leech und P. Brown. (2008) *The OpenGL®Graphics System: A Specification.* [Online]. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec30.pdf [Stand: 26.03.2021].

[4] Babylon.js contributors. (2021) *Simplifying Meshes With Auto-LOD.* [Online]. URL: https://doc.babylonjs.com/divingDeeper/mesh/simplifyingMeshes [Stand: 04.05.2021].

[5] unknown. (2007) *Object Files.* [Online]. URL: http://www.martinreddy.net/gfx/3d/OBJ.spec [Stand: 26.03.2021].

[6] P. Cozzi und T. Parisi. (2016) *glTF 1.0 Specification.* [Online]. URL: https://github.com/KhronosGroup/glTF/tree/master/specification/1.0 [Stand: 26.03.2021].

[7] S. Deiter. (2018) *An In-Depth Look at Unreal Engine 4.20's Proxy LOD Tool.* [Online]. URL: https://www.unrealengine.com/en-US/tech-blog/an-in-depth-look-at-unreal-engine-4-20-s-proxy-lod-tool [Stand: 03.05.2021].

[8] M. Edlund. (2021) *Mesh simplification for Unity.* [Online]. URL: https://github.com/Whinarn/UnityMeshSimplifier [Stand: 03.05.2021].

[9] A. Ebrahimi. (2018) *Unity Labs: AutoLOD – Experimenting with automatic performance improvements.* [Online]. URL: https://blogs.unity3d.com/2018/01/12/unity-labs-autolod-experimenting-with-automatic-performance-improvements/ [Stand: 03.05.2021].

[10] M. Garland und P. S. Heckbert. (1997) *Surface Simplification Using Quadric Error Metrics.* [Online]. URL: https://www.cs.cmu.edu/~./garland/Papers/quadrics.pdf [Stand: 17.04.2021].

[11] O. Nordquist und A. Karlsson. (2017) *BabylonJS and Three.js - Comparing performance when it comes to rendering Voronoi height maps in 3D.* [Online].

URL: http://www.diva-portal.org/smash/get/diva2:1228221/FULLTEXT01.pdf [Stand: 17.04.2021].

[12] StatCounter. (2021) *Browser Market Share Worldwide | StatCounter Global Stats*. [Online]. URL: https://gs.statcounter.com/browser-market-share#monthly-202003-202103 [Stand: 14.04.2021].

[13] M. Lynch. (2020) *TracerBench: Noise Mitigation Techniques*. [Online]. URL: https://github.com/TracerBench/tracerbench/blob/master/NOISE_MITIGATION.md [Stand: 17.04.2021].

[14] S. Bhatia, G. Hsu, A. Gritt, J. Copic, M. Appelsmeier und D. Frommhold. (2018) *glTF Extension MSFT lod*. [Online]. URL: https://github.com/KhronosGroup/glTF/tree/master/extensions/2.0/Vendor/MSFT_lod [Stand: 28.04.2021].

[15] Babylon.js contributors. (2021) *Progressively Load .glTF Files in Babylon.js*. [Online]. URL: https://doc.babylonjs.com/divingDeeper/importers/progressiveglTFLoad [Stand: 28.04.2021].

# Abbildungsverzeichnis

# Tabellenverzeichnis

# A. Anhang

## A.1. Aufgabenstellung

Aufgabenstellung

## A.2. Anhang 2

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.