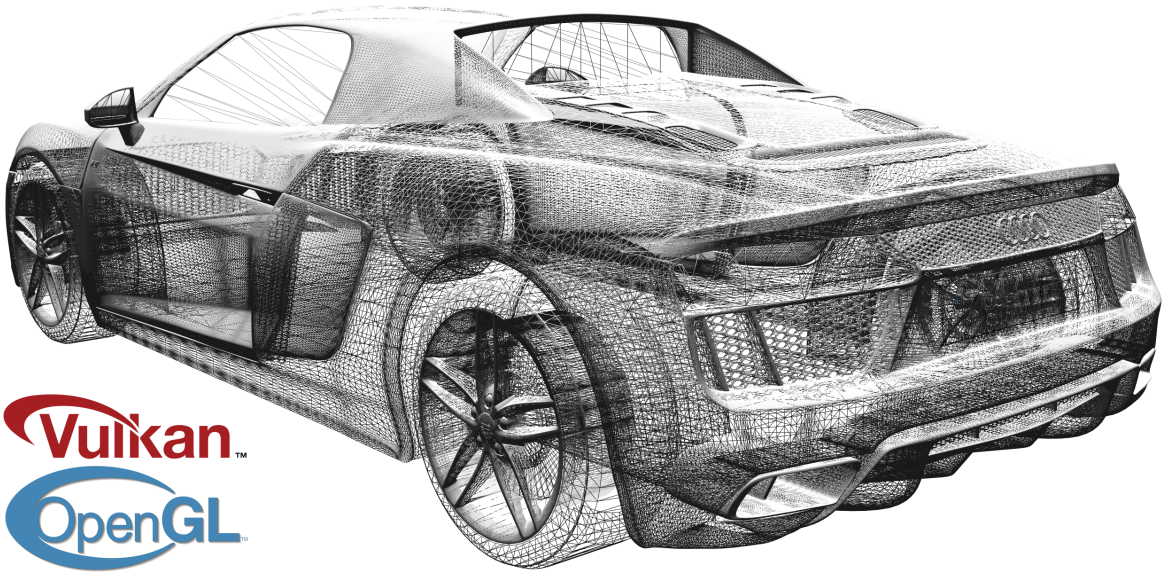
Maturitätsarbeit an der Kantonsschule Zürich Nord

3D-Rendering und Floating-Point: Vulkan und OpenGL als Grafik-API's im Vergleich

Cedric Schwyter
M6e
Betreut von:
Christian Prim

02. Dezember 2019



Inhaltsverzeichnis

1 Vorwort

Da sich diese Arbeit auf einem ein wenig abstrakterem Niveau als dem typischen Anfänger-Programmierlevel bewegt, wird minimales technisches und informatisches Grundverständnis vorausgesetzt, da es relativ unmöglich und völlig ausserhalb des Rahmens dieser Arbeit ist, die gesamten programmiertechnischen Basics zu erklären, welche hier angewendet werden. Die mathematische und physikalische Theorie zu den wichtigen Teilbereichen wird jedoch selbstverständlich erklärt, wobei auch hier ein basales mathematisches Verständnis und gewisse Grundkenntnisse vorausgesetzt werden, vor allem in der linearen Algebra, der Vektorgeometrie und der Analysis mit ihrer Differentialrechnung.

Weiterhin möchte ich an dieser Stelle meinen ausdrücklichen Dank an meinen lieben Kollegen Charpoan Kong, welcher mir ein wenig mit den Makefiles und dem Port zu Linux geholfen hat, und an meinen engagierten Betreuer Christian Prim, welcher meine Formeln stets zu korrigieren wusste, anbringen.

Dieses Thema wurde ausgewählt, da ich schon Vorwissen und Erfahrung in der Grafikprogrammierung hatte, wobei ich in OpenGL heimischer war als in Vulkan. Ich habe bereits mehrere OpenGL-Engines in mehreren Programmiersprachen (C++, Java), jedoch erst eine einzige Vulkan-Engine (C++) geschrieben. Ich persönlich sah es als Herausforderung für mich selbst, eine fähige Engine mit Vulkan zu erstellen und ich glaube, dass ich diese auch gemeistert habe. Das Ganze wurde mit dem Hintergedanken der Floating-Point-Ungenauigkeit angeschaut, welche oft ungewöhnliche Effekte nach sich zieht. Als Beispiel dafür gab es in diesem Projekt einen unbeabsichtigten Fall, bei welchem in einer Projektionsmatrix zwei Floats von 8-byte- in 4-byte-Datentypen umgewandelt und erst nach dieser Umwandlung durcheinander geteilt wurden. Dadurch ging, vor allem bei kleinen Werten, sehr viel an Genauigkeit verloren. Dies führte zu irritierenden Verzerrungen der Projektion des dargestellten Raumes. Auf ein chaotisches System wie ein Doppelpendel angewandt entstehen durch Floating-Point-Ungenauigkeiten sicherlich spannende Dinge. Dies war die Überlegung, welche sich stärker als erhofft bewahrheitet hat und auf Umwegen sogar dazu führte, dass ich mich sehr oberflächlich mit dem Lagrange-Formalismus auseinandergesetzt habe.

Das Titelbild wurde mit der Vulkan-Engine gerendert wobei mit internen Einstellungen der Polygon Mode verändert wurde, sodass das Modell nur als Netz aus Linien dargestellt wurde.

2 Einleitung

Bei Vulkan und OpenGL handelt es sich um zwei High-Performance-Grafik-API's. Die beiden sollen in dieser Arbeit in der Theorie und vor allem in der Praxis verglichen werden. Über ein System des deterministischen Chaos sollen allfällige Unterschiede bezüglich der Floating-Point-Ungenauigkeit der Endprodukte festgestellt werden.

2.1 Konkrete Fragestellungen

In dieser Arbeit wurde mit dem Fokus auf die folgenden Fragestellungen gearbeitet:

- Ist OpenGL immer noch eine geeignete Option im Jahr 2019?
- Was sind die Vor- und Nachteile von Vulkan gegenüber von OpenGL? Ein theoretischer Vergleich.
- Programmierung von zwei Render-Engines mit der jeweiligen API und einen Vergleich in der Praxis durchführen.
- Lässt sich ein Unterschied der beiden Engines bezüglich eines Systems des deterministischen Chaos (Doppelpendel) erkennen und wo könnten die Ursachen dafür liegen?

2.2 Zielsetzungen

Persönliche Zielsetzungen umfassten hierbei die korrekte und sinnvolle Programmierung der zwei Render-Engines. Die zwei Applikationen sollten vorzeigbare Resultate abliefern können und keine direkt erkennbaren Bugs und Fehler haben. Ausserdem sollten sie logisch, dynamisch und mit Möglichkeiten der Weiterentwicklung programmiert werden, und nicht nur im Hinblick auf das deterministische Chaos. Dies war der persönliche Hauptfokus. Das System des deterministischen Chaos soll ausserdem physikalisch korrekt dargestellt werden.

2.3 Vorgehen

In einem ersten Teil wurden die zwei Render-Engines in C++17 programmiert. Dieser erste Teil umfasste circa 90% der gesamten Arbeit. Insgesamt wurden rund 12'500 Zeilen Quellcode geschrieben, wobei jede einzelne von Hand getippt und alle Systeme selbst umgesetzt wurden. In einem zweiten Teil wurde dann das Doppelpendel-System in beiden Applikationen umgesetzt und auf mehreren Systemen getestet. Während der Entwicklung beider Programme wurde ein Arbeitsjournal geführt. Auf GitHub sind die beiden Projekte mit Installationsanleitung zu finden (siehe Anhang A).

2.4 Eigene Vorraussagen

Da Vulkan viel neuer als OpenGL ist wurde angenommen, dass die Vulkan-Implementation performancetechnisch überlegen ist. Da noch nicht so viel Erfahrung mit Vulkan vorhanden war, wurde erwartet, dass Probleme mit der Implementation nicht immer auf dem besten Wege gelöst werden können. Dennoch wurde bei der Annahme, dass das Resultat von Vulkan besser als OpenGL's sein würde, geblieben. Grössere Schwierigkeiten bei der Entwicklung der Applikationen wurden nicht erwartet, da schon Erfahrung in der Grafikprogrammierung mit beiden API's vorhanden war. Für die Implementation des Doppelpendels wurde angenommen, dass dort physikalisch interessante Probleme auftreten könnten.

3 Grundlagen von 3D-Rendering

Als Rendering wird ein computertechnisches Verfahren bezeichnet, bei dem aufgrund von Berechnungen primär visuelle Ergebnisse produziert resp. Raum-Situationen simuliert werden. Rendering als Begriff wird allerdings auch oft im Zusammenhang der Nachbearbeitung von Video-/Audiomaterial verwendet.

Im Kontext von 3D-Rendering spricht man von einer Simulation einer dreidimensionalen Welt/Umgebung mit Licht und allfälligen Schattenberechnungen, Farbberechnungen, Post Processing Effekten, Wellen-/Rauch-/Windsimulationen und vielem mehr. Zentral steht hierbei die Vektorgeometrie für Farb- und Lichtberechnungen, sowie die Matrizenrechnung mit ihren Transformationsmatrizen.

3.1 Was ist eine Grafik-API?

Als Application Programming Interface (API) wird im Allgemeinen eine Programmierschnittstelle bezeichnet, welche das Ansteuern von Software oder Hardware direkt aus dem Quellcode erlaubt. Diese Soft- oder Hardware kann sich auf demselben Computersystem, aber auch auf einem Servercluster am anderen Ende der Welt befinden. Die Schnittstelle vereinfacht vor allem das Benutzen von externen, aber auch systeminternen Computations- und Informationsbeschaffungsmöglichkeiten. Viele Technikunternehmen stellen deswegen auch eine eigene API bereit, um Programmierern die Integration der angebotenen Dienste zu vereinfachen.[1] Bekannte Beispiele dafür wären die Google Maps API, die YouTube API oder die NASA API.

Im Sinne der Grafikprogrammierung wird unter einer Grafik-API eine Vereinfachung der Ansteuerung von Computing-Potenzial auf Grafikkarten verstanden. Grafikkarten sind auf hochgradig parallelisierte Computation ausgelegte High-Performance-Prozessorchips. Parallelisierung meint dabei die simultane Durchführung der ein- und derselben Berechnungen auf einigen hundert bis zu mehreren tausend Rechenkernen für übliche Endbenutzer-Grafikkarten.[2] Dadurch, dass die Berechnungen gleichzeitig tausendfach durchgeführt werden können, müssen sie nicht nacheinander ausgeführt werden und sparen somit viel wertvolle Zeit.

3.2 Repräsentation von 3D-Modellen

3D-Modelle werden typischerweise als eine Reihe von Punkten (die *Vertices*), welche durch Linien (die *Edges*) miteinander verbunden werden und somit Flächen (die *Faces*) bilden, repräsentiert. Die Faces werden weiter in *Fragments* gerastert. Diese entsprechen einfärbbaren Punkten auf dem Face. Die Farbe des jeweiligen Fragments wird im Fragment Shader berechnet. Den Zusammenhang Vertex, Edge und Face kann man sich wörtlich übersetzt als Ecke, Kante und Fläche vorstellen. Auf dem Titelbild sind die Edges eines Modells, welches aus 1'036'416 Vertices besteht, dargestellt. Je nach Geometrie des Modells wird dieses zu Dreiecken, Vierecken oder anderen Polygonen verbunden. Je nach Modell kann die Wahl der Geometrie eine Rolle spielen, da z.B. Würfel nicht unbedingt aus Dreiecken geformt werden müssen, sondern Vierecke ausreichen. Somit hätte man weniger Faces (nämlich nur 6 anstatt 12), was Rechenaufwand einsparen würde. Am weitesten verbreitet sind jedoch trotzdem Dreiecke, da sich grössere Polygone immer aus mehreren kleineren Dreiecken formen lassen. Die dreieckigen Faces können dann eingefärbt und texturiert werden. In einer Modelldatei befinden sich also eine Reihe von 3D-Punkten und zusätzliche Informationen. Weiter können zum Beispiel Texturkoordinaten, Normalenvektoren der Faces, Farben, Tangenten/Bitangenten oder sonstige Zusatzdaten in den Modelldateien abgelegt werden. Texturkoordinaten bestimmen, welcher Teil einer Textur auf welchen Teil des Faces gemappt (also projiziert) wird. Die Normalenvektoren zum Beispiel werden für die später erklärten Lichtberechnungen benötigt. Der Rest war für die Applikationen irrelevant. Die beiden Engines wurden mithilfe der Model-Loading-Library ASSIMP geschrieben, welche verschiedenste 3D-Modellformate laden und parsen kann, wobei das wichtigste und verbreitetste davon das *.obj*-Format ist.

[1] Vangie Beal. *API - Application Program Interface. (Englisch) [API - Applikations-Programmierschnittstelle]*. Aufgerufen: 21.11.2019. URL: `https://www.webopedia.com/TERM/A/API.html`.

[2] Christine McKee. *CUDA Cores in Video Cards. (Englisch) [CUDA-Kerne in Grafikkarten]*. Aufgerufen: 21.11.2019. URL: `https://www.lifewire.com/what-is-nvidia-cuda-834095`.

3.3 Mathematische Grundlagen

Vorkenntnisse in der linearen Algebra und der Vektorrechnung werden vorausgesetzt um die folgenden Sachverhalte zu verstehen.

Die erste Schwierigkeit, die beim Rendern von 3D-Räumen auftaucht, ist die Darstellung auf einem 2D-Bildschirm. Irgendwie müssen dreidimensionale Raumkoordinaten auf zweidimensionale Bildschirmkoordinaten projiziert werden. Dies stellt sich als fundamentale Frage, denn, wenn dies nicht geklärt werden kann, können zwangsweise keine visuellen Ergebnisse entstehen. Die Werkzeuge, um dieses Problem zu lösen bietet die Matrizenrechnung mit Transformationsmatrizen. In der Grafikprogrammierung werden nach Konvention 4-dimensionale Vektoren folgender Form verwendet, wobei w standardmässig auf 1 gesetzt wird, sofern keine spezielle Verwendung für diese vierte Komponente besteht:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Das heisst, alle dreidimensionalen Vektoren in der Grafikrechnung können ebenfalls als vierdimensionale Vektoren, deren vierte Komponente 1 ist, dargestellt werden. Zum Beispiel werden Farbwerte mit 3 Kanälen (RGB) durch so einen Vektor mit $w = 1$, und Farbwerte mit 4 Kanälen (RGBA) als normalen vierdimensionalen Vektor (bei welchem w dem Wert des vierten Kanals entspricht) repräsentiert. Dieser Konvention und den mathematischen Rechengesetzen zufolge werden auch 4×4-Matrizen verwendet da die Matrixmultiplikation mit einem Vektor sonst nicht definiert wäre. Zunächst werden die benötigten Transformationen für die nachfolgenden theoretischen Betrachtungen erläutert.[3]

3.3.1 Skalierung

Ein Vektor kann mit einer Matrix wie folgt um den Skalierungsvektor $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ skaliert werden:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x \\ b \cdot y \\ c \cdot z \\ 1 \end{pmatrix}$$

3.3.2 Translation

Eine Translation um den Verschiebungsvektor $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ sieht folgendermassen aus:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a + x \\ b + y \\ c + z \\ 1 \end{pmatrix}$$

Die Translation bezeichnet grundsätzlich die Addition zweier Vektoren. Aus diesem Zusammenhang folgt sogleich, dass sich jeder 3D-Vektor auch als eine 4×4-Matrix ausdrücken lässt. Dies lässt sich ganz einfach darstellen wenn x, y und $z = 0$ sind:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a + 0 \\ b + 0 \\ c + 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ 1 \end{pmatrix} \widehat{=} \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

[3] Joey de Vries. *Learn OpenGL - Transformations. (Englisch) [Lerne OpenGL - Transformationen]*. Aufgerufen: 24.10.2019. URL: `https://learnopengl.com/Getting-started/Transformations`.

Später wird dann teilweise nur noch mit 3-Komponenten-Vektoren gerechnet, welche einfach als 4-Komponenten-Vektoren mit vierter Komponente $w = 1$ angeschaut werden können. Das wird in der Grafikprogrammierung zur Vereinfachung manchmal gemacht.

3.3.3 Rotation

Ein Vektor kann um eine (beliebige) Achse gedreht werden. Durch eine Transformationsmatrix ausgedrückt, sehen Rotationen wie folgt aus:

Um die x-Achse $\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$ mit Winkel θ:

$$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}x\\ \cos\theta\cdot y - \sin\theta\cdot z\\ \sin\theta\cdot y + \cos\theta\cdot z\\ 1\end{pmatrix}$$

Um die y-Achse $\begin{pmatrix}0\\1\\0\\1\end{pmatrix}$ mit Winkel θ:

$$\begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}\cos\theta\cdot x + \sin\theta\cdot z\\ y\\ -\sin\theta\cdot x + \cos\theta\cdot z\\ 1\end{pmatrix}$$

Um die z-Achse $\begin{pmatrix}0\\0\\1\\1\end{pmatrix}$ mit Winkel θ:

$$\begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{pmatrix}x\\y\\z\\1\end{pmatrix} = \begin{pmatrix}\cos\theta\cdot x - \sin\theta\cdot y\\ \sin\theta\cdot x + \cos\theta\cdot y\\ z\\ 1\end{pmatrix}$$

Um eine beliebige Achse $\begin{pmatrix}a\\b\\c\\1\end{pmatrix}$ für die gilt $\left|\begin{pmatrix}a\\b\\c\end{pmatrix}\right| = 1$ mit Winkel θ:

$$\begin{bmatrix}\cos\theta + a^2\cdot(1-\cos\theta) & a\cdot b\cdot(1-\cos\theta) - c\cdot\sin\theta & a\cdot c\cdot(1-\cos\theta) + b\cdot\sin\theta & 0\\ b\cdot a\cdot(1-\cos\theta) + c\cdot\sin\theta & \cos\theta + b^2\cdot(1-\cos\theta) & b\cdot c\cdot(1-\cos\theta) - a\cdot\sin\theta & 0\\ c\cdot a\cdot(1-\cos\theta) - b\cdot\sin\theta & c\cdot b\cdot(1-\cos\theta) + a\cdot\sin\theta & \cos\theta + c^2\cdot(1-\cos\theta) & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \cdot \begin{pmatrix}x\\y\\z\\1\end{pmatrix}$$

$$= \begin{pmatrix}x\cdot(\cos\theta + a^2\cdot(1-\cos\theta)) + y\cdot(a\cdot b\cdot(1-\cos\theta) - c\cdot\sin\theta) + z\cdot(a\cdot c\cdot(1-\cos\theta) + b\cdot\sin\theta)\\ x\cdot(b\cdot a\cdot(1-\cos\theta) + c\cdot\sin\theta) + y\cdot(\cos\theta + b^2\cdot(1-\cos\theta)) + z\cdot(b\cdot c\cdot(1-\cos\theta) - a\cdot\sin\theta)\\ x\cdot(c\cdot a\cdot(1-\cos\theta) - b\cdot\sin\theta) + y\cdot(c\cdot b\cdot(1-\cos\theta) + a\cdot\sin\theta) + z\cdot(\cos\theta + c^2\cdot(1-\cos\theta))\\ 1\end{pmatrix}$$

Diese Transformationsmatrizen werden sehr schnell sehr mühsam. Um dies anschaulich darzustellen wurde die Rotationsmatrix für eine Drehung um eine beliebige Achse abgebildet. Deswegen wird die lineare Algebra in beiden Applikationen von einer Bibliothek namens OpenGL Mathematics (GLM) übernommen, damit keine eigene Lösung implementiert werden muss.

Durch Matrizen lassen sich Vektoren also manipulieren und transformieren, wie man will. Durch die Matrizenmultiplikation können mehrere Transformationen miteinander verknüpft und in einer einzigen Matrix dargestellt werden. Durch diese Möglichkeiten der Veränderung von Vektoren kann die nachfolgend erklärte Übersetzung von 3D zu 2D durchgeführt werden. Zuerst jedoch müssen Koordinatensysteme eingeführt werden, welche oft benötigt werden.

3.4 Koordinatensysteme

In der Grafikprogrammierung trifft man verschiedene Koordinatensysteme an, welche allesamt für gewisse Operationen benötigt werden. Sie werden allesamt auf Englisch bezeichnet. Die MVP-Matrizen, welche in der Folge erklärt sind, werden auch auf Englisch bezeichnet. Es werden also die folgenden Koordinatensysteme benötigt:

- Local Space
- World Space
- View Space
- Clip Space
- Screen Space

Sie verhalten sich zueinander wie Abbildung 1 zeigt und werden im nächsten Unterkapitel anhand der tatsächlichen Anwendung erklärt.

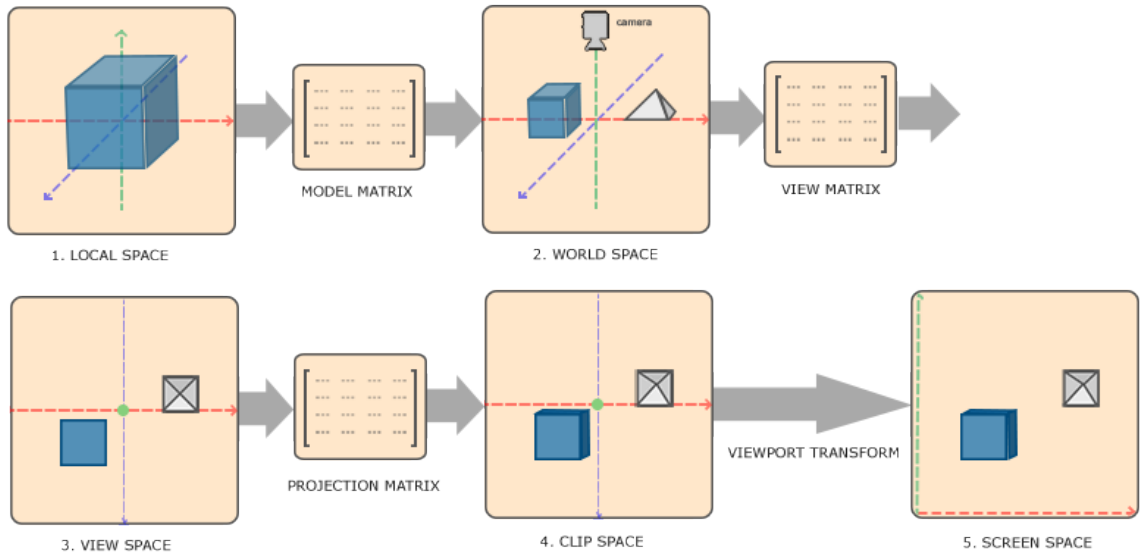


Abbildung 1: Koordinatensysteme und MVP-Transform[4]

3.5 Anwendung für 3D- zu 2D-Transformation

Wie kann dieses Vorwissen nun dazu dienen, 3D-Raumkoordinaten in 2D-Bildschirmkoordinaten umzurechnen? Angenommen, man hat ein 3D-Modell welches man rendern möchte. Dazu müssen nun die 3D-Vertices in 2D-Punkte transformiert werden. Dies läuft nach folgendem Schema ab: Die Vertices

[4] Joey de Vries. *Learn OpenGL - Coordinate Systems. (Englisch) [Lerne OpenGL - Koordinatensysteme]*. Aufgerufen: 24.10.2019. URL: `https://learnopengl.com/Getting-started/Coordinate-Systems`.

des Modells sind relativ zu seinem Ursprung, der meist innerhalb des Modells selbst liegt, angegeben. Dies nennt sich Local Space. Diese Koordinaten werden mithilfe der Model-Matrix (M) in World Space-Koordinaten umgewandelt. Die Model-Matrix bestimmt wo ($\widehat{=}$ Translation), mit welcher Grösse ($\widehat{=}$ Skalierung) und mit welcher Ausrichtung ($\widehat{=}$ Rotation) das Modell schlussendlich im Raum landet. Die View-Matrix (V) rechnet diese World Space-Koordinaten in ihre View Space-Äquivalente um. View Space bezeichnet die Koordinaten aus der Sicht der Kamera, ohne jegliche perspektivische oder orthographische Projektionen. Die perspektivische oder orthographische Projektion übernimmt die Projection Matrix (P); sie verwandelt die View Space-Koordinaten in Clip Space-Koordinaten. 'Clip Space', weil die Kamera nur einen gewissen Ausschnitt des gesamten Raumes zeigen kann, welcher im sogenannten *View Frustum* liegt, der Rest wird 'geclipped' (= weggeschnitten). Dies nennt sich auch Model-View-Projection-Transform (oder kurz MVP-Transform). Durch den Viewport Transform, welcher die Auflösung der Ausgabe und den festgelegten Ausschnitt durch den Benutzer bestimmt, werden Clip Space dann schlussendlich zu Screen Space-Koordinaten, den finalen Bildschirmkoordinaten. Diese geben an, auf welchem Pixel des Displays ein Vertex zu liegen kommen muss, um eine realistische Darstellung des 3D-Raumes zu erhalten. In Abbildung 1 ist dies anschaulich dargestellt.

Durch die Nicht-Kommutativität der Matrixmultiplikation müssen die Matrizen in der richtigen Reihenfolge miteinander multipliziert werden. Diese wäre aufgrund der Definition der Matrizenmultiplikation sogesehen 'rückwärts':

$$T_{\text{res}} = P \cdot V \cdot M$$

Diese Gleichung konstituiert die fundamentalen Prinzipien der Grafikprogrammierung. Mit ihr lässt sich die resultierende Transformationsmatrix T_{res} berechnen. Angewandt auf die tatsächlichen Positions-/Ortsvektoren $\vec{OV}$ der Vertices, wie sie in der 3D-Modelldatei abgespeichert sind, entspricht dies:

$$\vec{P}_{\text{res}} = T_{\text{res}} \cdot \vec{OV} = P \cdot V \cdot M \cdot \vec{OV}$$

$\vec{P}_{\text{res}}$ entspricht dabei den resultierenden Screen Space-Koordinaten. Auch hier handelt es sich um einen 4-dimensionalen Vektor, von welchem jedoch prinzipiell nur die ersten zwei Komponenten benötigt werden. Der Rest wird deswegen wie gewohnt ignoriert.

Diese Matrizenmultiplikation findet in den Vertex Shadern statt. Standard-Vertex-Shader, wie sie bei den beiden Programmen tatsächlich eingesetzt werden, können im Anhang C eingesehen werden.

3.6 Kamera

OpenGL und Vulkan kennen beide das Prinzip einer Kamera nicht. Der Programmierer muss sich also selbst eine Lösung überlegen. Da das Prinzip einer Kamera für die Grafik-API's unbekannt ist, wird der 3D-Raum invers den Inputs des Benutzers bewegt. Wenn der Benutzer sich nach links bewegen möchte, bewegt sich nicht die Kamera nach links, sondern die Welt nach rechts. Wenn der Benutzer nach oben schauen will, rotiert nicht die Kamera nach oben, sondern die Welt nach unten. Und hier kommt die View-Matrix ins Spiel. Die Model-Matrix bestimmt Position, Ausrichtung und Grösse des Modells und die Projection-Matrix sorgt für perspektivische Verzerrungen. Noch macht die View-Matrix nichts von Belang. Analog zur Model-Matrix könnte die View-Matrix die Position und Ausrichtung nicht des Objektes, sondern des gesamten Raumes relativ zur vom Programmierer eingeführten Kamera bestimmen. Und das wird in der Praxis auch so gemacht. Der Positionsvektor der Kamera wird als $\vec{P}$ bezeichnet. Das View Space-Koordinatensystem hat seinen Ursprung in der Kamera und ist durch drei senkrecht stehende Vektoren, den Up-Vektor $\vec{U}$, den Right-Vektor $\vec{R}$ und den Direction-Vektor $\vec{D}$, definiert. Abbildung 2 veranschaulicht dies. Der Direction-Vektor $\vec{D}$ zeigt hierbei hinten aus der Kamera hinaus. Dies ist jedoch Definitionssache, da ein einziges Vorzeichen den Unterschied macht. Alle Vektoren bis auf $\vec{P}$ sind normiert. Die folgende Matrix wird gemeinhin als LookAt-Matrix bezeichnet. Dabei geht es darum, eine Transformationsmatrix zu finden, wenn die Positon und das

[5] Joey de Vries. *Learn OpenGL - Camera. (Englisch) [Lerne OpenGL - Kamera].* Aufgerufen: 24.10.2019. URL: `https://learnopengl.com/Getting-started/Camera`.

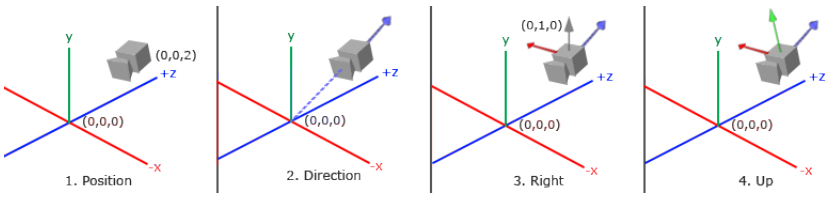


Abbildung 2: View Space-Koordinatensystem[5]

Target $\vec{T}$ der Kamera, also wo die Kamera hinblickt, gegeben ist.[6] Weiterhin gibt es noch den World Up-Vektor (bei Vulkan mit negativem Vorzeichen, da dort die y-Achse in die andere Richtung zeigt):

$$\vec{WU} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Die $\vec{U}$, $\vec{R}$ und $\vec{D}$ Vektoren lassen sich zu jedem Zeitpunkt relativ einfach berechnen:

$$\begin{aligned} \vec{D} &= \frac{\vec{P} - \vec{T}}{|\vec{P} - \vec{T}|} \\ \vec{R} &= \frac{\vec{WU} \times \vec{D}}{|\vec{WU} \times \vec{D}|} \\ \vec{U} &= \vec{D} \times \vec{R} \end{aligned}$$

$\vec{U}$ muss nicht normiert werden, da er aus dem Kreuzprodukt zweier bereits normierter Vektoren berechnet wird. Wenn ein Koordinatensystem wie das View Space-System durch 3 senkrechte Achsen definiert ist, lässt sich eine Matrix, welche Vektoren in dieses Koordinatensystem transformiert, erstellen.

In diesem Zusammenhang wird die Übersetzung von World Space- in View Space-Koordinaten vorgenommen, wobei exakt diese Transformation auf eine vorhergehende Verschiebung um die Position der Kamera angewendet wird. Diese Matrix wird dann LookAt-Matrix genannt[7]:

$$\begin{aligned} \text{LookAt} &= \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_x & R_y & R_z & -P_x \cdot R_x - P_y \cdot R_y - P_z \cdot R_z \\ U_x & U_y & U_z & -P_x \cdot U_x - P_y \cdot U_y - P_z \cdot U_z \\ D_x & D_y & D_z & -P_x \cdot D_x - P_y \cdot D_y - P_z \cdot D_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Diese LookAt-Matrix beinhaltet nun also die genaue Position ($\vec{P}$) und Ausrichtung ($\vec{R}$, $\vec{U}$ und $\vec{D}$) der Kamera. Genau die Bedingungen, die für die View-Matrix gelten. Somit ist die View-Matrix definiert und sorgt dafür, dass der Benutzer seine virtuelle Kamera bewegen und rotieren kann. In tatsächlichen Applikationen wird mit den sogenannten Eulerschen Winkeln gearbeitet. Diese bezeichnen die Rotation um jede der drei Achsen im Raum. In unserem Fall liegen die Bezeichnungen wie in Abbildung 3: Der Pitch θ für die x-Achse, Yaw ψ für die y-Achse und Roll ϕ für die z-Achse. Die Mausbewegungen des Users können als solche Winkel interpretiert werden. Durch einfache Trigonometrie

[6] Song Ho Ahn. *OpenGL Camera. (Englisch) [OpenGL Kamera].* Aufgerufen: 24.10.2019. URL: `http://www.songho.ca/opengl/gl_camera.html`.

[7] Song Ho Ahn, *OpenGL Camera. (Englisch) [OpenGL Kamera].*

[8] Joey de Vries, *Learn OpenGL - Camera. (Englisch) [Lerne OpenGL - Kamera].*

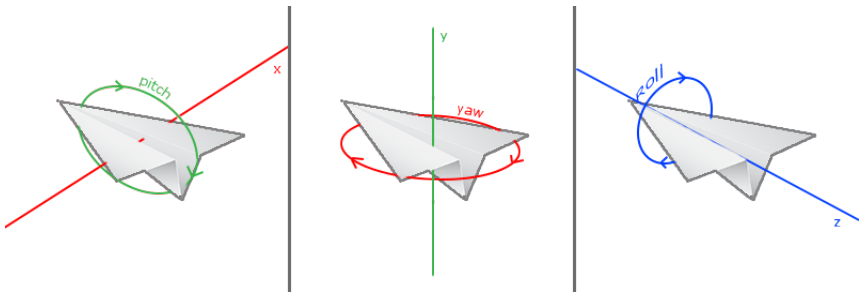


Abbildung 3: Eulersche Winkel anhand eines Flugzeugs[8]

kann dann der $\vec{D}$-Vektor berechnet werden:

$$\vec{D} = \begin{pmatrix} \cos\theta \cdot \cos\psi \\ \sin\theta \\ \cos\theta \cdot \sin\psi \end{pmatrix}$$

Da der Roll ϕ meistens nur bei Flugsimulatoren gebraucht wird, sieht dieser Vektor $\vec{D}$ in vielen Programmen so aus. Durch diese relativ einfachen Zusammenhänge funktioniert die vom Programmierer eingeführte Kamera in einem Videospiel oder einer 3D-Simulation.

3.7 Phong-Lighting

Ein wichtiger Bestandteil einer jeden Grafikapplikation, welche auf eine realitätsnahe Darstellung des Raumes abzielt, sind annähernd gute Berechnungen der Lichtverhältnisse. Ein Modell, welches sich durchgesetzt und bewährt hat, ist das sogenannte *Phong-Lighting* oder dessen Weiterentwicklung *Blinn-Phong-Lighting*. Dieses versucht, die geometrische Optik der Farbempfindung nachzuahmen. Dabei werden verschiedene Komponenten des Lichts unterschieden.[9]

3.7.1 Ambient Lighting

Es gibt das *Ambient Lighting*, welches als Umgebungslicht verstanden werden kann. Dies setzt sich in der echten Welt aus reflektiertem Licht von allen möglichen Oberflächen zusammen. Da dies rechentechnisch sehr schwer zu simulieren ist, wird normalerweise einfach eine bestimmte Menge an Umgebungsbeleuchtung angenommen.

3.7.2 Diffuse Lighting

Weiter existiert das *Diffuse Lighting*. Dies dient der Berechnung der direkten Beleuchtung eines Faces durch eine Lichtquelle mithilfe von ein paar vektorgeometrischen Eigenschaften. Über das Skalarprodukt des Vektors, welcher zur Lichtquelle zeigt, mit dem Normalenvektor des Faces kann ein Koeffizient für die Erhellung berechnet werden. Abbildung 4 zeigt dies schematisch.

3.7.3 Specular Lighting

Zu guter Letzt kommt das *Specular Lighting*. Es dient der Simulation von reflektionsbedingten optischen *Highlights*. Dies kann anschaulich an folgendem Beispiel erklärt werden: Man stelle sich ein Auto vor, welches durch die Metallkarosserie eine grosse optische Reflektivität hat. Die Sonne wird reflektiert, an gewissen Stellen stärker als an anderen, sodass man sogar geblendet wird. Um diese direkte Reflektion zu simulieren, wird auch hier über das Skalarprodukt des am Normalenvektor des Faces reflektierten Vektors mit dem Vektor, der zur Quelle zeigt, ein Koeffizient zur weiteren Aufhellung berechnet. Abbildung 5 stellt dies auch hier schematisch dar.

[9] Joey de Vries. *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung]*. Aufgerufen: 13.11.2019. URL: `https://learnopengl.com/Lighting/Basic-Lighting`.

[10] Joey de Vries, *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung]*.

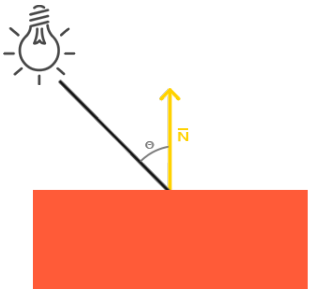


Abbildung 4: Diffuse Lighting im Querschnitt des Faces[10]

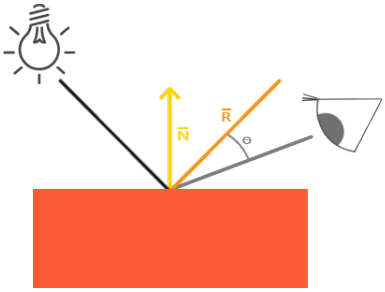


Abbildung 5: Specular Lighting im Querschnitt des Faces[11]

Die Kombination dieser drei Elemente zusammen ergibt eine relativ gute Simulation der Lichtverhältnisse, welche nicht zu viel Rechenaufwand abverlangt. In Anhang C können die jeweiligen Implementationen in den Fragment Shadern für beide Applikationen eingesehen werden. Früher, als Grafikkarten noch nicht so leistungsfähig waren wie heute, wurden diese Berechnungen nicht pro Fragment, sondern pro Vertex in den Vertex Shadern durchgeführt. Dies führte zu massiven Leistungsverbesserungen, aber auch zu einer Verfälschung des Resultats. Dies wurde dann Gouroud-Lighting genannt.[12]

3.8 Probleme und Schwierigkeiten

Diese vorhin erwähnten Rotationsmatrizen führen ein Problem, das unter dem Namen *Gimbal Lock* bekannt ist, ein. Dies entsteht vor allem dann, wenn anstatt um eine einzige Achse, um mehrere Achsen rotiert wird. Beispielsweise wird zuerst um die x-Achse, dann um die y-Achse rotiert, anstatt direkt um eine Achse, die dazwischen liegt. Die Rotationsmatrix um eine beliebige Achse schwächt diese Gimbal Lock ein wenig ab, kann sie jedoch nicht vollständig verhindern.[13] Bei der Gimbal Lock entsteht eine exakte Überlagerung zweier Achsen, wonach nur noch um zwei anstatt um drei Achsen rotiert werden kann. Um dies zu verhindern wurde im tatsächlichen Quellcode dann eine Begrenzung der Rotationswinkel der Kamera eingebaut, damit exakt dieses Problem nicht auftritt. Gimbal Lock tritt dann auf, wenn ein Rotationswinkel $\pm 90.0°$ erreicht. Das heisst, es muss sichergestellt sein, dass keiner der drei Rotationswinkel der Kamera jemals einen Wert von $\pm 90.0°$ annimmt. Da die Kamera in den Applikationen sich jedoch nur in zwei Richtungen steuern lässt (oben/unten, links/rechts) muss nur garantiert werden, dass der Pitch-Winkel niemals $\pm 90.0°$ oder grösser wird. Dies wäre ausserdem ebenfalls unpassend, da die Kamera dann kopfüber wäre und das Bild für den User somit nicht mehr viel Sinn ergäbe. Der Yaw muss nicht beschränkt werden, da sich die Kamera in diese Richtung ja unendlich oft drehen können soll.

Ausserdem tritt bei Berechnungen des Diffuse Lighting bei nicht-uniformen Skalierungen der Model-Matrix (also Skalierungen bei welchen die Länge des Skalierungsvektors $\neq 1$ ist) ein Fehler auf. Dann steht der Normalenvektor nicht mehr senkrecht zum Face und muss korrigiert werden. Des-

[11] Joey de Vries, *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung]*.
[12] Joey de Vries, *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung]*.
[13] Joey de Vries, *Learn OpenGL - Transformations. (Englisch) [Lerne OpenGL - Transformationen]*.

wegen wird eine separate Transformation auf die Normalvektoren angewandt, welche nur Rotationen und korrigierte Skalierungen überträgt. Dies funktioniert mit der sogenannten *Normal Matrix*. Sie entspricht der transponierten Inversen der Model-Matrix.[14] Dadurch, dass diese Matrix auf eine 3×3-Matrix gekürzt wird, fallen Translationen, welche ja in der vierten Spalte stehen, weg. Diese werden sowieso nicht gebraucht für die Normalenvektoren, da bei den Berechnungen mit den Normalenvektoren nur die Richtung wichtig ist.

[14] Joey de Vries, *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung]*.

4 C++ als Programmiersprache

Abbildung 6: Logo von ISO-C++[15]

4.1 Einführung

Die Programmiersprache C++ (gespr. *Cee-Plus-Plus*) wurde 1979 von Bjarne Stroustrup als Weiterentwicklung der älteren Programmiersprache C entworfen. Er wollte dabei C's Effizienz mit einigen weiteren Abstraktionen beibehalten. Deshalb begann er sein Projekt mit der Einführung von Klassen in C. Dies ist ein wesentlicher Unterschied der beiden Sprachen. C++ gilt als General-Purpose-Programmiersprache, das heisst, sie ist für alle möglichen Bereiche anwendbar. Heutzutage haben viele Betriebssysteme, Games, Browser, Treiber, Server-Backends und Machine-Learning-Tools (oft durch Python-C++-Bindings) und sonstige eher hardwareseitige Programme C++ als Kernsprache. C++ ist sehr effizient, da viele Optimierungen, welche in höherleveligen Programmiersprachen enthalten sind (z.B. Dynamisches Speicher-Management) bei C/C++ nicht automatisch vorgenommen werden.[16] Der Programmierer muss sich selbst darum kümmern. Dadurch steht ihm allerdings mehr Freiraum zur Verfügung und er kann die volle Rechenleistung seines Gerätes nutzen, weil er sich in C/C++ nicht mit oft performancelastigen Vereinfachungen anderer Sprachen abzumühen hat. Dies macht C++ im Jahr 2019 immer noch zur viertbeliebtesten Programmiersprache weltweit.[17] C++ ist ISO-standardisiert und etwa alle 3-4 Jahre kommen Updates für den C++-Standard und für die integrierte Standardbibliothek STL heraus. Abbildung 6 zeigt das ISO-C++-Logo. Die zwei Render-Engines wurden mit dem in 2017 erschienenen C++-17-Standard geschrieben und machen von einigen Neuerungen Gebrauch, die dieser Standard gebracht hat. Objekte der Standardbibliothek lassen sich daran erkennen, dass sie aus dem *std*-Namensraum kommen und somit *std::** als Präfix haben.

C und C++ stechen in der vielfältigen Welt der Programmier- und Scriptsprachen vor allem durch eines hervor: die sogenannten *Pointer* und *Referenzen*. Sie sind ein wesentlicher Grund dafür, dass C und C++ grundsätzlich sehr effizient sind. Worum handelt es sich bei einem Pointer? Ein Pointer kann als spezieller Datentyp angesehen werden, gilt jedoch streng genommen nicht als solcher. Er speichert nicht wie eine gewöhnliche Variable den Wert derjenigen, sondern die physische Speicheradresse dieses Wertes auf dem Arbeitsspeicher. Speicheradressen werden im Hexadezimalsystem durchnummeriert. Ein Pointer wird mit dem '*'-Zeichen deklariert. Eine Referenz wird mit dem Zeichen '&' angegeben und fordert die physische Speicheradresse einer normalen Variablen an. Mit dem folgenden Code-Snippet kann einem Pointer die Speicheradresse einer anderen Variablen zugewiesen werden:

[15] Standard C++ Foundation. Aufgerufen: 29.10.2019. URL: `https://isocpp.org/`.

[16] Parewa Labs Pvt. Ltd. *Learn C++ Programming. (Englisch) [Lerne C++ Programmieren].* Aufgerufen: 29.10.2019. URL: `https://www.programiz.com/cpp-programming`.

[17] TIOBE Software BV. *TIOBE Index for October 2019. (Englisch) [TIOBE Index für Oktober 2019].* Aufgerufen: 29.10.2019. URL: `https://www.tiobe.com/tiobe-index/`.

```
1 int meineVariable = 10;          // gewöhnliche Variable meineVariable
2                                  // wird deklariert und initialisiert
3 int* meinPointer;                // Pointer meinPointer wird deklariert
4 meinPointer = &meineVariable;    // Dem Pointer meinPointer wird die Speicher-
5                                  // adresse von meineVariable zugewiesen, welche
6                                  // durch das Referenzzeichen '&' angefordert wird
```

Auf den Wert der Variablen *meineVariable* kann nun ebenfalls über den Pointer zugegriffen werden. Wenn auf den Wert, der in der Speicheradresse geschrieben steht, zugegriffen werden will, muss dies ebenfalls mit dem Zeichen '*' gemacht werden:

```
1 std::cout << meinPointer << std::endl;      // Output: '0x79d7263b941c' (Speicher-
2                                             // adresse im Hexadezimalsystem)
3 std::cout << *meinPointer << std::endl;     // Output: '10'
```

Dabei ändert sich der Output von Zeile 1 jedes mal, denn der Computer wird immer eine andere Speicheradresse zuweisen und praktisch nie dieselbe. Aufgepasst, ein Pointer ist *nicht* dasselbe wie eine Referenz. Diese müssen als separate Dinge betrachtet werden, da sonst sehr schnell grosse Verwirrung aufkommen kann. Ein Pointer kann als Objekt angesehen werden, eine Referenz nicht. Dadurch, dass Speicheradressen abgespeichert werden können, erspart dies dem Programm einige mühsame Schritte z.B. beim Übergeben von Werten an Funktionen. Der Computer muss somit nicht den Wert der Variablen auslesen, diesen kopieren und dann erst übergeben, sondern kann die Variable direkt übergeben, indem er einfach die Adresse weitergibt. Dies hat noch einige weitere nützliche Anwendungsbereiche und führt vor allem bei grossen Datenstrukturen, wie zum Beispiel einem Array mit 10'000 Elementen, was keine Seltenheit ist, zu grossen Optimierungen. Der Grund wieso höhere Sprachen wie Python oder Java dieses Prinzip nicht kennen liegt am Abstraktionsgrad der Sprachen. Sie sind dazu da dem Programmierer die Entscheidung, ob eine Variable besser als Pointer definiert wird oder nicht, abzunehmen und zu automatisieren.

Von der sonstigen Syntax her ist C++ ein wenig wie Java. Es braucht geschweifte Klammern um zu strukturieren, es gibt mehr oder weniger dieselben Datentypen, Funktionen, Klassen, Kontrollsequenzen und sonstigen Statements. C++ muss wie Java kompiliert werden, damit es ausgeführt werden kann. C++ muss in die Maschinensprache Assembler übersetzt werden, während Java in seinen plattformneutralen Bytecode übersetzt werden muss. Assembler ist nicht garantiert plattformunabhängig, deswegen existieren Umsetzungen von sog. Compilern, welche die Übersetzung vornehmen, für alle gängigen und nicht so gängigen Betriebssysteme und Architekturen.[18]

Der Hauptgrund der Wahl für C++ als Programmiersprache für die beiden Render-Engines liegt darin, dass die Vulkan SDK, respektive die OpenGL Bindings durch GLAD in C/C++ geschrieben sind. Dies erlaubt einfaches Ansteuern der API's, ohne gross eine sprachspezifische Lösung wie z.B Lightweight Java Game Library (LWJGL) für Java installieren zu müssen. Ausserdem war bereits einiges an Erfahrung mit C++ vorhanden, was den Prozess der Entwicklung stark vereinfacht hat.

4.2 Compiler, Linker und Assembler

C/C++ kann nicht einfach so ausgeführt werden. Es sind verschiedene Zwischenschritte notwendig, um aus C/C++-Code eine ausführbare Datei zu generieren. Der Sourcecode muss zuerst so übersetzt werden, dass der Computer dies verstehen und ausführen kann. Dies übernimmt der sogenannte *Compiler*. Er übersetzt C/C++-Code in Assembler-Code, in Maschinensprache. Diese Maschinensprache sieht etwas komplizierter aus, kann aber von Menschen noch gelesen und verstanden werden. Früher wurde sogar in solchen Sprachen programmiert, da es noch keine weiterentwickelten Programmiersprachen gab. Ein Beispiel für eine einfache Addition in C/C++:

[18] tutorialspoint.com. *Assembly. (Englisch) [Assembler]*. Aufgerufen: 24.11.2019. URL: https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm.

```cpp
// add.cpp
int add(int a_, int b_) {
    int result = a_ + b_;
    return result;
}
```

Dies übersetzt in Assembler mit dem Befehl *g++ -S add.cpp* ergibt (gekürzt auf relevante Teile):

```
// add.s
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -20(%rbp)
movl    %esi, -24(%rbp)
movl    -20(%rbp), %edx
movl    -24(%rbp), %eax
addl    %edx, %eax
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
popq    %rbp
ret
```

Wenn man sich den Assembler-Code ein wenig genauer anschaut, dann sieht man, dass nur ein paar Register herumgeschoben werden und am Schluss in Zeile 8 der tatsächliche Additionsbefehl kommt. Diese Befehlsfolge kann ein Computer nun verstehen und ausführen. Unter Windows beinhalten ausführbare *.exe*-Dateien teilweise Assembler-Code. Würde man eine *.exe* mit den richtigen Encodings etc. in einem Texteditor öffnen, würde man zum Teil ähnliche Strukturen und Syntax wie in *add.s* sehen.

Bis der Compiler diese Übersetzung ausgibt, passiert vorher noch etwas anderes: der Präprozessor-Schritt. Der Präprozessor schaut sich den Quellcode an und sobald er *Preprocessor Directives*, also Präprozessor-Befehle, antrifft, führt er diese aus. Preprocessor Directives können zum Beispiel die folgenden Statements sein:

```cpp
#ifndef CORE_HPP
#define CORE_HPP
#include "Header.hpp"
#endif      // CORE_HPP
```

Mit Preprocessor Directives können auch Makros definiert werden, was in den Engines oft genug getan wurde um zum Beispiel Rückgabewerte auf erfolgreiche Ausführung einer Funktion zu überprüfen (sieht ähnlich aus wie das zweite aufgeführte Makro):

```cpp
#define ORIGIN vec3(0.0, 0.0, 0.0)
#define ASSERT(value_) if(value_ != true) throw std::runtime_error("Wrong Value!");
```

Der Präprozessor funktioniert als Text-Ersetzer. Er geht in diesem Fall überall hin wo *ORIGIN* und *ASSERT(value_)* steht und ersetzt diese stur durch Kopieren und Ersetzen mit deren jeweiligen Definitionen. Dabei passt er parametrisierte Makros logischerweise an. Wenn Definitionen von Datentypen, Strukturen und Funktionen aus einer anderen Datei gebraucht werden, zum Beispiel aus *Header.hpp*, wird das Preprocessor Directive *#include "Header.hpp"* benötigt. Der Präprozessor ersetzt in diesem Fall ganz einfach das *#include "Header.hpp"* durch den Inhalt der gesamten Datei *Header.hpp*.

Nach dem Präprozessor kommt der Compiler als Dolmetscher für Maschinensprache. Er produziert die sogenannten Object-Dateien, welche als Zwischenprodukte beim Kompilierungsprozess entstehen.

Als dritter Schritt kommt der sogenannte *Linker*. Er ist dafür zuständig, die in Machine Language übersetzten Object-Dateien und allfällige Bibliotheken korrekt zu dem Hauptprogramm zusammenzufügen und das schlussendliche Executable, die ausführbare Datei, zu generieren.

4.3 Sprachparadigmen

C++ kennt unter anderem folgende wichtigen Programmierkonzepte: es ist prozedural, objektorientiert und generisch.[19] Prozedurale Programmierung bezeichnet vereinfacht erklärt die Zerlegung von Algorithmen in Teilstücke (=Funktionen, Methoden oder sogar Unterprozesse, an welche sich Parameter übergeben lassen). Objektorientierung als eines der wichtigsten Konzepte der Computertechnologie meint die Abstrahierung von Programmcode in Klassen und Objekte, um diesen dynamischer und anpassungsfähiger zu machen. So können sehr viele von der Struktur gleiche aber von den exakten Eigenschaften unterschiedliche Objekte erstellt werden. Unter generischer Programmierung wird die Anpassung von Quellcode an verschiedene Datentypen verstanden. Die geschieht z.B. bei C++ mit sog. *Templates*, bei Java mit *Generic Types*.

C++ ist sehr dynamisch und hochleistungsfähig, zu einem grossen Teil dank diesem generischen Aufbau. Die integrierte Standardbibliothek STL zum Beispiel macht intern sehr oft von Templates Gebrauch. Sie sind ein sehr wichtiger Bestandteil von C++.

4.4 Was ist die Floating-Point-Ungenauigkeit?

Die Floating-Point-Ungenauigkeit bezeichnet generell Rechenfehler, welche bei mathematischen Operationen mit dem Datentypen des Floats, der sogenannten *Fliesskommazahl*, entstehen können. Fliesskommazahlen können Zahlenwerte nur bis zu einer gewissen Genauigkeit und somit nicht exakt speichern. Dadurch entstehen Rundungs- und sonstige Rechenfehler, welche sich durch weitere Berechnungen vervielfältigen können. In C++ haben Floats normalerweise eine Genauigkeit von 32-bit.[20]

4.5 Programmierstil in den Engines

In den zwei Engines wurde mit einem Programmierstil gearbeitet, der mit einigen Variationen sehr weit verbreitet ist. Es wurde die *Doxygen*-Syntax verwendet, um schnell eine HTML-Dokumentation generieren zu können. An den Anfang einer jeden Datei wurde ein Kommentar mit Autor, Version und Datum der Version eingefügt. Wenn der Code von anderen Programmierern 1 zu 1 übernommen wurde, sind dort Copyright-Angaben zu finden. Weiter wurde vor jede Funktions-Deklaration ein Kommentar angebracht, welcher kurz erklärt, was die Funktion mit welchen Parametern macht, was sie zurückgibt und wo allenfalls Fehler auftreten könnten. Funktions-Deklarationen wurden jeweils in *.hpp*-Dateien, die zugehörige Implementation der Funktion in der gleichnamigen *.cpp*-Datei vorgenommen. Ausserdem wurde zu Anfang und zu Ende einer Funktion jeweils eine Leerzeile eingefügt. Wenn eine Funktion mehr als 3 Parameter hatte, wurden diese jeweils auf eine neue Zeile geschrieben. Parameter von Funktionen wurden immer mit einem Unterstrich am Ende versehen, um diese von Variablen innerhalb des Scopes der Funktion unterscheiden zu können. Da im globalen Namensraum zu arbeiten immer bad practice ist, wurden die Applikationen jeweils in einem eigenen Namespace (= Namensraum) verstaut. Dabei enthielten sie mehrere Unter-Namespaces, der wichtigste davon war der *vk::core*/*ogl::core*-Namespace. Es wurde an einigen Stellen Gebrauch von mehreren C++-17-Features gemacht, unter anderem mit dem *std::optional*-Typen. Es wurden einige eher abstrakte Programmiertechniken wie das Operator-Overloading angewandt. Zum Beispiel wurden der Comparison- und der Invocation-Operator überladen. Durch Vererbung wurde vieles vereinfacht, unter anderem die Erstellung der verschiedenen Kameras und Buffer. Ausserdem wurde eine eigene Hash-Funktion für Vertex-Deduplikation implementiert. Diese erlaubte, die Vertices in einem sog. *Hash-Table* zu haben, was für die Deduplikation von grossem Performance-Vorteil war. Weiterhin wurden einige eher komplexere Themen des Multithreadings in C++ implementiert, zum Beispiel die *std::condition_variable*, welche das Aufwecken von schlafenden Threads erlaubt. API-spezifisch wurde teilweise auf dem Bitniveau gearbeitet, zum Beispiel mit den sogenannten *Flags*, welche später erklärt werden. In der eigens entwickelten Hash-Funktion wurde ebenfalls mit Bitshifting gearbeitet.

[19] Universität Heidelberg. *Introduction to the C++ Language. (Englisch) [Einführung in die C++ Programmiersprache]*. Aufgerufen: 29.10.2019. URL: `https://conan.iwr.uni-heidelberg.de/data/teaching/oopfsc_ss2017/structure.pdf`.

[20] learncpp.com/Alex. *4.8 — Floating point numbers. (Englisch) [4.8 — Fliesskommazahlen]*. Aufgerufen: 24.11.2019. URL: `https://www.learncpp.com/cpp-tutorial/floating-point-numbers/`.

5 OpenGL und Vulkan

5.1 OpenGL

Die Open Graphics Library (OpenGL) wurde 1992 als IRIS GL als eine erste Umgebung für 2D-, 3D- und Mobile-Grafikapplikationen von Silicon Graphics initiiert und im Juli 2006 von der Khronos Group, welche ein Industriekonsortium aus unter anderem Intel, Nvidia, AMD und Google stellt, übernommen.[21] Seit jeher ist OpenGL die am weitesten verbreitete Grafik-API.[22] Der Standard definiert einen Satz an Befehlen zur Computation und Darstellung von 3D-Szenerien in Echtzeit. OpenGL lässt proprietäre Erweiterungen von externen Organisationen zu. Die Implementation der Grafik-API findet durch Systembibliotheken oder Grafikkartentreiber statt. Durch diese Erweiterbarkeit wurden OpenGL-Implementationen für praktisch alle gängigen und nicht so gängigen Betriebssysteme und Umgebungen geschrieben, u.a. Microsoft Windows seit Windows 95, macOS, das X Window System (mit praktisch sämtlichen Linux-Distributionen), Solaris und vielen mehr. Ausserdem wurde mit der Open Graphics Library for Embedded Systems (OpenGL ES) die Verwendung von OpenGL auf mobilen Geräten (unter z.B. Apple iOS und Android) und Konsolen wie der Playstation oder der Xbox 360 möglich. Mit OpenGL ES wurde ebenfalls WebGL entwickelt, ein OpenGL-Kontext für Webbrowser, welcher durch OpenGL's Vielseitigkeit in praktisch jedem Browser implementiert ist.[23] Das Apple WebKit, welches die Grundlage für viele moderne Browser legt, nutzt zur Hardwarebeschleunigung OpenGL durch WebGL.[24]

5.1.1 Funktionsweise von OpenGL

OpenGL-Befehle folgen immer einer gewissen Syntax, um einheitlich und logisch strukturiert zu sein. Diese beinhaltet prinzipiell immer das Präfix *gl** oder *GL**. Beispiele dafür wären:

```
1 void glGenVertexArrays(GLsizei n, GLuint *arrays);
2 void glPolygonMode(GLenum face, GLenum mode);
3 void glClear(GLbitfield mask);
```

Analog funktioniert die Syntax der Windowing-Bibliothek GLFW. Diese macht von dem Präfix *glfw** Gebrauch. OpenGL definiert eigene Datentypen wie *GLenum* oder *GLuint*. Diese dienen prinzipiell dazu, dass die OpenGL-Implementation des Systems auf allen Plattformen mit denselben Datentypen arbeitet, welche standardisiert sind. Eine weitere Besonderheit ist, dass viele OpenGL-Befehle mit sogenannten *Flags* funktionieren. Flags dienen dazu, Optionen ein- oder auszuschalten, ohne dass für jede Option ein einzelner Parameter an die Funktion übergeben werden muss. Dieses Prinzip wird auch bei Vulkan verwendet und funktioniert wie folgt: Die Optionen können an- oder ausgeschalten werden und nehmen entweder den Wert *Ein* oder *Aus* an. Da Zahlenwerte im Binärsystem abgespeichert werden, kann jede Stelle im Binärsystem entweder den Wert 1 oder 0 haben. Ein Bit in der Computerwelt entspricht einer Stelle im Binärsystem. Dies kann durch bitweise logische Operationen ausgenutzt werden, um viele Optionen zu setzen. Insbesondere wird das bitweise ODER, welches den Operator | hat, benötigt. Wenn jetzt Option 1 den hexadezimalen Wert 0x0001, Option 2 den Wert 0x0002 und Option 3 den Wert 0x0004 zugewiesen bekommt, dann entsprechen diese hexadezimalen Werte den binären Werten 0001, 0010 und 0100. Dabei ist die 1 jeweils immer an genau einer Stelle. Diese Werte werden dann die Flag-Bits genannt. Wenn jetzt zum Beispiel Option 1 und Option 3, nicht jedoch Option 2 aktiviert werden wollen, dann werden die zwei Flags durch ein bitweises ODER verknüpft und es kommt 0001|0100 = 0101 heraus. Optionen können durch geschickte Werte also genau einem Bit einer Zahl zugewiesen werden. OpenGL kann dann selbst überprüfen, welches Bit auf

[21] spo-comm GmbH. *Was ist OpenGL?* Aufgerufen: 04.11.2019. URL: `https://www.spo-comm.de/de/blognews/detail/article/News/detail/was-ist-opengl/`.

[22] The Khronos Group Inc. *OpenGL Overview. (Englisch) [OpenGL Überblick].* Aufgerufen: 04.11.2019. URL: `https://www.opengl.org/about/`.

[23] OpenGL Wiki. *Related toolkits and APIs. (Englisch) [Verwandte Toolkits und API's].* Aufgerufen: 10.11.2019. URL: `https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs`.

[24] Apple Inc. *WPE - WebKit Port For Embedded Systems. (Englisch) [WPE - WebKit Port für eingebettete Systeme].* Aufgerufen: 10.11.2019. URL: `https://webkit.org/wpe/`.

Ein und welches auf *Aus* gestellt ist. OpenGL (und auch Vulkan) weiss dann genau, dass Optionen 1 und 3 aktiviert werden sollen, und Option 2 nicht. In echten Programmen sieht man diese Idee auch in anderen Zusammenhängen als nur der Grafikprogrammierung. Die Benutzung von Flags gilt als allgemeines Konzept in der Programmierung. Es gibt verschiedene Wege, dieses System umzusetzen. In Anhang B sind Beispiele für Flags und Flag-Bits aus OpenGL und Vulkan aufgelistet, da dies ein fundamentales Prinzip ist, von welchem beide Grafik-API's Gebrauch machen.

OpenGL funktioniert über die sogenannte *Pipeline*. Die Pipeline besteht aus mehreren *Stages*, wovon die wichtigsten die *Shader Stages* sind. Diese umfassen den *Vertex Shader*, *Fragment Shader* und *Geometry Shader*, welche allesamt durch den Entwickler programmierbar sind.[25] Shader sind kleine Programme, welche auf der GPU laufen. Sie sind in der OpenGL Shading Language (GLSL), einer von der Syntax her C-ähnlichen Sprache, geschrieben. Shader umfassen immer eine *main*-Methode, welche für jeden Input ausgeführt wird. Das heisst, ein Vertex Shader führt seine *main*-Methode für jeden Vertex und ein Fragment Shader seine *main*-Methode für jedes Fragment einzeln aus (für Beispiele von Shader-Programmen in GLSL siehe Anhang C). Es können durch sogenannte *Uniform Buffers* oder grössere *Storage Buffers* kleinere bis mittlere Datenmengen an die Shader geschickt werden. Der Vertex Shader führt Berechnungen also pro Vertex aus. Diese kommen dann durch die *Shape Assembly Stage*, wo sie zu primitiven Dreiecken verbunden werden. Ein optionaler Geometry Shader verfeinert allenfalls die Geometrie der Dreiecke oder führt sonstige Berechnungen auf Basis der generellen Geometrie der Vertices aus. Darauf folgt die *Rasterization Stage*. Diese rastert den Output des Geometry Shaders in Fragments. Diese kommen dann durch den Fragment Shader, wo die meisten Licht- und Schattenberechnungen durchgeführt werden. Zu guter Letzt folgen diverse Tests und sogenannte *Blending*-Operationen. Blending bezeichnet die Berechnung der Farbe wenn Transparenz vorhanden ist und dahinterliegende Objekte einen Einfluss auf die Farbe des spezifischen Fragments haben. Die angesprochenen Tests umfassen unter anderem Depth-, Stencil- und Scissor-Tests. Der Depth-Test überprüft, ob ein Fragment hinter oder vor einem anderen zu liegen kommt und zeigt dann dementsprechend das richtige an, respektive führt die Blending-Operationen in der richtigen Reihenfolge aus. Stencil-Tests dienen zum Beispiel dem Umrahmen von 3D-Objekten. Der Scissor-Test schneidet den Ausschnitt so zu wie der Programmierer es haben will. Abbildung 7 veranschaulicht die Pipeline. Dabei sind die blauen, die programmierbaren und die grauen die fixen Stages. Die Vertices werden durch einen sogenannten *Draw Call* durch die Pipeline gepumpt. Sobald also im Code ein Aufruf zur OpenGL-Funktion *glDrawArrays()* steht, wird ein Draw Call auf der Grafikkarte mit vordefinierter Pipeline ausgeführt.

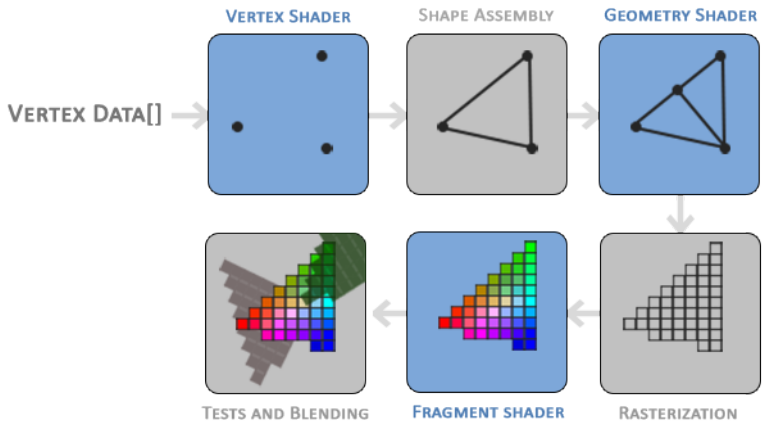


Abbildung 7: OpenGL-Pipeline[26]

[25] OpenGL Wiki. *Rendering Pipeline Overview. (Englisch) [Rendering Pipeline Übersicht].* Aufgerufen: 11.11.2019. URL: `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`.

[26] Joey de Vries. *Learn OpenGL - Hello Triangle. (Englisch) (Lerne OpenGL - Hallo Dreieck).* Aufgerufen: 12.11.2019. URL: `https://learnopengl.com/Getting-started/Hello-Triangle`.

5.2 Vulkan

Vulkan wird als Nachfolger von OpenGL angesehen. Vulkan wird ebenfalls von der Khronos Group entwickelt und vertrieben. Es wurde zuerst unter den Namen *Next Generation OpenGL* und *glNext* vorgestellt aber im Februar 2016 als Vulkan 1.0 veröffentlicht. Es ist abgeleitet von einer Grafik-API von AMD namens AMD Mantle. Es bietet durch seine Low-Level-Eigenschaft sehr viel Optimierungspotenzial für den Programmierer. Dies bringt aber auch Nachteile mit sich, bezüglich der Fehleranfälligkeit und des programmiertechnischen Aufwandes.[27] Vulkan wird seiner Stellung als Nachfolger von OpenGL jedoch durch diese Optimierbarkeit sehr gerecht. Die heutigen Spiele und Grafikapplikationen werden immer realistischer und rechentechnisch viel anspruchsvoller. Durch das immense Parallelisierungspotenzial der Computation auf Grafikkarten werden solche API's immer wichtiger. Der Bedarf für High Performance Grafik- und Compute-API's ist da. Vulkan hat im Gegensatz zu OpenGL die Möglichkeit, auf mehreren CPU-Threads zu laufen. Dadurch, dass moderne Central Processing Unit's (CPU's) ebenfalls immer mehr Kerne bekommen, sind diese Multithreading-Kapazitäten von enormer Bedeutung und stellen den Hauptvorteil gegenüber OpenGL. Vulkan ist eine sehr tief liegende API. Das bedeutet, dass der Programmierer sehr viel mehr Arbeit zu erledigen hat als bei OpenGL, welches höher abstrahiert und vereinfacht ist. Vulkan ist ebenfalls plattformübergreifend, jedoch noch nicht so weit verbreitet wie OpenGL.[28]

5.2.1 Funktionsweise von Vulkan

Vulkan funktioniert im Prinzip gleich wie OpenGL. Auch Vulkan macht Gebrauch von Präfixen: Jeder Vulkan-Befehl beginnt mit *vk**. Auch bei Vulkan gibt es die Pipeline. Diese sieht durch die Low-Level-Eigenschaft der API allerdings wesentlich komplizierter aus, wie in Abbildung 8 dargestellt ist. Es gibt viel mehr Möglichkeiten, Shader zu definieren. Unter anderem gibt es bei Vulkan Tessellation und Compute Shader. Mittlerweile kennt OpenGL das Prinzip des Tesselation Shaders seit Version 4.0 auch.[29] Die tiefliegende Verbundenheit mit der Computerarchitektur zeigt sich bei Vulkan auch darin, dass der Programmierer viele interne Objekte selber erstellen und zerstören muss. Während in OpenGL alles intern gehandhabt wird, muss man sich bei Vulkan um eine Instance, ein Physical und Logical Device, ein Surface und einige weitere spezialisierte Dinge bemühen, bevor überhaupt etwas sichtbar ist. Weiterhin funktioniert Vulkan durch sogenannte *Command Buffer*. Dies sind Ansammlungen von Befehlen, welche zuerst aufgenommen und in den Command Buffer geschrieben werden müssen und erst auf ein separates Kommando auf einer *Queue* auf der Grafikkarte ausgeführt werden. Eine Queue muss man sich wie einen Prozessorkern bei der CPU vorstellen. Es können Ressourcen zwischen Queues geteilt und Informationen ausgetauscht werden, damit das Programm schneller läuft. Im Prinzip funktioniert das Ganze wie Multithreading. Jeder Zugriff auf ein Objekt muss synchronisiert werden, speziell wenn mehrere Queues auf dasselbe Objekt zugreifen. Das heisst, Vulkan führt seine Command Buffer asynchron auf der GPU aus. Dies bringt einige Vorteile mit, zieht aber auch Unannehmlichkeiten nach sich. Hauptvorteil davon ist, dass der Vulkan Treiber seine Ressourcen im Voraus allozieren und reservieren kann. Der grosse Nachteil davon ist allerdings, dass der Programmierer Vulkan-Objekte zwischen den Queues synchronisieren muss. Dies macht die Sache wesentlich komplizierter. Hinzu kommt noch eine weitere Schwierigkeit: Da bei Vulkan viel mit Buffern, Pointern und internen Objekten gearbeitet wird, kann bei jedem einzelnen Schritt etwas falsch laufen. Zum Beispiel gibt man aus Versehen einen Nullpointer oder eine fragwürdige Eingabe ein. Damit wird dann z.B. ein internes Objekt erstellt, welches durch die Eingabe ungültig ist. Vulkan direkt hat aber keine Möglichkeit Informationen zur Konsole auszugeben und dem Programmierer dadurch das Debuggen zu vereinfachen. In einem solchen Fall würde das Programm einfach abstürzen, ohne Debug-Informationen auszugeben. Das heisst, es kann zu jedem Zeitpunkt an *irgendetwas* scheitern und der Programmierer hat nicht die geringste Ahnung, woran. Da sich so nur sehr schwer fähige Programme erstellen lassen, hat die Vulkan-Community die sogenannten *Validation Layers* entwickelt.

[27] Alexander Overvoorde. *Vulkan Tutorial - Introduction. (Englisch) [Vulkan Tutorial - Einführung].* Aufgerufen: 12.11.2019. URL: `https://vulkan-tutorial.com/`.

[28] The Khronos Group Inc. *Vulkan Overview. (Englisch) [Vulkan Übersicht].* Aufgerufen: 12.11.2019. URL: `https://www.khronos.org/vulkan/`.

[29] OpenGL Wiki. *Tessellation.* Aufgerufen: 12.11.2019. URL: `https://khronos.org/opengl/wiki/Tessellation`.

Diese funktionieren als Kontrollobjekte, welche sämtliche Eingaben des Programmieres auf Validität überprüfen. Gibt der Programmierer etwas ungültiges ein, so schlagen diese Alarm und melden zurück, was mit welchem Objekt nicht stimmt. Da der Vulkan-Code über seine Command-Buffer asynchron auf der Grafikkarte ausgeführt wird, ist dies allerdings nicht allzu hilfreich. Aber es ist ein Anfang, welcher wenigstens einen Hinweis gibt, in welcher Richtung das Problem gelagert sein könnte. Vulkan

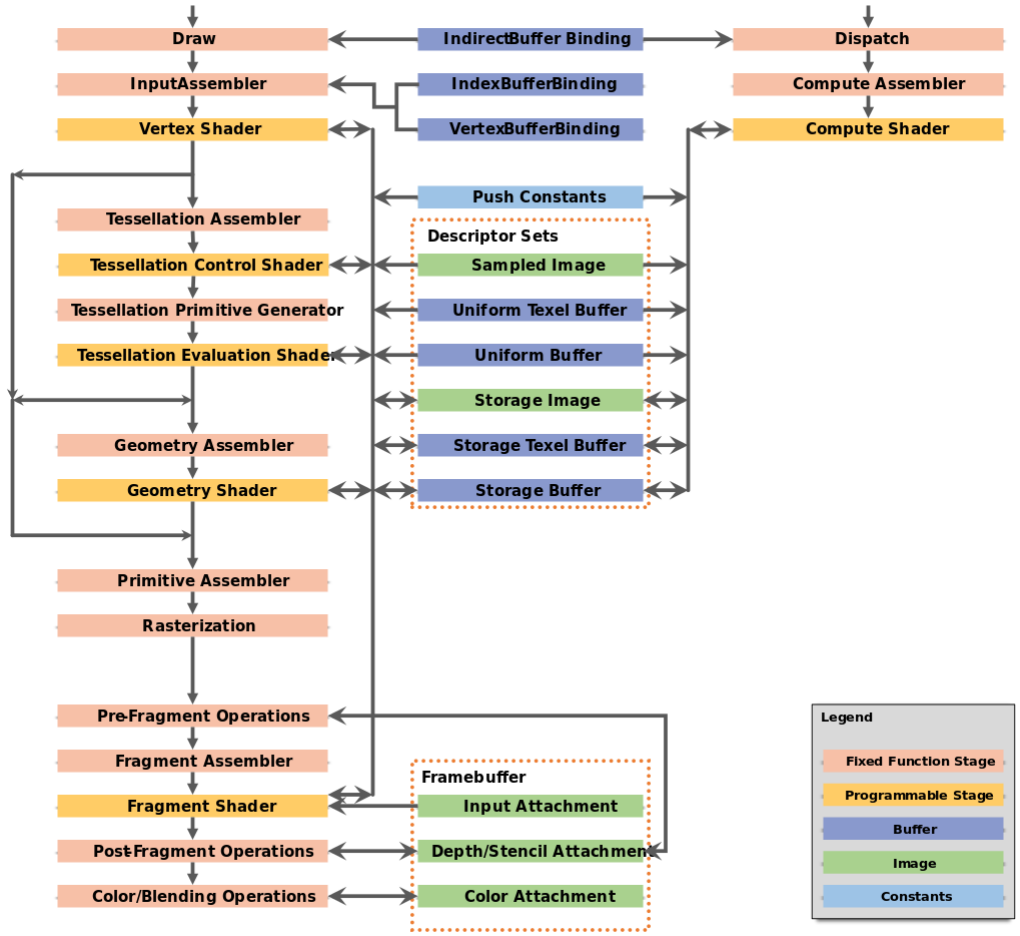


Abbildung 8: Vulkan-Pipeline[30]

hat eine weitere Eigenart, welche OpenGL nicht hat: Es arbeitet intern sehr viel mit Structures von C/C++. Während bei OpenGL z.B. Flags direkt als Funktionsparameter übergeben werden, werden bei Vulkan oft Structs leer initialisiert und dann mit Informationen gefüllt. Für viele Objekte, welche zwangsweise erstellt werden müssen, existiert ein sogenanntes *Create Info Structure*. Dieses beinhaltet alle nötigen Informationen zur korrekten internen Erstellung des Objektes. Nachdem das Create Info Struct korrekt ausgefüllt ist, folgt der Befehl zur tatsächlichen Erstellung des Objektes. Diese Befehle haben immer denselben Aufbau: Der Befehl lautet *vkCreate** + das Objekt, das zu erstellen ist. Dann folgen die Parameter, wovon der erste immer das Vulkan Logical Device ($\widehat{=}$ Grafikchip) ist. Des Weiteren folgt immer der Vulkan Memory Allocator und das sogenannte *Handle* des Objektes, welches man erstellen möchte, als Referenz, sodass Vulkan in dessen Speicheradresse schreiben kann. Ein Handle ist ein Datentyp von Vulkan, welcher der API genau sagt, welches Objekt intern damit gemeint ist. Auf das Objekt wird immer über das Handle zugegriffen. Der Befehl zur Erstellung gibt immer direkt einen *VkResult*-Statuscode zurück, welcher angibt, ob bei der Erstellung des Objektes irgendetwas schief gelaufen ist. In Anhang D kann so eine Erstellung eines Vulkan-Objektes mit Hilfe des Create Info Structs eingesehen werden. Vulkan-Structs haben immer den Parameter

[30]The Khronos Vulkan Working Group. *Vulkan Specification - 9. Pipelines. (Englisch) [Vulkan Spezifikation - 9. Pipelines]*. Aufgerufen: 12.11.2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#pipelines`.

sType, welcher für den sogenannten *VK_STRUCTURE_TYPE* steht. Dieser muss zwingend immer korrekt gesetzt werden und existiert aus dem folgenden Grund: Structs in C/C++ sind effektiv nur direkt hintereinanderliegende Variablen auf dem Speicher. Vulkan intern sieht nur das lineare Layout des Speichers. Das heisst, für Vulkan ist nur Speicheradresse nach Speicheradresse sichtbar. Es weiss nicht, wo ein Struct beginnt und wo es aufhört. Deswegen muss Vulkan dies mitgeteilt werden. Über den VK_STRUCTURE_TYPE weiss Vulkan um was für ein Objekt/Struct es sich handelt. Ausserdem können Erweiterungen so einfacher in Vulkan integriert werden, da diese dann keine Vulkan-Funktion komplett neu definieren müssen, sondern einfach ein Alias des Structures mit einem andere Structure Type angeben können. Eine weitere Besonderheit von Vulkan ist, dass die in GLSL geschriebenen Shader zuerst in *SPIR-V*-Format kompiliert werden müssen, bevor sie gebraucht werden können. Die Kompilierung passiert bei OpenGL zur Runtime, bei Vulkan zur separaten Compile-Time.

5.3 Direkter Vergleich

Da Vulkan so gesehen der direkte Nachfolger von OpenGL ist, gibt es hier einige Dinge, welche verglichen werden können. Der Vergleich wird tabellarisch gemacht. Tabelle 1 stellt einen Vergleich bezüglich gewissen Kriterien zwischen den beiden API's her.

Tabelle 1: Vergleich von Vulkan und OpenGL

Implementation		
	OpenGL	Vulkan
Aufwand	niedrig/mittel	sehr hoch
Komplexität	mittel	sehr hoch
Abstraktionsgrad	tief	hoch
Fehleranfälligkeit	tief	hoch
Debugmöglichkeiten	internes Error-Callback	eigenes Debug-Callback durch externe Validation Layers
Optimierungsmöglichkeiten	eher wenige	sehr viele
Parallelisierungsgrad	nur teilweise auf GPU	sehr hoch
Multithreading	sehr eingeschränkt möglich mit eher kleinem Ertrag	komplex, aber gut umsetzbar mit grossen Verbesserungen
Lerneffekt über 3D-Rendering	mittel	sehr genau und vertieft
Programmiertechnisches Vorwissen	nicht fordernd	eher fordernd (moderne Standards)
Plattformen	weitgehend unterstützt	nicht überall unterstützt, da sehr neu

Grafische Features		
	OpenGL	Vulkan
MultiSampling AntiAliasing	keine Angaben	bis zu 64-fach
Depth-Testing	ja	ja
Stencil-Testing	ja	ja
Blending	ja	ja
Face-Culling	ja	ja
Eigener Framebuffer	optional	zwingend
Texturen und Sampler	1D, 2D und 3D	1D, 2D und 3D
Instancing	ja	ja
Normal und Parallax-Mapping	ja	ja
Schatten	ja	ja
HDR	ja	ja
Ambient Occlusion	ja	ja
Gamma Correction	ja	ja

Leistung		
	OpenGL	Vulkan
Frametimes/FPS	top	top
Auslegung	Graphics	Graphics & Compute
Flüssige Darstellung	nicht immer	eher oft
Probleme	Teilweise Screen-Tearing	Je nach *VK_PRESENT_MODE*
	Stockend unter Linux	kann Screen Tearing auftreten
	V-Sync nicht deaktivierbar unter Linux	V-Sync nicht deaktivierbar unter Linux

Wie aus der Tabelle hervorgeht, haben die Engines in etwa dieselben grafischen Features. Lediglich bei der Implementation gibt es grosse Unterschiede und Vulkan steht im ersten Moment nicht allzu gut da. In erster Linie liegt dies daran, dass OpenGL kontinuierlich aktualisiert und optimiert wurde. Deswegen konnte OpenGL seine Einfachheit gegenüber Vulkan behalten und dabei dieselben Leistungen erzielen. Wie schon zuvor erwähnt ist Vulkan viel hardware-näher als OpenGL. Dies lässt sich auch aus der Tabelle herauslesen. Weiter ist klar erkennbar, dass es von den grafischen Features keine erkennbaren Unterschiede gibt. Bei der Leistung sticht Vulkan klar heraus, da es eigentlich immer flüssig läuft. OpenGL kommt bei grossen Buffern und vielen Objekten schnell mal ins Schwitzen. Durch Parallelisierung ist dies Vulkans Hauptvorteil.

Dass V-Sync unter Linux nicht deaktivierbar ist, liegt bei den Implementationen entweder an der Windowing-Library GLFW oder dem Windowing-System des Betriebssystemes und somit nicht direkt an den API's.

Der *VK_PRESENT_MODE* legt fest, wie ein Bild angezeigt wird, nachdem es gerendert wurde. Als Beispiel gibt es den *VK_PRESENT_MODE_FIFO_KHR*. FIFO steht dabei für *first in, first out*. Vulkan wartet bis der Bildschirm zum Darstellen des neuen Frames bereit ist und zeigt dieses erst dann. Dabei kann kein Screen Tearing auftreten, da dieses mit unterschiedlichen Aktualisierungs- und Darstellungsfrequenzen des Framebuffers und des Bildschirms zu tun hat. Der *Framebuffer* ist eine Speicheransammlung im High-Performance-Arbeitsspeicher auf der Grafikkarte, in welche das aktuell gerenderte Bild geschrieben wird, bis es angezeigt werden kann. Tearing tritt dann auf, wenn der Computer beginnt, das gerenderte Bild aus dem Framebuffer zu lesen und zum Bildschirm zur Darstellung zu senden, aber mitten in diesem Prozess des Herauslesens bereits das neue Frame hineingeschrieben wird. Dadurch zeichnet sich eine ganz klare Linie zwischen dem alten und dem neuen Frame ab; das Bild *reisst* (engl. *to tear*, 'reissen'). Dies wird als Störartefakt bezeichnet und kann durch eine Synchronisation der Frequenzen behoben werden. Dies ist im Prinzip das, was V-Sync macht. Bei den Applikationen wurde für die Tests *VK_PRESENT_MODE_MAILBOX_KHR* benutzt, welcher mit Tearing zu kämpfen hat und kein V-Sync besitzt. Jedoch wurde die Aktualisierungsfrequenz des Framebuffers unter Linux automatisch trotzdem mit der des Bildschirms synchronisiert. Dies konnte nicht ausgeschalten werden. Ursachen dafür wurden nicht weiter abgeklärt, da es vermutlich nicht mehr direkt mit den API's zu tun hat.

6 Progammierung und Implementation

6.1 Entwicklungsumgebung

Als primäre Entwicklungsumgebung wurde Microsoft Visual Studio Community 2017 (und später Microsoft Visual Studio Community 2019), welches in der Community-Edition Free-Ware ist, unter Windows 10 verwendet. Microsoft Visual Studio Community bringt eine vollkommene C++-IDE, d.h. Compiler, Debugger und Texteditor in einem. In dieser Umgebung wurde ein Grossteil der Vulkan-Applikation, welche gegenwärtig rund 8'500 Zeilen Quellcode umfasst, geschrieben. Später in der Entwicklung wurde komplett umgestellt auf Seiten der Entwicklungsumgebung: Da die beiden Applikationen grundsätzlich plattformübergreifend programmiert wurden und nur wenige Tweaks nötig waren, um die Software unter Linux kompilieren zu können, wurde dies dann auch gemacht. Es wurde unter Debian (genauer: Ubuntu 18.04/Linux Mint 19.2) mit Microsoft's allgegenwärtigem Visual Studio Code weitergearbeitet. Microsoft Visual Studio Code ist keine Integrated Development Environment (IDE) mehr, sondern nur noch ein reiner Texteditor. Dies bedeutet für den Entwickler, dass er auf andere Compiler und Debugger zurückgreifen muss. Debian bietet die meist bereits vorinstallierte GNU Compiler Collection (gcc) mit g++, dem Compiler für C++. Als Debugger wurde der ebenfalls meist schon vorinstallierte GNU Debugger (gdb) verwendet.

6.1.1 Verwendete Tools und Hilfsmittel

Für die Entwicklung der beiden Applikationen wurden mehrere Tools verwendet, die dem Entwickler die Arbeit sehr vereinfachen. Als Version Control System (VCS) wurde Git in Kombination mit der Source-Code-Sharing-Plattform GitHub verwendet. Dies erlaubt einfaches Zurückspringen zu alten Versionen und Tracking jeder einzelner Änderung. Es bringt gute Möglichkeiten der Zusammenarbeit, wenn mehrere Entwickler am selben Projekt arbeiten würden, was bei diesen zwei Projekten (bis auf eine kleine Ausnahme) nicht der Fall war. Ausserdem wurde GitKraken als Git-GUI-Client benutzt, da Git eigentlich hauptsächlich für die Konsole/das Terminal ausgelegt ist. Zusammen mit GitHub wurde die Continuous Integration (CI)-Lösung Travis CI eingesetzt. CI-Lösungen sind dafür zuständig, bei einer Änderung/Neuerung im Quellcode des Projektes, immer dieselben Tests durchzuführen, um sicherzustellen, dass die Software nach dieser Änderung weiterhin so funktioniert, wie sie sollte.

Vor dem Wechsel zu Linux wurde nur die Microsoft Visual Studio-IDE verwendet, da diese alles notwendige an einem Ort bot. Unter Linux wurden dann, wie schon gesagt, Visual Studio Code und g++ mit gdb benutzt. Ausserdem wurde mit Processing 3 eine kleine Doppelpendel-Simulation geschrieben, die dazu diente, die Bewegungsgleichungen und Formeln für das Pendel zu testen.

6.1.2 Verwendete Bibliotheken

In den Applikationen wurde nicht alles von Hand geschrieben. Für gewisse Teilbereiche wurden Bibliotheken benötigt, da die Zeit und teilweise das Knowhow fehlten, gewisse Dinge selber zu programmieren. Folgende Libraries wurden verwendet:

- Open Asset Import Library (ASSIMP)
- Graphics Library Framework (GLFW)
- tinyobjloader
- stb_image.h
- Simple DirectMedia Layer 2.0 (SDL2)
- OpenGL Mathematics (GLM)
- GLAD

ASSIMP bietet sehr schnelles und zuverlässiges Laden von 3D-Modellen vieler verschiedener Formate. GLFW als Windowing-Library bietet alles Notwendige für ein Applikationsfenster und OpenGL- resp. Vulkan-Kontexte. Tinyobjloader ist eine zweite Model-Loading-Bibliothek, welche zuerst als einfachere Variante implementiert wurde, später jedoch mit einigen 3D-Modellen nicht mehr funktionierte. Die Header-Only-Library std_image.h dient dem Laden von Texturen aus den Modell-Dateien und sonstigen Bilddateien wie dem Icon und dem Ladebildschirm der Applikation. SDL2 ist eine weitere Windowing-Library, welche OpenGL/Vulkan-Kontexte erzeugen kann, aber primär für den Ladebildschirm gebraucht wurde. GLM ist eine Bibliothek für lineare Algebra und Vektorgeometrie. Dies hätte auch ohne Probleme selber umgesetzt werden können, aber aufgrund der Zeitknappheit wurde darauf verzichtet. GLAD wurde nur für die OpenGL-Applikation verwendet. Es dient dazu die Function-Pointer für die OpenGL-Funktionen zu setzen, da diese sonst nicht bekannt wären. Alle Libraries sind OpenSource und plattformübergreifend. Teilweise mussten die Libraries selber kompiliert werden, was insofern ein Problem darstellte als dass diese meistens nur für 32-bit, die Applikationen jedoch auf 64-bit-Architekturen ausgelegt waren. Nach einiger Zeit wurde allerdings auch diese Hürde überwunden und beide Applikationen funktionierten seitdem auf 64-bit-Architekturen.

6.2 OGL

Die Entwicklung der OpenGL-Applikation verlief problemlos. Da zuerst die Vulkan-Engine geschrieben wurde, konnten bei OpenGL grosse Teile des Quellcodes 1 zu 1 kopiert werden. Beispiele dafür wären das Kamerasystem, das Model-Loading-System, der Startbildschirm, die eigens erstellte Logginglösung und die grundsätzliche Struktur der Applikationen. Sie sind in sich sehr ähnlich aufgebaut; der grösste Unterschied zwischen der Vulkan- und der OpenGL-Implementation besteht darin, dass bei Vulkan die y-Achse ein negatives Vorzeichen hat. Da sehr modular und im Hinblick auf Kompatibilität zwischen beiden Engines gearbeitet wurde war es ein leichtes, grosse Teile einfach zu übernehmen. Dadurch, dass OpenGL an sich nicht sehr kompliziert ist und sehr schnell Resultate, welche sehr vorzeigbar sind, erarbeitet werden können, dauerte die Entwicklung mit OpenGL nur ungefähr vier Arbeitstage und lieferte hierbei dieselben grafischen Leistungen wie die Vulkan-Applikation. Die nennenswertesten davon umfassen MipMapping, MultiSampling AntiAliasing, Color-Blending und Depth-Testing.

Grundsätzlich unterstützt OpenGL kein Multithreading. Das heisst, es lässt nativ keine Parallelisierung durch den Programmierer zu, sondern möchte alles API-intern handhaben und selbst optimieren. Dies bedeutet grundsätzlich längere Lade- und Startzeiten der Applikationen mit OpenGL, da diese nur mit einem Thread gleichzeitig Assets laden können. Unter Assets versteht man generell Modelle, Texturen, Audiodateien und dergleichen.

6.3 VK

Das Development der Vulkan-Engine verlief nicht ganz so problemlos, vor allem da diese zuerst geschrieben wurde. Das bedeutete, die ganzen Systeme, welche bei OpenGL einfach übernommen wurden, mussten hier zuerst erarbeitet und entwickelt werden. Dass hierbei viele kleine aber feine Bugs und Fehler entstehen, war erwartbar. In welcher Grössenordnung dies geschieht, war nicht vorauszusehen und es wurde definitiv nicht in dem schlussendlichen Ausmass erwartet. Dies führte zu grossen Verzögerungen und die Engines mussten auf einige eigentlich geplante Features verzichten. Da der Programmierer bei Vulkan sehr low-level arbeitet, muss er viel mehr selber machen als bei höheren API's wie OpenGL. Dies führt einerseits zu einer grösseren Freiheit für Leistungsoptimierungen, andererseits zu einem sehr viel grösseren Aufwand für dasselbe Ergebnis. Man schaue sich die rohen Zahlen an: Bei OpenGL betrug die Anzahl Zeilen Quellcode in der finalen Form ca. 3'500, bei Vulkan etwa 8'500. Dass bei rund 5'000 Zeilen mehr Code viel mehr Fehler entstehen können, ist zu erwarten. Insbesondere gab es ein Problem, welches über mehrere Wochen bestand: Es wurden Texturen nicht richtig dargestellt. Es wurde alles mehrmals überprüft: Texturkoordinaten, Texturdateien, die Shader, die Mappings auf das Modell... Nichts schien zu helfen, nicht eimal das Update auf die Vulkan Spec-Version 1.1.114.0. Dann wurden die Lighting-Berechnungen eingebaut und plötzlich waren die Artefakte einfach weg, wofür bis heute keine rationale Erklärung gefunden wurde. Ein weiteres Problem an welchem lange gearbeitet wurde stand im Zusammenhang mit der Asynchronität von Vulkans

Ausführung auf der Grafikkarte. Da nicht bekannt war, welches Modell gerade durch die Pipeline und die Shader gepumpt wurde, war zwangsweise ebenfalls nicht bekannt, wann welche Model-Matrix zu den Shadern gesendet werden musste. Die einfachste Lösung boten die sogenannten *Push Constants*. Push Constants dienen dazu, an einem bestimmten Punkt der Ausführung eines Command Buffers einen Wert zu den Shadern hochzuladen. Somit kann genau dann, wenn ein bestimmtes Model die Shader Stages durchläuft, ein Wert hochgeladen werden. An sich sind Push Constants sehr schnelle und performancegünstige Lösungen, um kleinere Datenmengen hochzuladen. Push Constants müssen laut der Vulkan API Spezifikation immer mindestens 128 Byte unterstützen.[31] Dies passt perfekt für eine 4×4-Matrix aus 32-bit Floats wie die Model-Matrix eine ist: $4 \cdot 4 \cdot 32$ Bit $= 512$ Bit $\widehat{=}$ 64 Byte. Die Model-Matrix muss also zwangsweise auf jedem System immer durch Push Constants hochgeladen werden können. Das Wörtchen 'Constant' jedoch birgt ein Problem: Die Model-Matrix wäre konstant, also jedesmal dieselbe. Das Modell würde sich nicht bewegen, rotieren oder sonst irgendwie verändern können. Um diese konstante Qualität wegzunehmen wurden die Command Buffer jedes Frame neu aufgenommen und somit implizit auch die *Descriptor Sets*, welche als Übermittler von Daten an Shader agieren, neu erstellt. Dies führt zu einem dramatischen Performanceverlust. Besser wäre es gewesen, einen grösseren Uniform Buffer hochzuladen und dann über eine Push Constant in ein Array an Model-Matrizen zu indexieren. Dies wurde jedoch absichtlich nicht umgesetzt, da es erstens nicht direkt funktionieren wollte und es zweitens Nachteile nach sich zieht: Entweder es wird ein dynamischer Uniform Buffer eingführt (kompliziert), oder es gibt eine Obergrenze an Modellen, welche dargestellt werden können. Deswegen wurde dies vorerst nicht umgesetzt, hauptsächlich wegen des Zeitdrucks. Die Leistung der Vulkan-Engine wurde aber nicht völlig vernachlässigt. Es wurde lange Zeit damit zugebracht, Multithreading für das Asset-Loading einzubauen. Dort gab es lange Probleme mit der Synchronisation, da zeitweise auf bis zu 12 Threads gleichzeitig gearbeitet wurde. Das System funktioniert über einen Handler-Thread und 12 Worker-Threads. Der Handler-Thread pusht dabei neu eintreffende Ladeaufträge zu einem Warteschlangen-Objekt der Standardbibliothek von C++ und der erste Worker-Thread, der frei ist, nimmt immer das vorderste Element dieser *std::queue* (engl. *queue*, 'Warteschlange'). So kann die CPU im Zusammenspiel mit Vulkan optimal ausgenutzt werden. Das Problem ist nur, dass Vulkan den Zugriff mehrerer Threads auf gewisse Objekte und Parameter selbst intern mit seinen *VkFence*'s, *VkSemaphore*'s oder *VkMemoryBarrier*'s synchronisiert und den Zugriff auf andere nicht. Dabei ist in der Vulkan Spezifikation genau definiert welche Parameter bei welchen Funktionen vom Programmierer mit externen Synchronisationsobjekten wie STL's *std::mutex* synchronisiert werden müssen.[32]

6.4 Lizensierung der Software

Beide Applikationen/Projekte sind unter der *GNU General Public License v3.0* lizensiert. In der Kurzfassung heisst dies, der Source-Code darf unter Angabe von Copyright, Quelle und Einbindung derselben Lizenz ohne jegliche Haftung oder Garantie des Entwicklers für kommerzielle, private und patentierte Zwecke genutzt werden. Die Software darf unter Angabe der Änderungen modifiziert und veröffentlich werden. Für die verwendeten Bibliotheken existieren Lizenzen innerhalb des Source-Codes oder wo nötig in separaten Dateien. Die Lizenzen der Libraries erlauben allesamt die Nutzung und Veröffentlichung in eigenen Projekten.

[31] The Khronos Vulkan Working Group. *Vulkan Specification - 36.1. Limit Requirements. (Englisch) [Vulkan Specifikation - 36.1. Limit-Anforderungen].* Aufgerufen: 03.11.2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#limits-required`.

[32] The Khronos Vulkan Working Group. *Vulkan Specification - 2.6 Threading Behaviour. (Englisch) [Vulkan Spezifikation - 2.6 Threading-Verhalten].* Aufgerufen: 12.11.2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#fundamentals-threadingbehavior`.

7 Das Doppelpendel

7.1 Was ist ein Doppelpendel?

Ein Doppelpendel ist im physikalischen Sinne ein beliebtes Modell zur Darstellung von deterministisch-chaotischen Prozessen. Das Pendel ist deterministisch, weil die Bewegungsgleichungen bekannt und berechenbar sind. Es ist chaotisch, weil sich das Doppelpendel bei kleinsten Änderungen der Startbedingungen drastisch anders verhält. In den Grundzügen ist ein Doppelpendel sehr einfach: An die Masse m_1 eines mathematischen Pendels der Länge ℓ_1 wird ein weiteres mathematisches Pendel mit Masse m_2 und Länge ℓ_2 gehängt. θ_1 und θ_2 bilden die Auslenkungswinkel gegenüber der Vertikalen. Die Verbindungsstücke sind starr, d.h. die Längen ℓ_1 und ℓ_2 sind konstant. Abbildung 9 veranschaulicht den generellen Aufbau eines Doppelpendels.

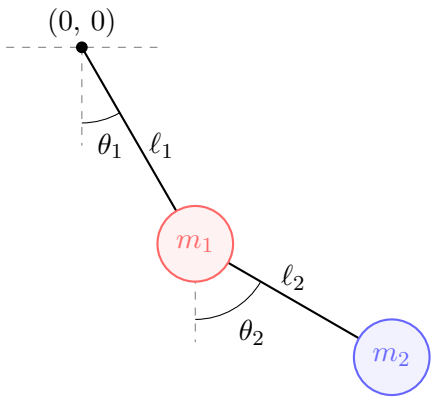


Abbildung 9: Schema eines Doppelpendels

7.2 Wieso ein Doppelpendel?

Da sich diese Arbeit grundsätzlich auf Floating-Point-Arithmetik bezieht, welche durch z.B. Rundungsfehler sehr schnell sehr ungenau werden kann, ist ein deterministisch-chaotisches System perfekt, um ebendies zu visualisieren. Da das Programmieren von kompetenten Render-Engines eine sehr aufwändige und komplizierte Angelegenheit ist, wurde auf ein einfach umzusetzendes deterministisch-chaotisches System gesetzt: das Doppelpendel. Andere Möglichkeiten wären zum Beispiel ein Drehpendel gewesen, oder sogar eine Wellensimulation, welche den berüchtigten Inverse Fast Fourier Transform (IFFT)-Algorithmus benötigt hätte. Letzteres wäre jedoch mit grossem Aufwand und viel Zeit verbunden gewesen, welche für diese Arbeit nicht zur Verfügung standen.

7.3 Physik hinter dem Doppelpendel

7.3.1 Variablen und Parameter

Die Variablen der beiden Pendel werden indiziert, d.h. für das erste Pendel Index 1, für das zweite Index 2. In den Formeln und Herleitungen werden folgende Variablen verwendet:

- $x_{1,2}$ = horizontale Position der Pendelmasse
- $y_{1,2}$ = vertikale Position der Pendelmasse
- $\theta_{1,2}$ = Winkel des Pendels gegenüber der Vertikalen, Gegenuhrzeigersinn ist positiv
- $\ell_{1,2} = const.$ = Länge des starren Verbindungsstückes
- $m_{1,2} = const.$ = Masse des Pendels
- $r_{1,2} \propto m_{1,2} = const.$ = Radius der Pendelmasse
- $g = const.$ = Ortsfaktor / Gravitationsbeschleunigung

7.3.2 Voraussetzungen und Annahmen

Wie berechnen sich die Positionen der beiden Massen relativ zum Ursprung? Für die folgende Herleitung werden gewisse Annahmen gemacht, resp. gewisse Voraussetzungen festgelegt:

- Es gibt keine Reibung und somit auch keinen Luftwiderstand.
- Die Verbindungsstücke mit Länge ℓ_1 und ℓ_2 haben keine Masse.
- Die Massen m_1 und m_2 sind konstant und werden als Massepunkte betrachtet.
- Die Radien $r_{1,2}$ der Kugeln sind proportional zur Masse der Kugel.
- Der Drehpunkt des ersten Pendels liegt in $(0, 0)$.
- Die y-Achse ist nach oben steigend, die x-Achse nach rechts. Die Achsen stehen im Ursprung senkrecht aufeinander und haben dieselben Einheiten.

7.3.3 Kinematik des Doppelpendels

Zu Beginn werden erst einmal die einfachen trigonometrischen Beziehungen benutzt, um Ausdrücke für die Positionen der Pendelmassen $M_1(x_1, y_1)$ und $M_2(x_2, y_2)$ in Abhängigkeit zu den von der Zeit t abhängigen Auslenkungswinkeln $\theta_1 = \theta_1(t)$ und $\theta_2 = \theta_2(t)$ zu finden. Dies wird komponentenweise gemacht. Es handelt sich somit um eine simple Umrechnung von Polar- in kartesische Koordinaten:

$$x_1 = \ell_1 \sin\theta_1 \tag{1}$$
$$y_1 = -\ell_1 \cos\theta_1 \tag{2}$$
$$x_2 = x_1 + \ell_2 \sin\theta_2 \tag{3}$$
$$y_2 = y_1 - \ell_2 \cos\theta_2 \tag{4}$$

Positionen abgeleitet nach der Zeit t ergeben die jeweiligen Geschwindigkeiten:

$$\frac{d}{dt}x_1 = \frac{d\theta_1}{dt}\ell_1 \cos\theta_1 \tag{5}$$
$$\frac{d}{dt}y_1 = \frac{d\theta_1}{dt}\ell_1 \sin\theta_1 \tag{6}$$
$$\frac{d}{dt}x_2 = \frac{d}{dt}x_1 + \frac{d\theta_2}{dt}\ell_2 \cos\theta_2 \tag{7}$$
$$\frac{d}{dt}y_2 = \frac{d}{dt}y_1 + \frac{d\theta_2}{dt}\ell_2 \sin\theta_2 \tag{8}$$

Und die Geschwindigkeiten wiederum nach der Zeit abgeleitet ergeben die Beschleunigungen:

$$\frac{d^2}{dt^2}x_1 = -(\frac{d\theta_1}{dt})^2\ell_1 \sin\theta_1 + \frac{d^2\theta_1}{dt^2}\ell_1 \cos\theta_1 \tag{9}$$
$$\frac{d^2}{dt^2}y_1 = (\frac{d\theta_1}{dt})^2\ell_1 \cos\theta_1 + \frac{d^2\theta_1}{dt^2}\ell_1 \sin\theta_1 \tag{10}$$
$$\frac{d^2}{dt^2}x_2 = \frac{d^2x_1}{dt^2} - (\frac{d\theta_2}{dt})^2\ell_2 \sin\theta_2 + \frac{d^2\theta_2}{dt^2}\ell_2 \cos\theta_2 \tag{11}$$
$$\frac{d^2}{dt^2}y_2 = \frac{d^2y_1}{dt^2} + (\frac{d\theta_2}{dt})^2\ell_2 \cos\theta_2 + \frac{d^2\theta_2}{dt^2}\ell_2 \sin\theta_2 \tag{12}$$

Hierbei entspricht die Winkelgeschwindigkeit $\omega = \frac{d}{dt}\theta$ der ersten, und die Winkelbeschleunigung $\alpha = \frac{d^2}{dt^2}\theta$ der zweiten Ableitung des Winkels nach der Zeit.

7.3.4 Mechanik des Doppelpendels

Auf die Pendelmassen wirken Kräfte. Zum einen die Kraft der Verbindungsstücke ($F_{1,2}$) auf die einzelnen Massen, zum anderen die Gravitationskraft. Die Idee hinter der folgenden betragsmässigen Betrachtung birgt das Aktionsprinzip resp. das zweite newtonsche Gesetz:

$$\vec{F} = m \cdot \vec{a}$$

Auf m_1 wirken die Kräfte der beiden Verbindungsstücke, also F_1 und F_2, sowie die Gravitationskraft $m_1 \cdot (-g)$ (negatives Vorzeichen da g in negative y-Richtung wirkt):

$$m_1 \frac{d^2 x_1}{dt^2} = -F_1 \sin\theta_1 + F_2 \sin\theta_2 \tag{13}$$

$$m_1 \frac{d^2 y_1}{dt^2} = F_1 \cos\theta_1 - F_2 \cos\theta_2 - m_1 g \tag{14}$$

Auf m_2 wirken die Kräfte des Verbindungsstücks zu m_1, also F_2, und die Gravitationskraft $m_2 \cdot (-g)$:

$$m_2 \frac{d^2 x_2}{dt^2} = -F_2 \sin\theta_2 \tag{15}$$

$$m_2 \frac{d^2 y_2}{dt^2} = F_2 \cos\theta_2 - m_2 g \tag{16}$$

7.3.5 Bewegungsgleichungen

Mit den Gleichungen (1) - (12) existieren drei verschiedene Arten auf die Position zu schliessen. Jedoch gibt es überall mindestens eine Unbekannte, welche auch in den Gleichungen (13) - (16) vorkommt: $\theta_{1,2}$, $\frac{d}{dt}\theta_{1,2}$ oder $\frac{d^2}{dt^2}\theta_{1,2}$. Der nächste Schritt besteht also darin einen Ausdruck für $\frac{d^2}{dt^2}\theta_{1,2}$ zu finden.

Die Gleichungen (15) und (16) werden nach $F_2 \sin\theta_2$ resp. $F_2 \cos\theta_2$ aufgelöst und in (13) resp. (14) substituiert:

$$m_1 \frac{d^2 x_1}{dt^2} = -F_1 \sin\theta_1 - m_2 \frac{d^2 x_2}{dt^2} \tag{17}$$

$$m_1 \frac{d^2 y_1}{dt^2} = F_1 \cos\theta_1 - m_2 \frac{d^2 y_2}{dt^2} - m_2 g - m_1 g \tag{18}$$

Weiter wird Gleichung (17) mit $\cos\theta_1$ und Gleichung (18) mit $\sin\theta_1$ multipliziert und gleichzeitig umgeschrieben:

$$F_1 \sin\theta_1 \cos\theta_1 = -\cos\theta_1 (m_1 \frac{d^2 x_1}{dt^2} + m_2 \frac{d^2 x_2}{dt^2}) \tag{19}$$

$$F_1 \sin\theta_1 \cos\theta_1 = \sin\theta_1 (m_1 \frac{d^2 y_1}{dt^2} + m_2 \frac{d^2 y_2}{dt^2} + m_2 g + m_1 g) \tag{20}$$

Zusammengefasst führt dies zu folgender Gleichung:

$$\sin\theta_1 (m_1 \frac{d^2 y_1}{dt^2} + m_2 \frac{d^2 y_2}{dt^2} + m_2 g + m_1 g) = -\cos\theta_1 (m_1 \frac{d^2 x_1}{dt^2} + m_2 \frac{d^2 x_2}{dt^2}) \tag{21}$$

Analog wird Gleichung (15) mit $\cos\theta_2$, Gleichung (16) mit $\sin\theta_2$ multipliziert und umgeschrieben:

$$F_2 \sin\theta_2 \cos\theta_2 = -\cos\theta_2 (m_2 \frac{d^2 x_2}{dt^2}) \tag{22}$$

$$F_2 \sin\theta_2 \cos\theta_2 = \sin\theta_2 (m_2 \frac{d^2 y_2}{dt^2} + m_2 g) \tag{23}$$

Was dann ebenfalls zu einer Gleichung zusammengefasst werden kann:

$$\sin\theta_2 (m_2 \frac{d^2 y_2}{dt^2} + m_2 g) = -\cos\theta_2 (m_2 \frac{d^2 x_2}{dt^2}) \tag{24}$$

Jetzt können die Gleichungen (21) und (24) mithilfe der Definitionen (9) bis (12) mit einem Computeralgebraprogramm gelöst werden. Somit können die folgenden Ausdrücke für $\frac{d^2}{dt^2}\theta_{1,2}$ gefunden werden:

$$\frac{d^2}{dt^2}\theta_1 = \frac{-g(2m_1+m_2)\sin\theta_1 - m_2 g\sin(\theta_1-2\theta_2) - 2\sin(\theta_1-\theta_2)m_2((\frac{d\theta_2}{dt})^2\ell_2 + (\frac{d\theta_1}{dt})^2\ell_1\cos(\theta_1-\theta_2))}{\ell_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \tag{25}$$

$$\frac{d^2}{dt^2}\theta_2 = \frac{2\sin(\theta_1-\theta_2)((\frac{d\theta_1}{dt})^2\ell_1(m_1+m_2) + g(m_1+m_2)\cos\theta_1 + (\frac{d\theta_2}{dt})^2\ell_2 m_2\cos(\theta_1-\theta_2))}{\ell_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \tag{26}$$

Dies sind die Bewegungsgleichungen für die Massen m_1 und m_2[33]. Hierbei handelt es sich um Differentialgleichungen. Es müssen jedoch keine komplizierten Lösungsverfahren angewendet werden, da diese Berechnungen numerisch durchgeführt werden.

7.4 Programmatische Umsetzung

Da es sich hierbei um nicht-konstant-beschleunigte Kreisbewegungen handelt, wurden im Quellcode pro Iteration die Winkelbeschleunigung zur Winkelgeschwindigkeit, und die resultierende Winkelgeschwindigkeit zum Winkel addiert:

```
1  float p1_theta;
2  float p2_theta;
3  float p1_vel;
4  float p2_vel;
5  float p1_acc;
6  float p2_acc;
7
8  void computePendulumState() {
9
10     [...]
11
12     p1_acc = getAccP1();
13     p2_acc = getAccP2();
14
15     [...]
16
17     p1_vel += p1_acc;
18     p2_vel += p2_acc;
19     p1_theta += p1_vel;
20     p2_theta += p2_vel;
21
22     [...]
23
24 }
```

Aber ist dies physikalisch vertretbar? Mit einem kleinen Trick, ja. Die Geschwindigkeit ist die Änderung des Ortes pro Zeit. Die Beschleunigung ist die Änderung der Geschwindigkeit pro Zeit. Dies ist praktisch die Definition der Differentialrechnung. 'Pro Zeit' $= \Delta t = const.$ Wenn das Δt infinitissimal klein wird, spricht man von dt. Es handelt sich dabei immer um dieselben Zeitabschnitte. Aber hier befinden wir uns in einer Computersimulation, wo Iterationen über eine Schleife garantiert nicht immer gleich lange dauern, und dort liegt auch das Problem. Wenn jedoch irgendwie sichergestellt wäre, dass die Iterationen immer etwa gleich lange dauern und das Δt somit *fast* konstant wäre ($\Delta t \approx const.$), könnte

[33] Erik Neumann. *Double Pendulum. (Englisch) [Doppelpendel].* Aufgerufen: 20.10.2019. URL: `https://www.myphysicslab.com/pendulum/double-pendulum-en.html`.

man dies eher vertreten. Der Trick besteht darin, dass man ein Prinzip aus der Simulationstechnik anwendet: der sogenannte Timestep. Als Timestep wird ein Verfahren bezeichnet, bei dem man in einer Iteration mehrmals dieselbe Berechnung durchführt, um gewisse Prozesse in Simulationen zu beschleunigen. Mehrere Iterationen pro Iteration der Hauptschleife helfen hier aber nicht weiter, wenn das Ziel ist, ein konstantes Δt zu garantieren. Deshalb wird das Verfahren umgekehrt: Es werden weniger Iterationen pro Hauptschleifeniteration gemacht, und durch eine Zeitdifferenz vergleichsweise genau gezählt. Und so funktioniert es: Am Anfang der Schleife nimmt man einen Zeitstempel auf. Am Ende der Schleife schreibt man diesen in eine statische Variable, welche bei der nächsten Iteration abrufbar ist. Bei der nächsten Iteration wird der Zeitstempel am Anfang der Schleife aktualisiert. Wenn jetzt die Differenz zwischen diesem neuen Anfangszeitstempel und dem Zeitstempel der letzten Iteration grösser als z.B. $\frac{1}{60}s$ (entspricht 60 Iterationen pro Sekunde = 60 Hz) ist macht man die nächste Iteration, ansonsten wird gewartet bis die Differenz grösser ist. Hier wurde der eingebaute Timer der Windowing-Library GLFW 3 *glfwGetTime()* verwendet. So wird mit einer minimalen Ungenauigkeit in der Grössenordnung von einigen Nano- bis Mikrosekunden[34] garantiert, dass es keine nennenswerten Unterschiede beim Δt gibt. So wurde das im Quellcode umgesetzt:

```
void computePendulumState() {

    double now = glfwGetTime();
    static double last = 0.0;

    if (now - last >= 1 / 60.0f) {

        [...]

        p1_acc = getAccP1();
        p2_acc = getAccP2();

        [...]

        p1_vel += p1_acc;
        p2_vel += p2_acc;
        p1_theta += p1_vel;
        p2_theta += p2_vel;

        [...]

        last = now;

    }

}
```

Und deshalb sollten diese einfachen Additionen physikalisch gesehen kein Problem sein. All dies führt zu einer nahtlosen Simulation, welche allerdings ein Problem mit sich bringt: Die Gesamtenergie des Systems wird je nach Wahl der Startparameter über kurz oder lang gegen $\pm\infty$ gehen, da alle Störkräfte wie Reibung und Luftwiderstand vernachlässigt werden und vor allem da mit 32-bit Floats gerechnet wird, was zu Rechenungenauigkeiten führt. Wenn die aktuelle Energie E_{tot} aufgrund von Rechenungenauigkeiten über die theoretisch maximal mögliche Energie E_{max} hinaussteigt, explodieren die Werte. Dies wird durch die iterative Struktur der Bewegungsgleichungen (neue Winkelbeschleunigung wird in Abhängigkeit der alten Winkelgeschwindigkeit berechnet) verstärkt. Dadurch könnte es sein, dass die Zahlen sehr schnell steigen und sehr hohe Beschleunigungen, Geschwindigkeiten und Winkel

[34] GLFW 3. *GLFW 3 Documentation. (Englisch) [GLFW 3 Dokumentation].* Aufgerufen: 23.10.2019. URL: `https://www.glfw.org/docs/latest/group__input.html`.

herauskommen. Nach IEEE-754-Standard können 32-bit Floating-Point-Zahlen Werte bis maximal $(2-2^{-23})\cdot 2^{127} \approx 3.4028235\cdot 10^{38}$ darstellen. Wird versucht, einen grösseren Wert in diesen zu kleinen Speicher zu schreiben, könnte dies einen ungehandelten Buffer-Overflow zur Folge haben. Es würde somit mit ungültigen Werten weitergerechnet. Im extremsten Fall würden die Matrix-Berechnungen mit den Variablen für die Winkel, welche als 32-bit Floats definiert wurden, Resultate liefern, die keinen Sinn machen (sog. NaN's). Das Doppelpendel würde nicht mehr dargestellt werden können. Und das Experiment zeigt, dass dem auch so ist. Mit gewissen Startwerten für Startgeschwindigkeiten, Startbeschleunigungen und Auslenkungswinkel passiert genau dies. Über kurz oder lang bauen sich die Variablen gegen $\pm\infty$ auf. Dies kann in Bruchteilen von Sekunden oder über einen längeren Zeitraum, wo man sieht wie die Rotationen der Pendel immer ekstatischer werden, geschehen. Und dann ist das Pendel plötzlich einfach weg. Dies war während der Implementation des Doppelpendels lange Zeit ein grosses Problem und hing vor allem mit einem Umrechnungsfehler von Grad- zu Bogenmass zusammen, wodurch mit viel völlig falschen Werten gerechnet wurde. Mit den richtigen Parametern lässt sich eine stabile Simulation erzeugen, welche nicht plötzlich kollabiert. Mit gewissen Startwerten für gewisse Parameter lässt sich trotzdem noch ein Buffer-Overflow erzeugen.

Da dieser Umrechnungsfehler erst sehr viel später entdeckt wurde, wurde zuerst angenommen, dass die Bewegungsgleichungen nicht stimmen oder dass irgendetwas mit der Implementation falsch ist. Dem war nicht so, jedoch wurde eine energetische Betrachtung durchgeführt um zu überprüfen, dass es sich um korrekte Gleichungen handelt und dabei kamen interessante Dinge zutage.

$$E_{\text{kin}} = \frac{1}{2}mv^2 \tag{27}$$

$$E_{\text{pot}} = mgh \tag{28}$$

$$E_{\text{tot}} = E_{\text{kin}} + E_{\text{pot}} = \frac{1}{2}mv^2 + mgh = E_{\max} \tag{29}$$

Die Gesamtenergie E_{tot} darf kleiner werden, darf konstant bleiben aber darf nicht steigen. Somit ist der höchstmögliche Wert, den die Gesamtenergie haben darf, durch die Energie, welche mit den Startparametern berechnet wird, gegeben. Die Maximalenergie entspricht der totalen Energie zu Beginn der Simulation. Die Gesamtenergie E_{tot} setzt sich aus der potenziellen und der kinetischen Energie des Systems zusammen und entspricht somit:

$$E_{\text{pot}} = m_1 g(-\ell_1 \cos\theta_1) + m_2 g(-\ell_1 \cos\theta_1 - \ell_2 \cos\theta_2) \tag{30}$$

$$E_{\text{kin}} = \frac{1}{2}m_1(\ell_1\frac{d\theta_1}{dt})^2 + \frac{1}{2}m_2((\ell_1\frac{d\theta_1}{dt})^2 + (\ell_2\frac{d\theta_2}{dt})^2 + 2\ell_1\ell_2\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}\cos(\theta_1-\theta_2)) \tag{31}$$

$$\begin{aligned} E_{\text{tot}} &= m_1 g(-\ell_1 \cos\theta_1) + m_2 g(-\ell_1 \cos\theta_1 - \ell_2 \cos\theta_2) \\ &\quad + \frac{1}{2}m_1(\ell_1\frac{d\theta_1}{dt})^2 + \frac{1}{2}m_2((\ell_1\frac{d\theta_1}{dt})^2 + (\ell_2\frac{d\theta_2}{dt})^2 + 2\ell_1\ell_2\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}\cos(\theta_1-\theta_2)) = E_{\max} \end{aligned} \tag{32}$$

$E_{\max}$ ist somit die Maximalenergie des Systems. Diese darf aufgrund der Energieerhaltung zu keinem Zeitpunkt überschritten werden. Jedoch wurde im Verlauf der Überprüfung der Bewegungsgleichungen herausgefunden, dass sie dies tut und nicht konstant bleibt. Der Wert schwankt ein wenig. Es werden realistische visuelle Resultate produziert, jedoch bleibt die totale Energie nicht konstant. Wenn die Simulation in der Quelle der Bewegungsgleichungen[35] angeschaut wird, kann dort eine Schaltfläche für die Energie gefunden werden. Je nach Differentialgleichungslösungsverfahren bleibt die Energie konstant oder nicht. Dies bestätigt zuerst einmal die Tatsache, dass die Bewegungsgleichungen nicht zu 100% realitätsgetreu sind. Die Energie verhält sich nicht willkürlich, sondern sie schwankt einfach ein wenig. Im Zuge der Überprüfung der Bewegungsgleichungen wurden die Bewegungsgleichungen, welche über die Anwendung des Lagrange-Formalismus gefunden werden können,[36] implementiert. Allerdings bleibt auch dort die Energie nicht konstant, sondern eskaliert sogar komplett (infolge Buffer-Overflow verschwindet das Pendel). Aus diesem Grund kann dies so als Simulation, als Annäherung an die Realität akzeptiert werden, auch wenn die Energie nicht konstant bleibt.

[35] Erik Neumann, *Double Pendulum. (Englisch) [Doppelpendel]*.

[36] Eric W. Weisstein. *Double Pendulum. (Englisch) [Doppelpendel]*. Aufgerufen: 16.11.2019. URL: `http://scienceworld.wolfram.com/physics/DoublePendulum.html`.

Diese Umsetzung erfolgt nun allerdings auf der CPU und nicht auf der Grafikkarte über die jeweilige API. Dies rechtfertigt sich dadurch, dass durch die Computation auf der Grafikkarte die Berechnungen abhängig von der Anzahl Vertices einige 100'000 bis sogar mehrere Millionen mal pro Sekunde (je nach Ort der Berechnungen; Vertex, Fragment oder sogar Compute Shader) gemacht werden müssten. Die Modelle für das Doppelpendel bestehen aus knapp 15'000 Vertices für eine Massenkugel und 160 Vertices für ein Verbindungsstück. Dies entspricht bei mindestens 60 Frames pro Sekunde (FPS)

$$30'320 \text{ Vertices} \cdot 60 \text{ FPS} \approx 180'000 \text{ Iterationen}$$

der Formeln pro Sekunde nur schon für eine Implementation in den Vertex Shadern. Angesichts der erschlagenden FPS-Zahlen von teilweise weit über 5'000 wird die Anzahl der Iterationen sehr schnell sehr gross. Dies würde durch die doch recht aufwändige Berechnung der Bewegungsgleichungen und Energiegleichungen sehr grosse Performance-Einbussen nach sich ziehen und wäre API-spezifisch zu implementieren, was die Sache komplizierter macht. Es würde die Leistungsmessungen stark verfälschen und wäre somit der schlechtest-mögliche Weg, so eine Simulation umzusetzen. Ausserdem gibt es keinen Weg Ergebnisse von den Shadern zur Konsole auszugeben. Die Floating-Point-Ungenauigkeit auf der Grafikkarte sollte ebenfalls nicht gross anders ausfallen, als auf der CPU, da beide Male mit 32-bit Floats gerechnet wird. Sollte sie es doch tun, dann würde das einem Vesatz der Modelle von einer minimalen Distanz entsprechen, welcher niemals bemerkt werden könnte. Dies kann jedoch nur schlecht überprüft werden, von dem her wird es einfach einmal angenommen. Andererseits traten ja durch das Verschwinden der Pendel durch Buffer-Overflows schon interessante Artefakte auf, welche zumindest teilweise auf Floating-Point-Ungenauigkeiten zurückzuführen sind. Von dem her ist bestätigt, dass die CPU's anfällig für Rechenungenauigkeiten sind.

8 Schlussfolgerung

8.1 Ergebnisse

Die Applikationen wurden unter Ubuntu 18.04 TLS, Linux Mint Cinnamon 19.02 und Microsoft Windows 10 mit verschiedenen Konfigurationen an Computerkomponenten getestet. In den Tabellen 2 bis 4 sind die Resultate der Tests aufgeführt. Die Grösse der *Frametime* bezeichnet dabei die Zeit bis ein Frame, also ein Bild, gerendert wird. So lässt sich die Leistung der API messen, denn je mehr Frames pro Sekunde (also je kürzer die Frametime) desto leistungsfähiger ist die API oder das Programm.

Tabelle 2: Resultate unter Windows 10

Windows 10			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti[37]	NVIDIA GTX 1050 Ti[38]	Intel UHD Graphics 630[39]
OpenGL Version	4.6	4.6	4.5
Vulkan Version	1.1.122	1.1.122	1.0
OGL	flüssig	flüssig	flüssig
Frametime OGL [ms]	0.1695195	1.376819	1.147673
VK	flüssig	flüssig	-
Frametime VK [ms]	0.533498	3.483420	-

Tabelle 3: Resultate unter Linux Mint 19.02

Linux Mint 19.02			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
OpenGL Version	4.6	4.6	4.5
Vulkan Version	1.1.122	1.1.122	1.0
OGL	stockend	stockend	stockend
Frametime OGL [ms]	16.634210	16.623408	16.632190
VK	flüssig	flüssig	-
Frametime VK [ms]	16.623408	16.682348	-

Tabelle 4: Resultate unter Ubuntu 18.04

Ubuntu 18.04			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
OpenGL Version	4.6	4.6	4.5
Vulkan Version	1.1.122	1.1.122	1.0
OGL	stockend	stockend	stockend
Frametime OGL [ms]	16.658375	16.687655	16.623491
VK	flüssig	flüssig	-
Frametime VK [ms]	16.647447	16.679864	-

[37] techpowerup.com. *NVIDIA GeForce GTX 1080 Ti*. Aufgerufen: 12.11.2019. URL: `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`.

[38] techpowerup.com. *NVIDIA GeForce GTX 1050 Ti*. Aufgerufen: 12.11.2019. URL: `https://www.techpowerup.com/gpu-specs/geforce-gtx-1050-ti.c2885`.

[39] wikichip.org. *UHD Graphics 630 - Intel. (Englisch) [Intel UHD Graphics 630]*. Aufgerufen: 12.11.2019. URL: `https://en.wikichip.org/wiki/intel/uhd_graphics/630`.

Die Werte für die Frametimes stellen ein arithmetisches Mittel über die ersten 100 Sekunden Laufzeit der Programme dar, wobei alle 10 Sekunden ein Mittel über die letzten 10 Sekunden errechnet wurde. Die Rohdaten können in Tabellenform unter Anhang E gefunden werden. Bei der Datenaufnahme wurden alle Hintergrundprogramme und sonstigen Prozesse, welche grössere Leistung beanspruchten, geschlossen. Es wurde nur das Doppelpendel dargestellt, ohne Hintergrund oder andere Modelle, damit die Anzahl Vertices bei beiden Applikationen gleich war. Des weiteren wurde keine Mausbewegungen gemacht oder Tasten gedrückt. Beide Applikationen hatten 8-faches MSAA und MipMapping aktiviert. Ausserdem wurden alle Ausgaben auf der Konsole bis auf die Frametimes unterdrückt und somit Prozessorzeit gespart.

Jetzt stellt sich die Frage nach der Genauigkeit der Werte. Der Timer von GLFW *glfwGetTime()* hat ja laut Dokumentation eine Ungenauigkeit von einigen Nano- bis höchstens Mikrosekunden.[40] Die Resultate können doch unmöglich auf mindestens die Grössenordnung der Nanosekunde genau sein. Jetzt kommt das Raffinierte, denn sie sind es. Und zwar, weil die Frametimes nicht durch eine Zeitdifferenz gemessen, sondern die tatsächlich gerenderten Frames gezählt und davon die Kehrwerte genommen wurden, ganz nach der Frequenzformel:

$$T = \frac{1}{f}$$

Hier steht T für die Periodendauer und f für die Frequenz. Die Resultate sind sehr genau, da die Iterationsfrequenz der Hauptschleife exakt bestimmbar ist. Aber, um die Iterationsfrequenz zu bestimmen, muss doch auch eine Zeit gemessen werden, dann gibt es doch auch wieder Ungenauigkeiten? Technisch gesehen wäre dies eine berechtigte Frage, da nur durch einen Kehrwert nicht plötzlich mehr Genauigkeit vorhanden ist. Aber dadurch, dass die Frametime mindestens 3 Grössenordnungen grösser als der Fehler von *glfwGetTime()* ist, entfällt dieser Einwand. Es würden vielleicht 1 bis maximal 2 Frames mehr pro Sekunde ausmachen. Bei durchschnittlichen FPS von über 5'000 macht dies keinen nennenswerten Unterschied mehr. Die Ungenauigkeit des Timers wurde hier somit umgangen. Diese wäre hier aber offensichtlich nicht zum Problem geworden, um eine qualitative oder quantitative Aussage über die Leistung zu machen, denn der Fall ist relativ eindeutig.

8.2 Interpretation

Unter Windows 10 laufen beide API's problemlos. Bemerkenswert hierbei ist, dass OpenGL als wesentlich ältere API besser läuft als die Vulkan-Implementation. Unter Debian lässt sich V-Sync nicht ausschalten, was die Performance-Messung unter Linux unmöglich macht. V-Sync (Vertical Synchronization) synchronisiert die Bildwiederholfrequenz der Applikation mit der des Bildschirms. Diese beträgt auf den Testsystemen 60 Hz, also 60 Frames pro Sekunde. Die Frametime ist somit auf 16.666 ms getrimmt. Unter Linux kann deswegen keine Aussage über die Leistungsfähigkeit der API-Implementationen gemacht werden. Da die beiden Testsyteme beide Derivate von Debian sind, sind die Ergebnisse nicht repräsentativ für die gesamte Linux-Familie von Betriebssystemen. Unter Windows jedoch überrascht OpenGL mit seiner Performance und schlägt Vulkan wider Erwarten um Längen. Was könnten Gründe dafür sein? Bei der Umsetzung der Vulkan-Applikation wurden einige performancelastige Workarounds und unschöne Lösungen eingebaut. Der grösste davon war derjenige mit den Descriptor Sets und der erneuten Aufnahme der Command Buffer in jedem Frame. Dies geschah zum Teil wegen mangelnder Zeit oder mangelnder Erfahrung mit Vulkan um gewisse Dinge auf dem idealsten Weg umzusetzen. Ein weiterer Faktor war, dass die Engines ab einem gewissen Punkt einfach nur noch funktionieren mussten, relativ egal wie leistungsfähig sie waren, da das Doppelpendel noch implementiert und gerendert werden musste.

Weiterhin auffällig ist der Fakt, dass OGL mit Intel's Integrierten Grafiken unter Windows besser läuft als mit der dedizierten 1050 Ti. Ein Blick in den Taskmanager zeigt, dass die integrierten Grafiken auf 100% laufen, während die dedizierte Karte nur auf etwa 30% steht. Das heisst, die Applikation kann die dedizierte Karte nicht voll ausnutzen. Bei Vulkan ist die Auslastung der dedizierten Karten 100%. Diese werden somit voll ausgenutzt. Dies kann bei OpenGL auf nicht vorhandene Parallelisierung

[40] GLFW 3, *GLFW 3 Documentation. (Englisch) [GLFW 3 Dokumentation].*

und bei Vulkan auf das grosse Parallelisierungspotenzial der API zurückgeführt werden. Weil die API Spec-Version 1.1.114 für Vulkan verwendet wurde, konnten für integrierte Grafiken mit Vulkan keine Daten erhoben werden. Die Vulkan Implementation findet keine GPU, welche diese Version von Vulkan unterstützt und stürzt ab, da integrierte Grafiken nur Vulkan 1.0 unterstützen.

8.3 Beantwortung der Leitfragen

OpenGL ist durchaus im Jahre 2019 immer noch sehr kompetitiv und eine echte Konkurrenz zu Vulkan, aufgrund verschiedenster Vorteile. Hauptsächlich jedoch wegen der Einfachheit, welche trotzdem zu den exakt selben Ergebnissen führt. Leistungstechnisch sticht OpenGL im Praxisvergleich unerwartet hervor, was sich vermutlich eher auf mangelnde Erfahrung mit Vulkan und somit verschwendetem und nicht ausgenutztem Optimierungspotenzial zurückführen lässt. Ein weiterer Grund für die unerwarteten Performanceverhältnisse könnten allfällige Probleme bei den Treibern sein. Der Vulkan-Treiber ist auf wenig Optimierung aus und möchte alles den Programmierer machen lassen. Jedoch kann es sein, dass ebendieser Treiber durch Vulkans Asynchronität Ressourcen spart, welche nicht gebraucht werden und deswegen die Vulkan-Applikation im Vergleich langsamer läuft. Auf der anderen Seite hat auch OpenGL zeitweise seine Schwierigkeiten. Unter Linux wird bei der OpenGL-Applikation kein flüssiges Bild dargestellt, intern jedoch werden 60 mal pro Sekunde alle Berechnungen durchgeführt. Dort gibt es anscheinend ein Problem, welches möglicherweise mit dem X Window System zusammenhängen könnte. Dies wäre dann auch wieder eine Sache des Treibers, respektive ein Problem der Windowing Library GLFW. Ausserdem lässt sich bei Linux die vertikale Synchronisation V-Sync nicht deaktivieren. Diese synchronisiert die Ausgabe mit der Bildwiederholfrequenz des Monitors und macht so einen Leistungsvergleich unmöglich. OpenGL hat sich über fast 30 Jahre nun immer wieder bewährt und wird wohl noch eine Zeit mit aktuelleren Grafik-API's mithalten können. Vermutlich wird es erst dann, wenn Khronos den Support und die Weiterentwicklung einstellt, von der Konkurrenz überholt werden.

Zwischen den beiden Engines kann ein grosser Unterschied bezüglich des Doppelpendels festgestellt werden. Durch die berüchtigten Floating-Point-Ungenauigkeiten verhalten sich die Pendel schon nach wenigen Sekunden bis Minuten komplett anders, wenn die selben Parameter gewählt werden. Dies kann sehr schön visualisiert werden, wenn die Pendel gleichzeitig nebeneinander gestartet werden. In den GitHub-Repositories kann ein eigens erstelltes Video dazu gefunden werden. Auf unterschiedlichen Computersystemen verhalten sich die Pendel noch einmal anders. Vermutlich geschieht dies, weil die Floating-Point-Arithmetik von Prozessor zu Prozessor ein kleines bisschen variiert und dadurch andere Floating-Point-Ungenauigkeiten entstehen. Dort sind die Werte jeweils sehr unterschiedlich und das Verhalten des Pendels ebenso. Das Pendel verhält sich perfekt chaotisch, da eine sehr geringfügige Änderung eines Startparameters eine drastische Verhaltensänderung zur Folge hat. Die Floating-Point-Fehler verstärken also das chaotische Verhalten des Pendels.

Die beiden Engines sind funktional und im Hinblick auf Erweiterbarkeit programmiert worden. Die zwei Programme liefern programmiertechnisch ein starkes Ergebnis, vor allem die Vulkan-Applikation, da praktisch keine vorherige Erfahrung vorhanden war.

8.4 Eigene Erfahrungen

Mit Vulkan zu arbeiten ist mühsam. Die Validation Layer Informationen sind nicht allzu hilfreich. Mit OpenGL können sehr viel schneller die gleichen, oder sogar die besseren Ergebnisse produziert werden. Um das volle Leistungspotenzial von Vulkan auszunutzen, bedarf es vieler Optimierungen, welche zeitaufwändig und kompliziert zu implementieren sind. Diese wurden hier aufgrund des Zeitdrucks oft weggelassen, was zu spürbaren Performanceverlusten geführt hat. Teilweise wurde Vulkans Low-Level-Eigenschaft zum Problem, da Vulkan durch diese Optimierbarkeit ein sehr hoher computertechnischer Wert zukommt. Wurde dieser nicht vollends ausgenutzt, schlug sich dies direkt in den Zahlen zur Leistungsmessung nieder. Mit Vulkan passieren rasch relativ einfache Flüchtigkeitsfehler, welche in stundenlanger Arbeit wieder repariert werden müssen. In der Entwicklungsphase der Vulkan-Applikation geschah dies mehrere Male. Das beste Beispiel war ein Fehler während der Entwicklung, welcher satte 7 Tage Suchaufwand gekostet hat, obwohl nur ein einziges Zeichen falsch war: Bei einer

Rotation wurde ein Nullvektor als Rotationsachsenparameter übergeben. Dort hätte eine 0 zu einer 1 geändert werden müssen. Diese Dinge waren sehr frustrierend und kamen leider viel zu oft vor. Es sind die Sorte Fehler, welche ein Compiler eigentlich sofort zurückmelden müsste und die nach 10 Sekunden korrigiert wären. Aber da Vulkan keine Debug-Informationen gibt mag das schnell mal ein paar Tage Aufwand bedeuten.

OpenGL funktioniert sehr zuverlässig und ohne grosse Schwierigkeiten. Dies mag durchaus zu einem gewissen Grad an der bereits vorhandenen Erfahrung mit OpenGL liegen, jedoch ganz sicher nicht ausschliesslich. OpenGL ist faktisch wesentlich einfacher aufgebaut als Vulkan. Man rufe sich die Anzahl Zeilen Code der beiden Programme erneut in Erinnerung: 3'500 zu 8'500. Die Umsetzung von OpenGL in einer Render-Engine ist eine vergleichsweise einfache Sache. Eine genau so starke Vulkan-Implementation ist jedoch eine Herausforderung für sich.

Die Implementation des Doppelpendels war eine sehr interessante Aufgabe, vor allem da gut versteckte Fehler gefunden werden mussten, welche auch hier durch Unaufmerksamkeit entstanden sind, im Zeitplan aber grosse Verzögerungen verursachten. Die Betrachtung der Energie des Systems förderte weitere interessante Dinge zutage und führte indirekt den Begriff des Lagrange-Formalismus ein. Zuerst wurde noch versucht über allfällige Drehimpulserhaltung eine zweite Gleichung für die zwei unbekannten Geschwindigkeiten zu finden und somit ein lösbares Gleichungssystem auf die Beine zu stellen, welches die Berechnung der theoretisch maximal möglichen Geschwindigkeiten der beiden Massepunkte des Pendels möglich gemacht hätte. Dies wäre gebraucht worden, hätte das Problem mit der explodierenden Energie nicht anders gelöst werden können. Die Drehimpulserhaltung bei einem Doppelpendel ist noch einmal eine sehr interessanter Aspekt, welche jedoch nicht weiter verfolgt wurde, da nur ein einziger Erhaltungssatz benötigt wurde, um die Korrektheit der Gleichungen zu verifizieren/falsifizieren.

9 Reflexion

Die beiden Engines wurden mit mehr Features geplant, als schlussendlich implementiert wurden. Dies aufgrund der unerwarteten Ausmasse der Schwierigkeiten mit Vulkan. Es war vorerst geplant, Schattenberechnungen zu implementieren. Diese mussten leider weggelassen werden, obwohl sie sehr spannend und wichtig für gute graphische Darstellungen gewesen wären. Das Doppelpendel kam nach einigen Schwierigkeiten mit den Formeln für Energie und den Bewegungsgleichungen am Ende dann doch exakt so zustande, wie ursprünglich geplant. Mit der Space-Taste lässt sich das Pendel stoppen, mit T und B können die Auslenkungswinkel verändert werden. Diese zwei Dinge waren nie geplant, aber machen das Pendel so viel dynamischer. Ausserdem kann mit den Tasten 1, 2 und 3 zwischen den Darstellungsmöglichkeiten gewechselt werden. 1 stellt die Modelle normal dar, 2 als Dreiecke mit den Faces und 3 als Vertices, also nur als Punkte. Mit F kann zwischen den Kamera-Modi gewechselt werden (Freies Herumfliegen oder Rotation um das Pendel). Durch Makros können verschiedene grafische Features und Pre-Processing-Schritte an-/ausgeschalten werden und das Laden von 3D-Modellen wurde zwei Mal mit zwei verschiedenen Libraries implementiert. Die Software ist plattformübergreifend, kann selber kompiliert werden und funktioniert auch mit Multithreading zuverlässig. Die Engines sind performativ, da durchdacht und mit einem Plan an die Sache herangegangen wurde. Sie sind dynamisch programmiert und gut weiterentwickelbar. Das Doppelpendel ist physikalisch annähernd korrekt und wird realistisch simuliert. Zwischen den Engines lassen sich bei gleichen Ausgangsbedingungen mit demselben Computersystem stark andere Verhaltensweisen beobachten. Diese lassen sich einzig und allein auf Floating-Point-Rechenfehler zurückführen, da die Implementation des Pendels exakt dieselbe ist und die Iterationsfrequenz der Bewegungsgleichungen auf 60 Hz festgelegt ist. Die energetische Betrachtung hat ausserdem einige interessante Aspekte zutage gefördert und war sehr aufschlussreich. Es wurden dadurch weitere Fragen aufgeworfen:

- Wieso ist die Energie bei numerischen Berechnungen mit zwei Varianten der Bewegungsgleichungen nicht konstant?
- Wieso ist die Energie und das Vehalten des Pendels bei verschiedenen Differentialgleichungslösungsverfahren unterschiedlich?
- Betrachtung des totalen Drehimpulses eines Doppelpendels

Die eigenen Voraussagen haben sich nur zum Teil bewahrheitet. Da Vulkan in der Schlussrechnung weitaus schlechter abschnitt als OpenGL wurde hier die eigene Hypothese wiederlegt. Dass unerwartete Schwierigkeiten während der Entwicklung auftreten würden wurde korrekt antizipiert, jedoch niemals in diesem Ausmasse. Für das Doppelpendel hat sich die Voraussage bewahrheitet bezüglich interessanten physikalischen Problemen. Es wurden einige theoretische Betrachtungen des chaotischen Systems durchgeführt (Energie-/Drehimpulserhaltung, Gleichungen über Lagrange'sche Mechanik), wovon die energetische Betrachtung tatsächlich gebraucht werden konnte und sogar neue Fragen aufwarf. Beim Drehimpuls traten Schwierigkeiten auf und die Formeln waren vermutlich nicht ganz korrekt, deswegen wurden diese schlussendlich nicht in die schriftliche Arbeit aufgenommen. Diese komplexeren Betrachtungen verantworten als Nebeneffekt Einblicke in höhere mathematische und physikalische Konzepte, welche einmal mehr sehr interessant waren, jedoch nur grob an der Oberfläche verstanden wurden.

Die Engines werden an diesem Punkt nicht einfach so stehen gelassen, sondern auch nach Abgabe der Arbeit weiterentwickelt. Nach einem halben Jahr wäre es schade, die Projekte jetzt in einem unvollständigen Zustand stehen zu lassen.

Persönliches Fazit: Dieses Projekt war eine sehr interessante und spannende, wenn auch zum Teil nervenaufreibende und frustrierende Arbeit mit unerwarteten Ergebnissen. Ich würde diese Arbeit trotz den Rückschlägen genau so erneut machen, wenn ich die Wahl hätte, da es durchaus im grossen Ganzen sehr Spass gemacht hat und den Willen weiterzumachen in mir geweckt hat. Seit Anfang der Entwicklung der Applikationen wurden praktisch täglich im Schnitt zwischen 1 und 1.5 Stunden in dieses Projekt investiert, vermutlich 80% davon für die Vulkan-Engine. Es gab nur wenige Tage an denen nicht programmiert wurde. Insgesamt kommen so mindestens 200 Arbeitsstunden zusammen.

10 Zusammenfassung

In dieser Arbeit ging es grundsätzlich um die Programmierung von zwei Render-Engines: eine mit Vulkan, eine mit OpenGL. Dies wurde erfolgreich umgesetzt, allerdings nicht mit allen Features wie ursprünglich geplant, da unerwartet grosse Verzögerungen durch Schwierigkeiten bei der Programmierung der Vulkan-Engine auftraten. Die zwei Renderer sind abstrahiert, objektorientiert, plattformübergreifend und erweiterungsfreundlich programmiert worden. Mit einer einzigen Zeile Code lässt sich ein Modell verschiedenster Dateiformate laden, darstellen und bewegen. Die Engines gebrauchen das klassische Phong-Lighting-Modell, welches das Licht von einer Lichtquelle realitätsnah simuliert. Schlussendlich entstanden zwei mittelklassige Render-Engines, wobei mit mehr Zeitreserven sehr viel mehr umsetzbar gewesen wäre. Angesichts der Ein-Mann-Produktion über einen relativ knappen Zeitraum von ein paar Monaten hinweg dürfen sich die Ergebnisse vor allem angesichts der nicht vorhandenen Erfahrung mit Vulkan durchaus sehen lassen. Durch diese plötzliche Zeitknappheit wurden bei Vulkan einige unschöne Workarounds und provisorische Übergangslösungen eingebaut, um die Engine vorerst zum Funktionieren zu bringen. Insgesamt wurden so über 12'500 Zeilen Quellcode geschrieben. Als die Programme schlussendlich so funktionierten, wie sie sollten, wurde eine Doppelpendel-Simulation in beiden Engines implementiert. Diese fundierte auf physikalisch korrekten Überlegungen und Annahmen und sah visuell sehr realistisch aus. Bei der Umsetzung dieser Simulation traten ebenfalls mehrere Probleme auf, welche vor allem mit einem Umrechnungsfehler von Bogen- zu Gradmass zu tun hatten, welcher sehr versteckt war und erst viel zu spät bemerkt wurde. Deswegen wurde erst einmal angenommen, dass mit der Implementation oder den Bewegungsgleichungen irgendetwas fehlerhaft war. Dies war nicht der Fall, jedoch wurden durch eine energetische Betrachtung des Systems zur Überprüfung der Bewegungsgleichungen interessante Aspekte entdeckt. Die totale Energie müsste bei korrekten Bewegungsgleichungen konstant bleiben. Dies tat sie jedoch nicht und je nach Differentialgleichungslösungsverfahren schwankte sie unterschiedlich stark. Aufgrund dessen wurden die Bewegungsgleichungen, welche sich nicht über die Newtonsche Mechanik (wie sie bis anhin eingebaut waren), sondern über die Lagrange'sche Mechanik mit dem Lagrange-Formalismus herleiten lassen, implementiert. Auch diese schienen die Energieerhaltung zu brechen. Deswegen wurden die Bewegungsgleichungen einfach so akzeptiert, wie sie waren, da sie korrekte visuelle Resultate ablieferten.

Performancetechnisch gab es auch hier zum Teil erwartete und zum Teil unerwartete Wendungen: OpenGL schnitt wesentlich besser und Vulkan wesentlich schlechter ab als erwartet. Dies erklärt sich durch semiideale Workarounds bei Vulkan, Optimierungen, Ressourcen-Alloziierung und sonstige Eigenheiten der jeweiligen Treiber und weit verbreiteter, leistungsfähiger OpenGL-Support der Hardware, da OpenGL immerhin schon fast 30 Jahre alt ist und Vulkan erst 2016 auf den Markt kam.

Die Implementationen des Doppelpendels zeigen auf demselben Computersystem stark verschiedene Verhaltensweisen mit denselben Startparametern. Diese lassen sich einzig und allein auf Floating-Point-Rechenfehler zurückführen, da die Implementation des Pendels exakt dieselbe ist und die Iterationsfrequenz der Bewegungsgleichungen auf 60 Hz festgelegt ist.

11 Glossar

11.1 Abkürzungen

API Application Programming Interface. 2, 3, 7, 13, 15–21, 23, 24, 31–34

ASSIMP Open Asset Import Library. 3, 22, 23

CI Continuous Integration. 22

CPU Central Processing Unit. 18, 24, 31

FPS Frames pro Sekunde. 21, 31, 33

gcc GNU Compiler Collection. 22

gdb GNU Debugger. 22

GLFW Graphics Library Framework. 16, 21–23, 29, 34

GLM OpenGL Mathematics. 6, 22, 23

GLSL OpenGL Shading Language. 17, 20

GUI Graphical User Interface. 22

IDE Integrated Development Environment. 22

IFFT Inverse Fast Fourier Transform. 25

LWJGL Lightweight Java Game Library. 13

NaN Not a Number. 30

OpenGL Open Graphics Library. 1, 2, 7, 13, 16–21, 23, 32–35, 42

OpenGL ES Open Graphics Library for Embedded Systems. 16

SDL2 Simple DirectMedia Layer 2.0. 22, 23

VCS Version Control System. 22

11.2 Technische Begriffe

Assembler

Assembler-Code bezeichnet eine low-level Programmiersprache, die ein Computer ohne weitere Übersetzung verstehen und ausführen kann. 13, 14

Compiler

Als Compiler wird ein programmiersprachspezifisches Tool bezeichnet, welches Quellcode in Assembler-Code übersetzen kann. 13, 14, 22, 35

Debian

Debian bezeichnet eine Unterfamilie von Linux-Computersystemen. Dazu gehören z.B. Ubuntu, Linux Mint, Raspian etc.... 22

Debugger

Als Debugger wird ein Tool bezeichnet, welches einem Programmierer hilft, Fehler (sog. engl. *bugs*, 'Ungeziefer') in seinem Quellcode zu finden und zu beheben. 22

Edge

Als Edge (engl. *edge*, 'Kante') werden Verbindungslinien zwischen zwei Vertices eines Modells bezeichnet. 3

Engine

Als eine Engine (engl. *engine*, 'Motor', 'Antrieb') im programmiertechnischen Sinne wird ein Konstrukt verstanden, welches als Grundgerüst eines Programms angeschaut werden kann. Bei 3D-Videospielen und Applikationen zum Beispiel handelt es sich hierbei um den Programmteil, welcher 3D- zu 2D-Transformationen, Licht-/Schatten-/Farbberechnungen und physikalische Berechnungen wie Wurftrajektorien etc. vornimmt. Auf diesem Grundgerüst kann dann mit kleinsten Änderungen von höchstens einigen Zeilen Code ein neues Modell, eine neue Lichtquelle oder ein neues Geschoss in die Welt gesetzt werden und die Engine berechnet automatisch die neuen Umgebungs-/Bewegungsverhältnisse. 1–3, 12, 13, 15, 21, 23–25, 33–36

Face

Ein Face (engl. *face*, 'Gesicht', 'Oberfläche', 'Fläche', pl. *faces*) bezeichnet die Fläche, welche entsteht wenn die Vertices zu den primitiven Formen (Dreieck, Rechteck) zusammengefügt und durch Linien verbunden werden. 3, 9, 10, 20, 36, 42

Fliesskommazahl

siehe **Float**. 15

Float

Ein Float oder eine Floating-Point-Zahl (engl. *to float*, 'schweben', 'gleiten') ist ein Datentyp, welcher Kommazahlen bis zu einer gewissen Genauigkeit speichern kann (meist 32-bit). In der deutschen Sprache werden Floats gerne als Fliess- oder Gleitkommazahlen bezeichnet. 1, 15, 24, 29–31

Floating-Point

siehe **Float**. 1, 2, 15, 25, 30, 31, 34, 36, 37

Fragment

Als Fragments (engl. *fragment*, 'Fragment', 'Bruchstück') werden Punkte auf einem Face eines Modells bezeichnet, welche durch die Rasterung des Faces in der Rasterization Stage entstehen. Die Frameents werden in den Fragment Shadern eingefärbt. 3, 10, 17, 31

Frame

Ein Frame bezeichnet ein gerendertes Bild eines Programmes. Durch die Angabe der FPS lassen sich Aussagen über die Leistung von Applikationen machen. 21, 24, 32, 33

Frametime

Die Grösse der Frametime bezeichnet die Leistung einer Applikation, und wird meist in Millisekunden angegeben. Sie entspricht dem Kehrwert der FPS. 21, 32, 33, 49–51

GitHub

GitHub (`https://github.com/`) ist eine Source-Code-Sharing-Plattform, basierend auf dem VCS *Git*. 2, 22, 42, 43

Linker

Als Linker (engl. *to link*, 'verknüpfen', 'anschliessen') wird ein Tool bezeichnet, welches C/C++-Bibliotheken und C/C++-Quellcode zu einer ausführbaren Datei zusammenfügt. 14

Linux

Linux oder GNU/Linux bezeichnet eine Familie von Betriebssystemen für Computersysteme, basierend auf dem Linux-Kernel von Entwickler Linus Torvalds. 1, 16, 21, 22, 32–34, 42, 50

Makefile

Makefiles sind Kompilierungsinstruktionen, um ein grösseres Projekt richtig zu kompilieren. 1

Multithreading

Als Multithreading wird eine Programmiertechnik bezeichnet, wobei mehrere Threads benutzt werden um Software parallelisiert auszuführen. 15, 18, 20, 23, 24, 36

Rendering

Als Rendering bezeichnet man ein Verfahren in der Computertechnik, bei welchem man durch Berechnungen audiovisuelle Ergebnisse produziert. 3, 20

Shader

Als Shader wird ein kleines Programm bezeichnet, welches parallelisiert auf Grafikkarten ausgeführt wird. Es ist in GLSL geschrieben, einer zu C sehr ähnlichen Sprache. Shader werden in Grafik-API's über die Grafik-Pipeline angesteuert und dienen der parallelen Computation stets derselben Berechnungen. 3, 7, 10, 17, 18, 20, 23, 24, 31

Texteditor

Als Texteditor wird ein Tool bezeichnet, mit welchem man Dateien bearbeiten kann. 14, 22

Thread

Als Thread (engl. *thread*, 'Faden', 'Strang') wird eine parallele Ausführungskette in einem Computerprogramm beschrieben. Durch Threads kann ein Programm an mehreren Stellen gleichzeitig ausgeführt werden. Threads sind zentrale Objekte in der parallelisierten Programmierung. 15, 18, 24

Tool

Als Tool (engl. *tool*, 'Werkzeug', 'Hilfsmittel') wird ein Stück Software bezeichnet, welches eine bestimmte Funktion/Aufgabe übernimmt. 12, 22

Tracking

Tracking bezeichnet Verfolgung bzw. Überwachung. 22

Tweak

Tweaks sind kleinere Anpassungen. 22

Vertex

Ein Vertex (engl. *vertex*, 'Ecke', 'Eckpunkt', pl. *vertices*) ist ein Punkt/Eckpunkt eines 3D-Modells. 3D-Modelle sind durch Vertices definiert und werden so respräsentiert. 3, 7, 10, 15, 17, 31

Vertices

siehe **Vertex**. 3, 6, 7, 15, 17, 33, 36

12 Quellenverzeichnis

Alexander Overvoorde. *Vulkan Tutorial - Introduction. (Englisch) [Vulkan Tutorial - Einführung].* Aufgerufen: 12.11.2019. URL: https://vulkan-tutorial.com/.

Apple Inc. *WPE - WebKit Port For Embedded Systems. (Englisch) [WPE - WebKit Port für eingebettete Systeme].* Aufgerufen: 10.11.2019. URL: https://webkit.org/wpe/.

Christine McKee. *CUDA Cores in Video Cards. (Englisch) [CUDA-Kerne in Grafikkarten].* Aufgerufen: 21.11.2019. URL: https://www.lifewire.com/what-is-nvidia-cuda-834095.

Eric W. Weisstein. *Double Pendulum. (Englisch) [Doppelpendel].* Aufgerufen: 16.11.2019. URL: http://scienceworld.wolfram.com/physics/DoublePendulum.html.

Erik Neumann. *Double Pendulum. (Englisch) [Doppelpendel].* Aufgerufen: 20.10.2019. URL: https://www.myphysicslab.com/pendulum/double-pendulum-en.html.

GLFW 3. *GLFW 3 Documentation. (Englisch) [GLFW 3 Dokumentation].* Aufgerufen: 23.10.2019. URL: https://www.glfw.org/docs/latest/group__input.html.

Joey de Vries. *Learn OpenGL - Basic Lighting. (Englisch) [Lerne OpenGL - Einfache Beleuchtung].* Aufgerufen: 13.11.2019. URL: https://learnopengl.com/Lighting/Basic-Lighting.

— *Learn OpenGL - Camera. (Englisch) [Lerne OpenGL - Kamera].* Aufgerufen: 24.10.2019. URL: https://learnopengl.com/Getting-started/Camera.

— *Learn OpenGL - Coordinate Systems. (Englisch) [Lerne OpenGL - Koordinatensysteme].* Aufgerufen: 24.10.2019. URL: https://learnopengl.com/Getting-started/Coordinate-Systems.

— *Learn OpenGL - Hello Triangle. (Englisch) (Lerne OpenGL - Hallo Dreieck).* Aufgerufen: 12.11.2019. URL: https://learnopengl.com/Getting-started/Hello-Triangle.

— *Learn OpenGL - Transformations. (Englisch) [Lerne OpenGL - Transformationen].* Aufgerufen: 24.10.2019. URL: https://learnopengl.com/Getting-started/Transformations.

learncpp.com/Alex. *4.8 — Floating point numbers. (Englisch) [4.8 — Fliesskommazahlen].* Aufgerufen: 24.11.2019. URL: https://www.learncpp.com/cpp-tutorial/floating-point-numbers/.

OpenGL Wiki. *Related toolkits and APIs. (Englisch) [Verwandte Toolkits und API's].* Aufgerufen: 10.11.2019. URL: https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs.

— *Rendering Pipeline Overview. (Englisch) [Rendering Pipeline Übersicht].* Aufgerufen: 11.11.2019. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

— *Tessellation.* Aufgerufen: 12.11.2019. URL: https://khronos.org/opengl/wiki/Tessellation.

Parewa Labs Pvt. Ltd. *Learn C++ Programming. (Englisch) [Lerne C++ Programmieren].* Aufgerufen: 29.10.2019. URL: https://www.programiz.com/cpp-programming.

Song Ho Ahn. *OpenGL Camera. (Englisch) [OpenGL Kamera].* Aufgerufen: 24.10.2019. URL: http://www.songho.ca/opengl/gl_camera.html.

spo-comm GmbH. *Was ist OpenGL?* Aufgerufen: 04.11.2019. URL: https://www.spo-comm.de/de/blognews/detail/article/News/detail/was-ist-opengl/.

Standard C++ Foundation. Aufgerufen: 29.10.2019. URL: https://isocpp.org/.

techpowerup.com. *NVIDIA GeForce GTX 1050 Ti.* Aufgerufen: 12.11.2019. URL: https://www.techpowerup.com/gpu-specs/geforce-gtx-1050-ti.c2885.

— *NVIDIA GeForce GTX 1080 Ti.* Aufgerufen: 12.11.2019. URL: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877.

The Khronos Group Inc. *OpenGL Overview. (Englisch) [OpenGL Überblick].* Aufgerufen: 04.11.2019. URL: https://www.opengl.org/about/.

— *Vulkan Overview. (Englisch) [Vulkan Übersicht].* Aufgerufen: 12.11.2019. URL: https://www.khronos.org/vulkan/.

The Khronos Vulkan Working Group. *Vulkan Specification - 2.6 Threading Behaviour. (Englisch) [Vulkan Spezifikation - 2.6 Threading-Verhalten].* Aufgerufen: 12.11.2019. URL: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#fundamentals-threadingbehavior.

— *Vulkan Specification - 36.1. Limit Requirements. (Englisch) [Vulkan Specifikation - 36.1. Limit-Anforderungen].* Aufgerufen: 03.11.2019. URL: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#limits-required.

The Khronos Vulkan Working Group. *Vulkan Specification - 9. Pipelines. (Englisch) [Vulkan Spezifikation - 9. Pipelines]*. Aufgerufen: 12.11.2019. URL: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#pipelines.

TIOBE Software BV. *TIOBE Index for October 2019. (Englisch) [TIOBE Index für Oktober 2019]*. Aufgerufen: 29.10.2019. URL: https://www.tiobe.com/tiobe-index/.

tutorialspoint.com. *Assembly. (Englisch) [Assembler]*. Aufgerufen: 24.11.2019. URL: https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm.

Universität Heidelberg. *Introduction to the C++ Language. (Englisch) [Einführung in die C++ Programmiersprache]*. Aufgerufen: 29.10.2019. URL: https://conan.iwr.uni-heidelberg.de/data/teaching/oopfsc_ss2017/structure.pdf.

Vangie Beal. *API - Application Program Interface. (Englisch) [API - Applikations-Programmierschnittstelle]*. Aufgerufen: 21.11.2019. URL: https://www.webopedia.com/TERM/A/API.html.

wikichip.org. *UHD Graphics 630 - Intel. (Englisch) [Intel UHD Graphics 630]*. Aufgerufen: 12.11.2019. URL: https://en.wikichip.org/wiki/intel/uhd_graphics/630.

13 Abbildungsverzeichnis

14 Tabellenverzeichnis

Anhang

A Quellcode der Applikationen

A.1 OGL

https://github.com/D3PSI/OGL

Abbildung 10: QR-Code: Direktlink zu GitHub-Repository von OGL

A.2 VK

https://github.com/D3PSI/VK

Abbildung 11: QR-Code: Direktlink zu GitHub-Repository von VK

B Flags und Flag-Bits Beispiele

B.1 Aus dem OpenGL-Standard

```
#define GL_DEPTH_BUFFER_BIT 0x00000100
#define GL_STENCIL_BUFFER_BIT 0x00000400
#define GL_COLOR_BUFFER_BIT 0x00004000
```

B.2 Aus dem Vulkan-Standard

```
typedef enum VkDebugUtilsMessageSeverityFlagBitsEXT {
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT = 0x00000001,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT = 0x00000010,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT = 0x00000100,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT = 0x00001000,
    VK_DEBUG_UTILS_MESSAGE_SEVERITY_FLAG_BITS_MAX_ENUM_EXT = 0x7FFFFFFF
} VkDebugUtilsMessageSeverityFlagBitsEXT;
typedef VkFlags VkDebugUtilsMessageSeverityFlagsEXT;
```

B.3 Arbeiten mit Flags

B.3.1 Flags setzen

```
VkDebugUtilsMessengerCreateInfoEXT debugUtilsMessengerInfo = {};
debugUtilsMessengerInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
debugUtilsMessengerInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT
                            | VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT
                            | VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
debugUtilsMessengerInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
                            | VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT
                            | VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
```

B.3.2 Flags herausfiltern

```
if (qF.queueCount > 0 &&
    (qF.queueFlags & VK_QUEUE_TRANSFER_BIT) &&   // Ist VK_QUEUE_TRANSFER_BIT gesetzt?
    !(qF.queueFlags & VK_QUEUE_GRAPHICS_BIT)) {  // Ist VK_QUEUE_GRAPHICS_BIT nicht gesetzt?

    family.transferFamilyIndex = i;

}
```

C Beispiele für Shader-Programme mit Phong-Lighting der beiden Engines

C.1 OGL

C.1.1 Vertex Shader

```
#version 330 core

layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 nor;
layout(location = 2) in vec2 tex;
layout(location = 3) in vec3 tan;
layout(location = 4) in vec3 bit;

out vec3 outPos;
out vec2 outTex;
out vec3 outNor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {

    gl_Position         = projection * view * model * vec4(pos, 1.0);
    outTex              = tex;
    outPos              = vec3(model * vec4(pos, 1.0));
    outNor              = mat3(transpose(inverse(model))) * nor;

}
```

C.1.2 Fragment Shader

```
#version 330 core

in vec3 outPos;
in vec2 outTex;
in vec3 outNor;

out vec4 FragColor;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;

uniform sampler2D texDiffuse1;

void main() {

    vec3 objectColor            = texture(texDiffuse1, outTex).xyz;

    // ambient lighting
    float ambientStrength       = 0.1;
    vec3 ambient                = ambientStrength * lightColor;
```

```
22
23      // diffuse lighting
24      vec3 norm                   = normalize(outNor);
25      vec3 lightDir               = normalize(lightPos - outPos);
26      float diff                  = max(dot(norm, lightDir), 0.0);
27      vec3 diffuse                = diff * lightColor;
28
29      // specular lighting
30      float specularStrength      = 0.5;
31      vec3 viewDir                = normalize(viewPos - outPos);
32      vec3 reflectDir             = reflect(-lightDir, norm);
33      float spec                  = pow(max(dot(viewDir, reflectDir), 0.0), 32);
34      vec3 specular               = specularStrength * spec * lightColor;
35
36      vec3 result                 = (ambient + diffuse + specular) * objectColor;
37      FragColor                   = vec4(result, 1.0);
38
39  }
```

C.2 VK

C.2.1 Vertex Shader

```
1   #version 450
2   #extension GL_ARB_separate_shader_objects : enable
3
4   layout(location = 0) in vec3 pos;
5   layout(location = 1) in vec3 nor;
6   layout(location = 2) in vec2 tex;
7   layout(location = 3) in vec3 tan;
8   layout(location = 4) in vec3 bit;
9
10  layout(binding = 0) uniform VPBuffer {
11
12      mat4 view;
13      mat4 proj;
14
15  } vp;
16
17  layout( push_constant ) uniform MBuffer {
18
19      mat4 model;
20
21  } m;
22
23  layout(location = 0) out vec3 outPos;
24  layout(location = 1) out vec2 outTex;
25  layout(location = 2) out vec3 outNor;
26
27  void main() {
28
29      gl_Position         = vp.proj * vp.view * m.model * vec4(pos, 1.0);
30      outTex              = tex;
31      outPos              = vec3(m.model * vec4(pos, 1.0));
```

```
    outNor              = mat3(transpose(inverse(m.model))) * nor;

}
```

C.2.2 Fragment Shader

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 outPos;
layout(location = 1) in vec2 outTex;
layout(location = 2) in vec3 outNor;

layout(binding = 1) uniform sampler2D diffSampler1;
layout(binding = 2) uniform sampler2D diffSampler2;

layout(binding = 3) uniform LightData {

    vec3 lightPos;
    vec3 viewPos;
    vec3 lightCol;

} ld;

layout(location = 0) out vec4 outColor;

void main() {

    vec3 objectColor            = texture(diffSampler1, outTex).xyz;

    // ambient lighting
    float ambientStrength       = 0.1;
    vec3 ambient                = ambientStrength * ld.lightCol;

    // diffuse lighting
    vec3 norm                   = normalize(outNor);
    vec3 lightDir               = normalize(ld.lightPos - outPos);
    float diff                  = max(dot(norm, lightDir), 0.0);
    vec3 diffuse                = diff * ld.lightCol;

    // specular lighting
    float specularStrength      = 0.5;
    vec3 viewDir                = normalize(ld.viewPos - outPos);
    vec3 reflectDir             = reflect(-lightDir, norm);
    float spec                  = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular               = specularStrength * spec * ld.lightCol;

    vec3 result                 = (ambient + diffuse + specular) * objectColor;
    outColor                    = vec4(result, 1.0);

}
```

D Initialisierung der Vulkan-Pipeline und des Pipeline Create Info-Structs

```
VkGraphicsPipelineCreateInfo graphicsPipelineCreateInfo = {};
graphicsPipelineCreateInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
graphicsPipelineCreateInfo.stageCount = 2;
graphicsPipelineCreateInfo.pStages = stages.stages.data();
graphicsPipelineCreateInfo.pVertexInputState = vertexInputStateCreateInfo_;
graphicsPipelineCreateInfo.pInputAssemblyState = inputAssemblyStateCreateInfo_;
graphicsPipelineCreateInfo.pViewportState = viewportStateCreateInfo_;
graphicsPipelineCreateInfo.pRasterizationState = rasterizationStateCreateInfo_;
graphicsPipelineCreateInfo.pMultisampleState = multisampleStateCreateInfo_;
graphicsPipelineCreateInfo.pDepthStencilState = depthStencilStateCreateInfo_;
graphicsPipelineCreateInfo.pColorBlendState = colorBlendStateCreateInfo_;
graphicsPipelineCreateInfo.pDynamicState = dynamicStateCreateInfo_;
graphicsPipelineCreateInfo.layout = pipelineLayout;
graphicsPipelineCreateInfo.renderPass = renderPass_;
graphicsPipelineCreateInfo.subpass = 0;
graphicsPipelineCreateInfo.basePipelineHandle = VK_NULL_HANDLE;
graphicsPipelineCreateInfo.basePipelineIndex = -1;

result = vkCreateGraphicsPipelines(
    vk::core::logicalDevice,        // Logical Device (GPU)
    VK_NULL_HANDLE,
    1,
    &graphicsPipelineCreateInfo,    // Create Info Struct
    vk::core::allocator,            // Allocator
    &pipeline                       // Handle der Pipeline (VkPipeline-Handle)
    );
ASSERT(result, "Failed to create graphics pipeline", VK_SC_GRAPHICS_PIPELINE_CREATION_ERROR);
```

E Rohdaten

Tabelle 5: Rohdaten unter Windows 10 von VK

Windows 10/VK			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	0.537750	3.190810	-
20	0.531887	2.610285	-
30	0.536021	4.814636	-
40	0.530082	2.464268	-
50	0.523560	4.723666	-
60	0.532283	2.835271	-
70	0.534845	4.863813	-
80	0.527482	2.379819	-
90	0.538416	3.869969	-
100	0.542652	3.081664	-

Tabelle 6: Rohdaten unter Windows 10 von OGL

Windows 10/OGL			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	0.170730	1.497230	1.200048
20	0.191861	1.408054	1.126634
30	0.194738	1.398210	1.166453
40	0.161384	1.375138	1.112718
50	0.161888	1.345714	1.144427
60	0.160779	1.344447	1.222046
70	0.162496	1.357036	1.121202
80	0.161970	1.339226	1.134173
90	0.163305	1.358327	1.111111
100	0.166044	1.344809	1.137915

Tabelle 7: Rohdaten unter Linux Mint 19.02 von VK

Linux Mint 19.02/VK			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	16.234079	16.823478	-
20	16.666667	16.666667	-
30	16.666667	16.666667	-
40	16.666667	16.666667	-
50	16.666667	16.666667	-
60	16.666667	16.666667	-
70	16.666667	16.666667	-
80	16.666667	16.666667	-
90	16.666667	16.666667	-
100	16.666667	16.666667	-

Tabelle 8: Rohdaten unter Linux Mint 19.02 von OGL

Linux Mint 19.02/OGL			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	16.342098	16.234960	16.321894
20	16.666667	16.666667	16.666667
30	16.666667	16.666667	16.666667
40	16.666667	16.666667	16.666667
50	16.666667	16.666667	16.666667
60	16.666667	16.666667	16.666667
70	16.666667	16.666667	16.666667
80	16.666667	16.666667	16.666667
90	16.666667	16.666667	16.666667
100	16.666667	16.666667	16.666667

Tabelle 9: Rohdaten unter Ubuntu 18.04 von VK

Ubuntu 18.04/VK			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	16.474465	16.798634	-
20	16.666667	16.666667	-
30	16.666667	16.666667	-
40	16.666667	16.666667	-
50	16.666667	16.666667	-
60	16.666667	16.666667	-
70	16.666667	16.666667	-
80	16.666667	16.666667	-
90	16.666667	16.666667	-
100	16.666667	16.666667	-

Tabelle 10: Rohdaten unter Ubuntu 18.04 von OGL

Ubuntu 18.04/OGL			
Grafikkarte/Grafikchip	NVIDIA GTX 1080 Ti	NVIDIA GTX 1050 Ti	Intel UHD Graphics 630
Laufzeit [s]	Frametime [ms]		
10	16.583748	16.876545	16.234903
20	16.666667	16.666667	16.666667
30	16.666667	16.666667	16.666667
40	16.666667	16.666667	16.666667
50	16.666667	16.666667	16.666667
60	16.666667	16.666667	16.666667
70	16.666667	16.666667	16.666667
80	16.666667	16.666667	16.666667
90	16.666667	16.666667	16.666667
100	16.666667	16.666667	16.666667