HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Studiengang Angewandte Informatik
Fakultät Elektrotechnik und Informatik

Performancevergleich eines Partikelsystems umgesetzt mit OpenGL und Vulkan

Informatikprojekt

Verfasser: Niklas Birk

Matrikelnummer: 28709

Datum: 1. Apr 2020

Betreuer: Prof. Dr. Daniel Scherzer

Zusammenfassung

Auf der Grafikkarte können gut SIMD-Verarbeitungen durchgeführt werden. Ein Partikelsystem eignet sich gut als Beispiel hierfür. Um eine Grafikkarte anzusprechen werden allerdings Grafik-APIs benötigt. *OpenGL* und *Vulkan* sind solche APIs. In dieser Arbeit wird ein Performance- und Umsetzungsvergleich zwischen diesen beiden APIs anhand eines Partikelsystems durchgeführt. Für diesen Vergleich ist ein Partikelsystem implementiert worden, dass zum *OpenGL* und zum anderen *Vulkan* nutzt, um die Simulation und das Rendering auf der Grafikkarte auszuführen. Der Umsetzungsaufwand wurde hauptsächlich durch die Anzahl der *Lines of Code* (*LOC*) gemessen. Die Performance wurde durch die Ausführungszeit der Renderschleife bestimmt. Hierbei kam jeweils heraus, dass *Vulkan* einen deutlich höheren Umsetzungsaufwand als *OpenGL*, jedoch auch bei dem Partikelsystem eine 25% bessere Performance erzielt hat.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommene Gedanken sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Tettnang, 01.04.2020, N. Bik

(Ort, Datum, Unterschrift)

Inhaltsverzeichnis

Zusammenfassung..... I
Eidesstattliche Erklärung.....II
Inhaltsverzeichnis..... III
Abbildungsverzeichnis..... IV
Tabellenverzeichnis..... V
1 Einführung.....1
1.1 Motivation und Problemstellung.....1
1.2 Zielsetzung.....1
1.3 Abgrenzung.....2
2 Grundlagen.....3
2.1 Partikelsysteme.....3
2.1.1 Arten von Partikelsystemen.....5
2.1.2 Partikel und Emitter.....8
2.2 Grafik APIs.....10
2.2.1 OpenGL.....12
2.2.2 Vulkan.....14
3 Entwurf und Implementierung.....17
3.1 Partikelsystem.....17
3.2 Implementierungsdetails.....18
3.2.1 CPU Variante.....20
3.2.2 OpenGL Variante.....22
3.2.3 Vulkan Variante.....26
4 Performancevergleich und Aufwand.....33
4.1 Aufwand.....33
4.1.1 Benötigte Zeit.....33
4.1.2 Lines of Code.....34
4.2 Performance.....38
4.2.1 Erste Messung.....39
4.2.2 Zweite Messung.....40
4.2.3 Dritte Messung.....41
4.2.4 Messergebnisse.....42
5 Fazit und Ausblick.....43
5.1 Fazit.....43
5.2 Ausblick.....43
Literaturverzeichnis..... V

Abbildungsverzeichnis

Schaubild 2.1.1: *Ablauf* eines Partikelsystems nach....4
Schaubild 2.1.2: Eine kugelförmige Punktwolke aus *[Reeves 1983, S. 5]*....5
Schaubild 2.1.1.1: Einteilung von Partikelsystemen aus [Orlamünder & Mascolus 2004]6
Schaubild 2.1.1.2: Datenfluss in einem zustandslosen System nach [Pätzold 2005, S.6f] 7
Schaubild 2.1.1.3: Datenfluss in einem zustandserhaltenden System nach [Pätzold 2005, S.7]....8
Schaubild 2.1.2.1: Partikelsystem als Baumdiagramm....9
Schaubild 2.2.1: Grober Aufbau einer Grafikkarte....10
Schaubild 2.2.2: Einige bekannte Grafik-APIs....11
Schaubild 2.2.3: Grafikpipeline....11
Schaubild 2.2.1.1: Die Grafikpipeline von OpenGL [Segal & Akeley 2019, S. 35]....12
Schaubild 2.2.2.1: Die Grafikpipeline von Vulkan [VULKANSPEC]....14
Schaubild 2.2.2.2: Die Vulkan-API im Detail mit zugehörigen Objekten....15
Schaubild 3.1.1: Das struct particle....17
Schaubild 3.1.2: Das struct emitter und particle_system....18
Schaubild 3.2.1: Softwarekomponenten....18
Schaubild 3.2.1.1: cpuMain Render Loop....20
Schaubild 3.2.1.2: Die CPU Variante mit 10.000 Partikeln....21
Schaubild 3.2.2.1: Vertex-Shader der OpenGL-Variante....22
Schaubild 3.2.2.2: Fragment-Shader der OpenGL-Variante....22
Schaubild 3.2.2.3: Input-Variablen für den Compute-Shader....23
Schaubild 3.2.2.4: Die main() des Compute-Shader für OpenGL....24
Schaubild 3.2.2.5: Die Renderschleife von der OpenGL Variante....25
Schaubild 3.2.2.6: Die OpenGL Variante mit 10.000.000 Partikeln....26
Schaubild 3.2.3.1: Das struct Compute der Vulkan-Variante....27
Schaubild 3.2.3.2: Das struct Graphics der Vulkan-Variante....28
Schaubild 3.2.3.3: Weitere Vulkan-Funktionen der Vulkan-Variante....29
Schaubild 3.2.3.4: Die Renderschleife der Vulkan-Variante....30
Schaubild 3.2.3.5: Aufzeichnen des Compute-Command-Buffers....31
Schaubild 3.2.3.6: Die Vulkan-Variante mit 10.000.000 Partikeln....32
Schaubild 4.1.2.1: Beispielcode mit Zeilenangaben....34
Schaubild 4.1.2.2: Balkendiagramm mit den LOC zum visuellen Vergleich....36
Schaubild 4.1.2.3: Anteile SLOC am Gesamtprojekt....37

Schaubild 4.1.2.4: Anteil SLOC von Vulkan im direkten Vergleich zu OpenGL.....37
Schaubild 4.2.1.1: Mittlere Frametimes der CPU-Variante bei 100.000 Partikeln.....39
Schaubild 4.2.1.2: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 100.000 Partikeln..... 40
Schaubild 4.2.2.1: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 1.000.000 Partikeln..... 41
Schaubild 4.2.3.1: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 10.000.000 Partikeln.....42

Tabellenverzeichnis

Tabelle 4.1.2.1: LOC-Tabelle des Beispielquellcodes.....35
Tabelle 4.1.2.2: LOC-Tabelle der einzelnen Projektdateien.....35
Tabelle 4.1.2.3: LOC nach Variante *bzw.* Modul zusammengefasst.....36
Tabelle 4.2.4.1: Die Messergebnisse zusammengefasst.....42

1 Einführung

1.1 Motivation und Problemstellung

In modernen Grafikanwendungen, wie Spiele oder Filme, werden vermehrt Partikelsysteme verwendet, um etwa Feuer, Nebel, Rauch oder andere natürliche Phänomene zu visualisieren [CONITEC].
Entsprechende Effekte werden dadurch erzielt, dass viele kleinen Teilchen, *Partikel* genannt, animiert, koloriert, bei Bedarf texturiert und anschließend visualisiert werden.
Durch die vielen kleinen Teilchen sieht es für den Betrachter aus wie ein zusammenhängendes System, das auch mit der Umwelt interagieren kann, sofern entsprechende Physik implementiert ist.
Die Berechnung und Speicherung ist je nach Anzahl der vielen Partikel sehr aufwändig.
Um die *CPU* zu entlasten bietet es sich an, die Berechnungen von der *GPU* ausführen zu lassen, da diese sich aufgrund der Architektur besser eignet.
Als Schnittstelle für den Programmierer zur Grafikeinheit des Computers kommen dabei *Grafik-APIs*, wie *OpenGL* oder *Vulkan*, zum Einsatz.

OpenGL, wurde diese API 1992 erstveröffentlicht, ist der wesentlich neueren Schnittstelle *Vulkan*, 2016, dahingehend unterlegen, dass die Performance u.a. durch weniger *Treiber-Overhead* und auch die Möglichkeit mehrere *Threads* zu verwenden gestiegen ist [VULKAN].
Moderne Grafikkarten unterstützen nach wie vor beide APIs. Doch es stellt sich nun die Frage ob die Performance bei einem Partikelsystem mit Einsatz von *Vulkan* oder gegenüber *OpenGL* ebenfalls steigt und wie hoch der Aufwand der jeweiligen Umsetzung ist.

1.2 Zielsetzung

Diese Arbeit beschäftigt sich mit Partikelsystemen. Neben einem kleinen Abriss, wie William T. Reeves diese 1983 eingeführt hat, wird primär behandelt, wie gut sich ein Partikelsystem in OpenGL und Vulkan umsetzen lassen. Dabei findet nicht nur das *Rendering* mit den APIs statt, sondern auch die Simulation.Außerdem wird der

Implementierungsaufwand begutachtet und auch die Performance bei der Laufzeit wird untersucht.

1.3 Abgrenzung

Es gibt viele Möglichkeiten ein Partikelsystem zu konfigurieren und zu beeinflussen. Außerdem ist das Erstellen von möglichst imposanten Effekten möglich. Dies ist jedoch nicht Teil dieser Arbeit. In dieser Arbeit geht es ausschließlich darum einen Performancevergleich zwischen *OpenGL* und *Vulkan* zu erstellen, dazu muss das verwendete Partikelsystem nicht schön und auch nicht aufwändig sein.

2 Grundlagen

2.1 Partikelsysteme

Partikelsysteme eignen sich besonders gut für dynamische Objekte und natürliche Phänomene, wie Rauch, Wasser oder Feuer [Reeves 1983, S.1]. Aber auch Haare, Laub und Gräser können mit Partikelsysteme simuliert und dargestellt werden.
Alle genannten Bespiele gemein haben, dass diese meist aus kleinen Teilen, sogenannte Partikel, bestehen, die dann animiert und visualisiert werden.
Zur Anwendung kommen Partikelsysteme in Filmen und Videospielen, aber auch in wissenschaftlichen Simulationen.

Partikel und dynamische Objekte sind schon länger bekannt, doch den Begriff Partikelsystem prägte William T. Reeves.
William T. Reeves war 1982 bei *Lucasfilm Ltd.* angestellt und hat dort am Film *Star Trek II: Der Zorn des Khan* mitgearbeitet. In diesem Film war auch der erste Einsatz von Partikelsystemen in der Filmindustrie. Im Juli1983 veröffentlichte er das Paper [Reeves 1983], in diesem er Partikelsysteme formal einführte und die Anwendung im oben genannten Film thematisiert.

Reeves beschreibt Partikelsysteme wie folgt:

> *„A particle system is a collection of many minute particles that together represent a fuzzy object." [Reeves 1983, S. 2]*

Als unscharfe Objekte sieht er Phänomene wie Wolken, Rauch, Wasser und Feuer. Diese Objekte sind mit herkömmlichen Techniken schwierig zu modellieren. Dies liegt daran, dass die Oberflächen der Objekte oder gar die Objekte selbst komplex und irregulär sind.
Partikel sind sehr dynamisch, denn über Zeit werden sie erschaffen, ändern und bewegen sich und sterben [Reeves 1983, S. 1f].
In [Reeves 1983, S. 3] wird außerdem der Fluss von Partikel bzw. eines Partikelsystems beschrieben. Dieser Fluss ist in Schaubild 2.1.1 unten veranschaulicht.

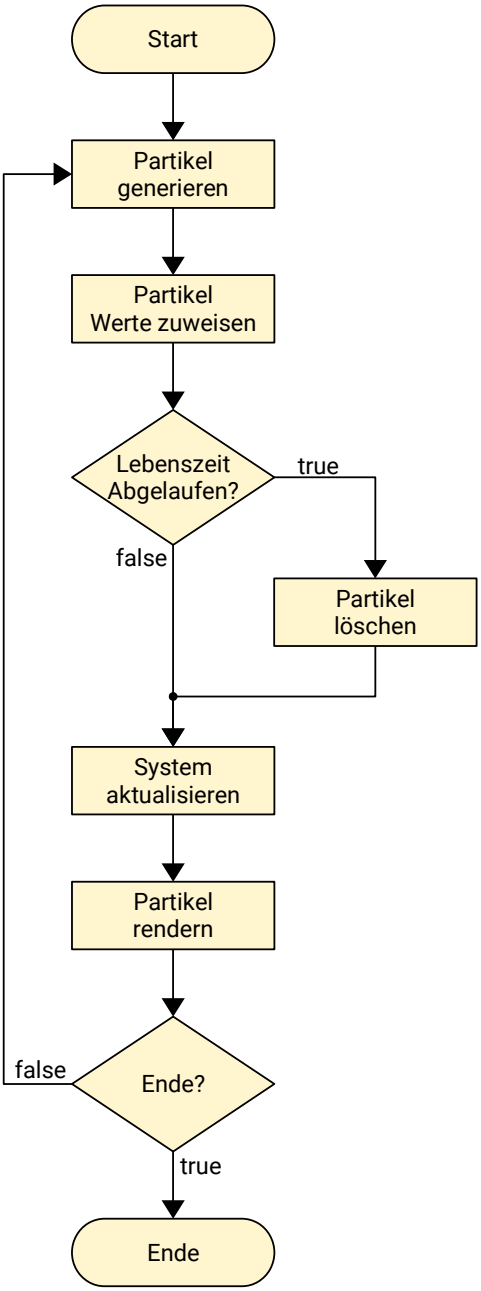

Schaubild 2.1.1: Ablauf eines Partikelsystems nach

Statt also die herkömmlichen Techniken zu verwenden und Objekte als eine Oberfläche von Polygonen zu modellieren verwenden Partikelsysteme sehr viele kleine primitive Partikel, die in ihrer Gesamtheit das Volumen ausbilden. Partikel selbst können als Punkt in einem drei-dimensionalen Raum gesehen werden. Im Extremfall sind diese

Partikel dann direkt auf Pixel abgebildet. Sehr viele dieser Partikel werden als Partikelwolken zusammengefasst und bilden so das Volumen des darzustellenden Objektes aus. In Abbildung 2.1.2 ist eine Punktwolke abgebildet, die eine Kugel bildet [Reeves 1983, S. 5]

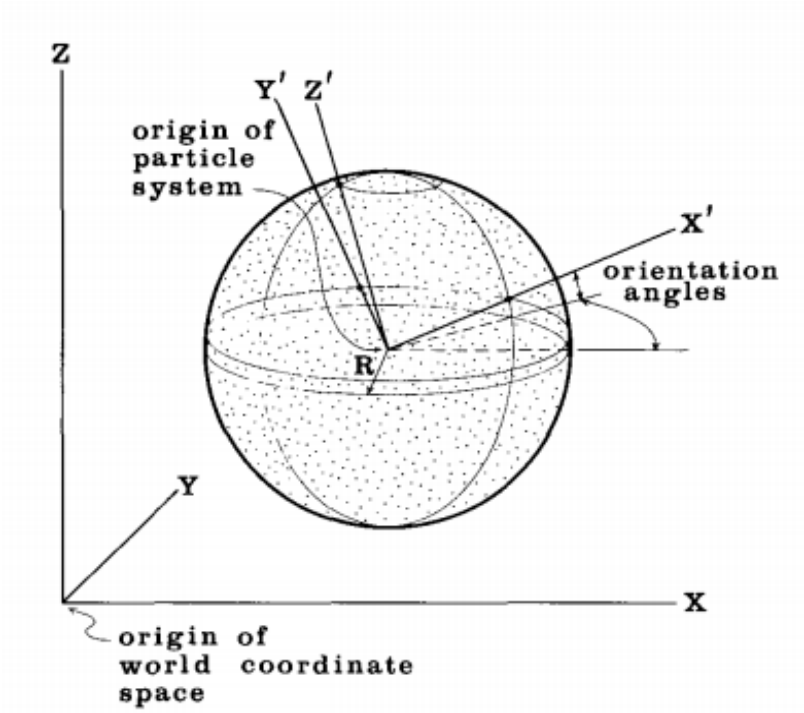

Schaubild 2.1.2: Eine kugelförmige Punktwolke aus [Reeves 1983, S. 5]

2.1.1 Arten von Partikelsystemen

Mittlerweile ist die Theorie von Partikelsystemen vorangeschritten und es haben sich verschiedene Arten von Partikelsysteme herausgebildet. In Schaubild 2.1.1.1 ist eine Einteilung zu sehen, wie sie in [Orlamünder & Mascolus 2004] beschrieben ist.

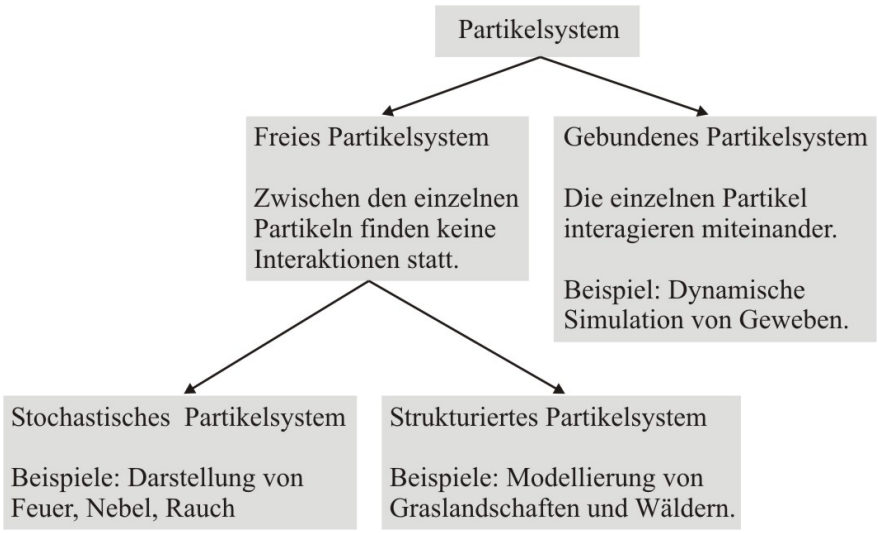

Schaubild 2.1.1.1: Einteilung von Partikelsystemen aus [Orlamünder & Mascolus 2004]

Bereits [Reeves 1983, S. 13fff] gibt neben den stochastischen Partikelsystemen weitere Anwendung an, die weniger auf Zufall, sondern mehr auf vorbestimmte Ereignisse setzen. Dazu zählt Feuerwerk, bei dem Streifen überwiegen, statt z.B. Punkte. Außerdem explodieren Feuerwerkskörper kontrollierter, sodass Formen oder Muster entstehen, anstatt zufällig wild lodernde Flammen. Des weiteren beschreibt er Gras, bei dem nur Teile eine zufällige Richtung haben, aber nicht das gesamte Partikelsystem zufällig berechnet werden.

Neben dieser anwendungs- und verhaltensbezogenen Einteilung lassen sich Partikelsysteme auch in zwei sehr grundlegende Unterscheidungen einteilen: den zustandslosen und zustandserhaltenden Systemen [Pätzold 2005, S.6f].
Bei den zustandslosen Partikelsystemen werden die im Aktualisierungsschritt berechneten Attribute nicht wieder in den Partikel zurückgeschrieben, sondern verworfen. Das bedeutet, dass die initialen Attribute eines Partikels konstant sind und aus diesen bei jeder Aktualisierung die Momentan-werte des Partikels berechnet wird.
Für solche Systeme verwendet man eine Funktion, die für die gesamte Zeitspanne zuständig ist. In [Pätzold 2005, S.6f] wird beispielhaft folgende Funktion angegeben, die für die Positionsberechnung verwendet werden kann:

$$\vec{p} = \overrightarrow{p_{init}} + \overrightarrow{v_{init}} t + \frac{1}{2}\vec{a}\, t^2$$

mit $\vec{p}$:= atuelle Position des Partikels

$\overrightarrow{p_{init}}$:= initiale Position des Partikels

$\overrightarrow{v_{init}}$:= initiale Geschwindigkeit des Partikels

$\vec{a}$:= atuelle Beschleunigungskraft

t := globale Zeit

Somit lässt sich auch der Datenfluss in zustandslosen Partikelystemen wie in Schaubild 2.1.1.2 visualisieren. Im Schaubild erkennt man die unidirektionale Beziehung zwischen den Partikelattributen und der Berechnung der aktuellen Werte. Ein Vorteil dieser Variante ist, dass durch das Fehlende Zurückschreiben weniger Speicherzugriffe nötig sind. Ein Nachteil ist allerdings, dass Kollisionen mit anderen Partikeln oder andere lokale Kräfte nur sehr schwer oder gar nicht möglich sind.

Schaubild 2.1.1.2: Datenfluss in einem zustandslosen System nach [Pätzold 2005, S.6f]

Den zustandslosen Partikelsystemen gegenüber stehen die zustandserhaltenden Partikelsysteme. Bei dieser Art von Partikelsystemen werden die berechneten Ergebnisse wieder dem Partikel zugeführt und dieses erhält die Werte, bis diese im nächsten Durchlauf wieder aktualisiert werden. Auch hierfür hält [Pätzold 2005, S.7] eine beispielhafte Formel für die Positionsberechnung bereit:

$$\overrightarrow{p_{new}} = \overrightarrow{p_{old}} + \vec{v}\,dt + \frac{1}{2}\vec{a}(dt)^2$$

mit $\overrightarrow{p_{new}} :=$ neue Position des Partikels

$\overrightarrow{p_{old}} :=$ alte Position des Partikels

$\vec{v} :=$ atuelle Geschwindigkeit des Partikels

$\vec{a} :=$ atuelle Beschleunigungskraft

$dt :=$ Zeitdifferenz seit letzter Aktualisierung

Der Datenfluss in Schaubild 2.1.1.3 enthält nun eine bidirektionalen Verbindung zwischen den Attributen und der Berechnung. Anders als bei zustandslosen Partikelsystemen kann hier auf Kräfte, die seit der letzten Aktualisierung einwirken, besser reagiert werden.

Schaubild 2.1.1.3: Datenfluss in einem zustandserhaltenden System nach [Pätzold 2005, S.7]

2.1.2 Partikel und Emitter

Ein Partikel ist die kleinste Einheit in einem Partikelsystem. Partikel besitzen bestimmte Eigenschaften bzw. Attribute, die je nach Anwendungsfall verschieden sein können, zumeist sind es aber folgende:

i. Position
ii. Geschwindigkeit (Schnelligkeit und Richtung)
iii. Farbe
iv. Lebensspanne
v. …

Die Position ist eine Vektorgröße und betrifft meist den 3D-Raum. Als Lebensspanne können verschiedene Parameter dienen. Eine Variante ist die Anzahl an Frames, die das Partikel leben soll [Reeves 1983, S.6]. Pro Frame wird die Lebensspanne dekrementiert und wenn diese bei null ist, dann stirbt der Partikel. Wobei sterben in diesem Fall bedeutet, dass der Speicher freigegeben wird. Eine Alternative dazu wäre, dass der Partikel nicht wirklich gelöscht, sondern lediglich auf Standardwerte zurückgesetzt wird, bzw. der reservierte Speicherbereich weiterverwendet wird. Speicher zu reservieren ist eine zeit-kostenintensive Operation und bei vielen Partikeln kann somit Zeit eingespart werden. Als alternativer Parameter für die Lebensspanne kann z.B. auch die Farbe dienen. Wenn der Partikel einen gewissen Schwellenwert bei der Farbe erreicht, dann stirbt der Partikel.

Partikelsysteme bestehen üblicherweise aus Emitter, welche dann einzelne Partikel verwalten und ausstoßen. Schaubild 2.1.2.1 zeigt die Struktur eines Partikelsystems als Baum. Darin ist ersichtlich, dass ein Partikelsystem auch aus mehreren Emittern bestehen kann und diese dann mehrere Partikel verwalten.

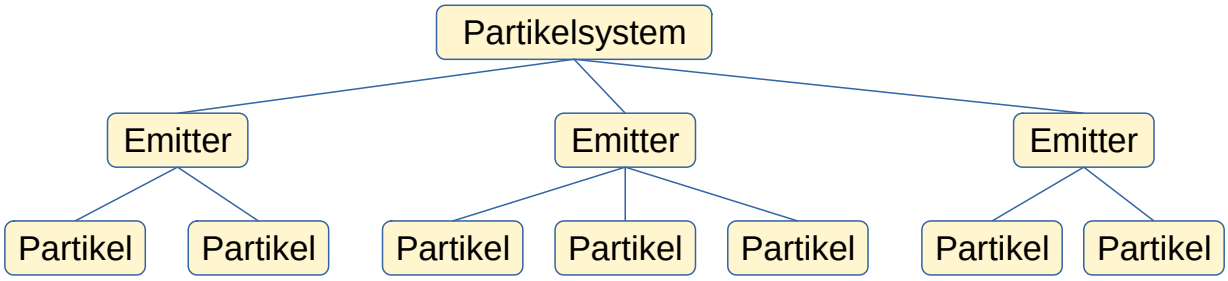

Schaubild 2.1.2.1: Partikelsystem als Baumdiagramm

Ein Emitter kann ein Punkt, eine Linie oder eine Oberfläche sein. Innerhalb des Emitters werden dann neue Partikel geboren und initialisiert. Bei einer Linie oder Oberfläche

können die Partikel an verschiedenen Positionen auf dem Emitter ausgestoßen werden. Wie viele Partikel wo ausgestoßen werden, kann ebenfalls der Emitter festlegen.

2.2 Grafik APIs

APIs (*application programming interface)* sind Programmierschnittstellen zu einer Software. Ein Entwickler kann durch diese Schnittstelle Funktionalitäten einer Software ansprechen oder selbst anderen Entwicklern zur Verfügung stellen. Moderne Grafikanwendungen, vor allem Spiele, benötigen oft viel Rechenleistung, weshalb Grafikprozessoren (*GPU*) (weiter-)entwickelt wurden, die heute in der Regel auf Grafikkarten zu finden sind [Brodtkorb u.a. 2013, S. 2f]. Grafikkarten bieten neben der *GPU* auch einen eigenen Hauptspeicher, den Grafikspeicher (*Video RAM*). Schaubild 2.2.1 zeigt einen exemplarischen Aufbau einer Grafikkarte. Eine Grafikkarte nimmt über den Anschluss am Mainboard Befehle und Daten entgegen. Die Daten können entweder direkt von der *GPU* genutzt werden oder werden in den Grafikspeicher geladen.

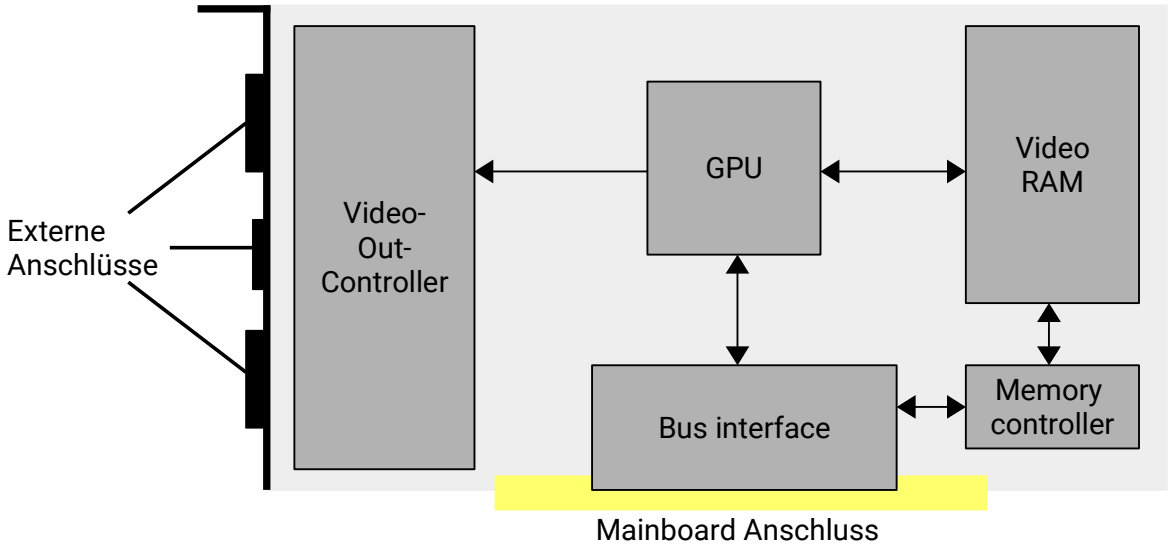

Schaubild 2.2.1: Grober Aufbau einer Grafikkarte

Die Software-Schnittstelle zu einer Grafikkarte bzw. für Grafikberechnungen wird Grafik-API genannt. Die Grafik-API legt hierbei grundlegende Schritte fest, wie ein Bild in einen *Framebuffer* gerendert wird. Die API wird von den Grafikchip- bzw. Grafikkartenerstellern

implementiert und Entwickler können diese dann für ihre Grafikanwendungen nutzen. In Schaubild sind einige bekannte Grafik-APIs zu finden.

Schaubild 2.2.2: Einige bekannte Grafik-APIs

In der Computergrafik gibt es das Modell einer *Grafikpipeline*. Die Grafikpipeline ist der Weg, den die Daten von der Anwendung bis zum fertigen Bild durchlaufen. Schaubild 2.2.3 zeigt eine Grafikpipeline mit den wichtigsten Schritten.

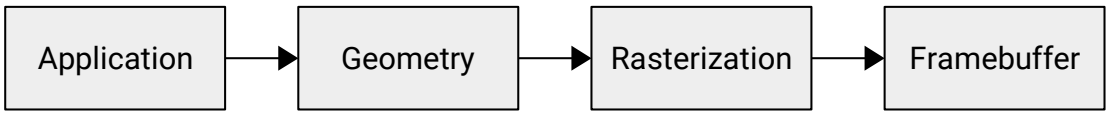

Schaubild 2.2.3: Grafikpipeline

Solch eine Pipeline ist an bestimmten Punkten programmierbar und andere sind fest vorgegeben. Die programmierbaren Teile der Grafikpipeline werden durch sogenannte *Shader* programmiert. *Shader* sind kleine Programme, die von der *GPU* ausgeführt

werden. Am Ende steht der *Framebuffer*, der das fertige Bild enthält, das dann z.B. auf einem Ausgabegerät, wie einen Bildschirm, angezeigt werden kann.

2.2.1 OpenGL

In der Spezifikation zu *OpenGL* wird die API beschrieben und festgelegt, welche Funktionen implementiert werden und wie diese reagieren müssen. Die Implementierung setzen dann die Grafikchip- bzw. Grafikkartenersteller um.

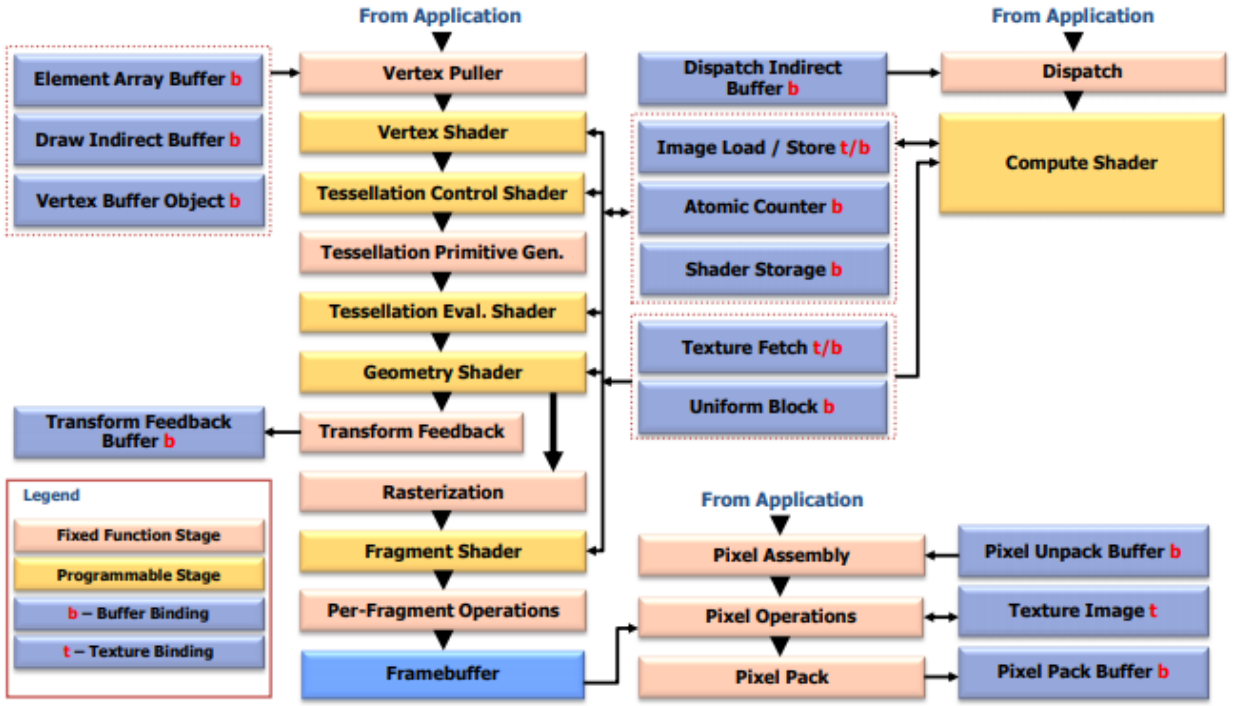

Schaubild 2.2.1.1: Die Grafikpipeline von OpenGL [Segal & Akeley 2019, S. 35]

Schaubild 2.2.1.1 zeigt die Grafikpipeline von *OpenGL* im Detail. Die wichtigsten Schritte werden im folgenden kurz beschrieben.

Im *Vertex Shader* Schritt könne *Vertices* durch *Vertex Shader* verändert werden. Insbesondere Translationen, Rotationen und Skalierungen werden in diesem Schritt vorgenommen. *Vertices* sind Punkte eines Objektes. Für jeden Vertex wird ein *Vertex Shader* aufgerufen.

Während im *Vertex Shader* nur vorhandene Punkte verändert werden können, können bei einem *Geometry Shader* Punkte bzw. Geometrien entfernt oder hinzugefügt werden. Nachdem die Eckpunkte von Objekten verarbeitet wurden, werden diese im *Rasterization* Schritt gerastert. Ein Rasterpunkt wird Fragment genannt und entspricht einem Pixel vom fertigen Bild. Durch *Fragment Shader* kann jeder Fragment koloriert oder texturiert werden. Am Ende der Pipeline steht dann der Framebuffer mit dem gerenderten Bild.
Grafikprozessoren nutzen *SIMD*-Verarbeitung (*Single Instruction Multiple Data*). Bspw. werden für alle Fragmente die gleichen Instruktionen ausgeführt. Das bedeutet eine Instruktion wird auf vielen Daten angewendet. Es gibt nicht-grafische Anwendungsfälle, die davon profitieren können. Bspw. kann die physikalische Simulation vieler kleiner Objekte, wie Partikel, als guter Anwendungsfall für *SIMD*-Verarbeitung angesehen werden. Während die Grafikpipeline für das Rendern von Bildern vorgesehen ist, gibt es die *Compute Shader*, die Teil der *Compute Pipeline* sind. Die *Compute Pipeline* dient nicht dazu ein Bild zu rendern, sondern dazu, um Berechnungen mit *SIMD*-Verarbeitung zu durchzuführen. Statt also einen *Framebuffer* zu befüllen, werden allgemeinere Berechnungsaufgaben damit umgesetzt. Für das Nutzen des Grafikprozessors für allgemeine Aufgaben, statt Bilder rendern, gibt es den Begriff *General Purpose GPU* (*GPGPU*). Die Simulation der Partikel ist eine solche Aufgabe. Die Ergebnisse des *Compute-Shader*s können dann allerdings als Eingabe für die Grafikpipeline verwendet werden.

OpenGL selbst ist als Zustandsmaschine implementiert. Das bedeutet vor jedem Rendern muss ein gewünschter Zustand gesetzt werden. Wenn dieser Zustand gesetzt ist, dann kann die Grafikpipeline gestartet und ein Bild gerendert werden. So ein Zustand besteht aus vielen Teilen. Ein Teil sind die für diesen Durchlauf verwendeten *Shaderprogramme* und Daten.
Für die *Shader*-Programmierung kommt eine *C* ähnliche Sprache namens *GLSL* (*OpenGL Shading Language*) zum Einsatz.

2.2.2 Vulkan

Vulkan ist eine neuere Grafik-API und ebenso wie *OpenGL* von der *Khronos Group* entwickelt. Die *Khronos Group* ist ein Zusammenschluss von Mitgliedern, wie z.B. AMD und Nvidia, um offene Standards im Multimediabereich zu entwickeln [KHRONOS].

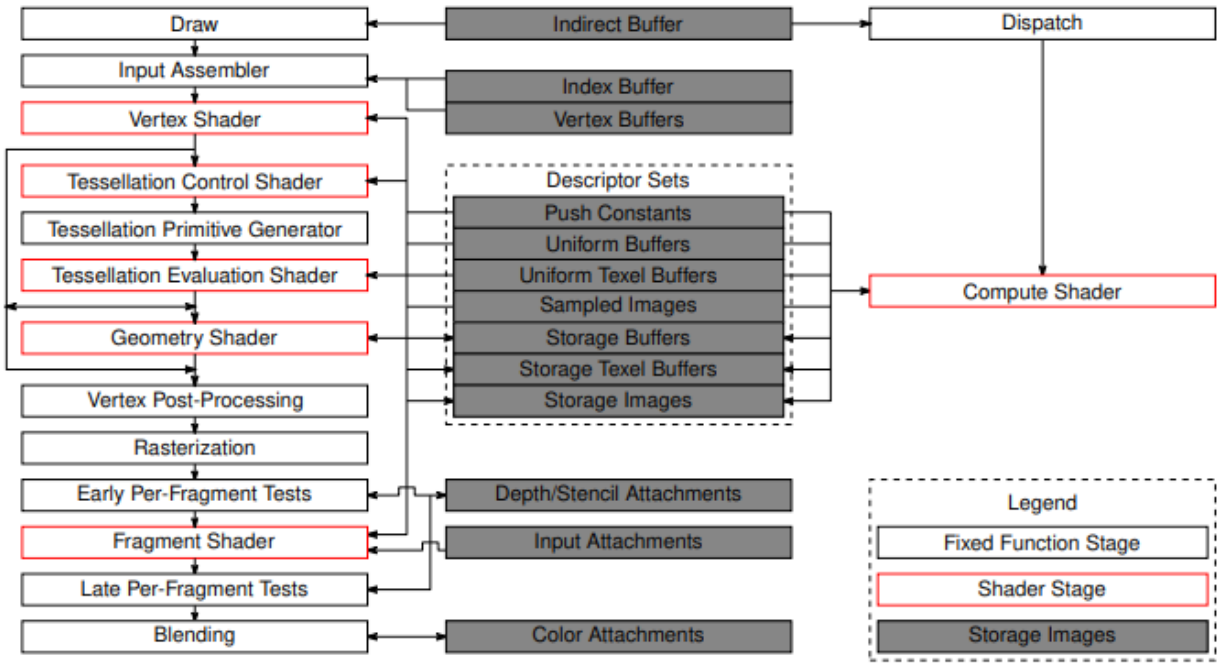

Schaubild 2.2.2.1: Die Grafikpipeline von Vulkan [VULKANSPEC]

Schaubild 2.2.2.1 zeigt die Grafikpipeline von *Vulkan*. An den wesentlichen Schritten hat sich zu *OpenGL* nichts geändert. Die *Shader* müssen für Vulkan allerdings im *SPIR-V* Format vorliegen. *SPIR-V* ist eine *intermediate language*. Das bedeutet, für Sprachen, die einen Compiler für *SPIR-V* bieten, lassen sich *Shader* für Vulkan schreiben. Ein *Shader* lässt sich somit in *GLSL* von *OpenGL* oder in HLSL für *DirectX* schreiben und anschließend in *SPIR-V* übersetzen. Das macht die *Shader*-Programmierung bei *Vulkan* unabhängiger von der genutzten Sprache.

Anders als *OpenGL* ist Vulkan nicht als Zustandsmaschine implementiert. Dies bietet u.a. den Vorteil, dass mehrere *Threads* gleichzeitig eine Pipeline ausführen können. Während bei *OpenGL* große Treiber vorhanden sind, benötigt man für *Vulkan* nur kleine Treiber und ein Entwickler hat mehr Kontrolle über die Anwendung. Allerdings ist

dadurch der Implementierungsaufwand höher, da mehr konfiguriert werden muss, bevor mit *Vulkan* ein Bild gerendert werden kann.

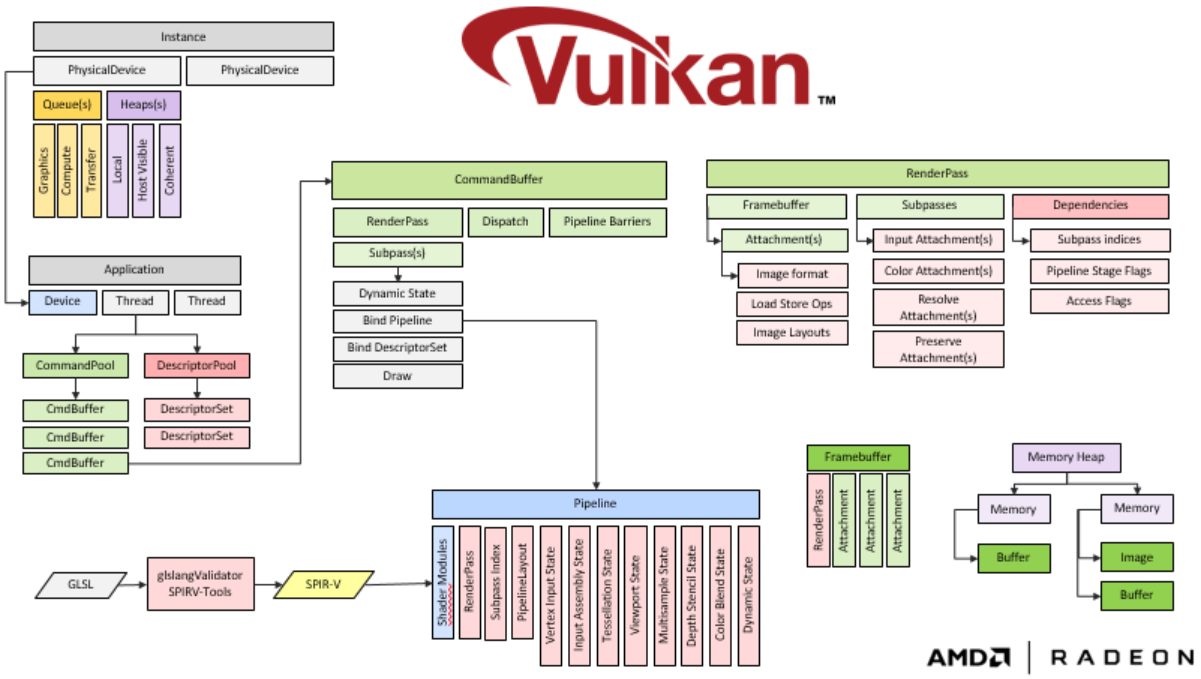

Schaubild 2.2.2.2: Die Vulkan-API im Detail mit zugehörigen Objekten

Schaubild 2.2.2.2 zeigt die *Vulkan*-API im Detail [GPUOPEN]. Die *Instance* dient dabei als Verbindung zwischen der Vulkanbibliothek und der Anwendung. Ein System kann kein, ein oder mehrere *PhysicalDevices*, also z.B. Grafikkarten die Vulkan unterstützen, haben. Diese *PhysicalDevices* legen entsprechend auch die Grenzen fest. Bspw. die Anzahl der maximalen Arbeitsgruppen für *Compute*-Aufgaben oder der verfügbare Video-RAM. Auf Grundlage dieser Grenzen kann die Anwendung das passende *PhysicalDevice* auswählen, dass für die Aufgabe der Anwendung am passendsten erscheint. Wichtig dabei sind die *Queue*-Familien. Eine *Queue* ist schließlich Konstrukt, dem Kommandos übergeben werden, die auf dem *PhysicalDevice* ausgeführt werden sollen. Dabei können nicht alle *Queues* Operationen verarbeiten. Es gibt drei wichtige Funktionalitäten, die eine *Queue* bereitstellen kann:

- *Compute*
- *Graphics*
- *Transfer*

Compute steht dafür, dass die *Queue Compute* Aufgaben, also letztlich *Compute*-Shader ausführen kann.
Queues, die *Graphics* können, können Grafikaufgaben, ausführen. Dazu zählen das Ausführen von *Vertex-* oder *Fragment-Shadern* und weitere Schritte aus der Grafikpipeline.
Transfer bedeutet, dass diese *Queue* Speicher vom Host-System in den Videospeicher transferieren kann.
Eine *Queue* kann also mindestens eine dieser drei Aufgaben ausführen. *Queues* die gleich sind, also die gleichen Operationen können, werden in *Queue*-Familien zusammengefasst.
Das *PhysicalDevice* wird dann als ein logisches *Device* abstrahiert, das dann als Schnittstelle zwischen Anwendung und dem *PhysicalDevice* dient.
Die Kommandos, die einer *Queue* übergeben werden, werden in einen *CommandBuffer* aufgezeichnet. Bevor jedoch ein *CommandBuffer* die Befehle aufzeichnet, muss eine *Pipeline* gebunden werden, die u.a. den Ablauf festlegt und auch die *Shader*-Module enthält, von denen dann auch die Befehle in einen *CommandBuffer* aufgezeichnet werden. Eine *Pipeline* benötigt neben den *Shader*-Modulen und vielen weiteren Konfigurationen einen *Renderpass*. Eine Ausnahme hierbei bildet eine *Compute-Pipeline*, da bei dieser kein *Rendering* stattfindet. Ein *Renderpass* steht eng in Verbindungen mit den *Framebuffern*. Im *Renderpass* werden u.a. Speicheranhänge zu den *Framebuffern* beschrieben.
Alle diese Teile müssen erstellt und konfiguriert werden, damit schließlich Befehle in die *CommandBuffer* aufgezeichnet und dann in einer *Queue* auf einem *PhysicalDevice* ausgeführt werden.

3 Entwurf und Implementierung

Im Folgenden wird die Software beschrieben und es wird auf die wichtigsten Quellcode-Teile eingegangen. Die Software ist in *C* geschrieben und die *Shader* in *GLSL*.

3.1 Partikelsystem

Ein Ziel dieser Arbeit ist die Umsetzung eines Partikelsystems. Das Partikelsystem selbst ist nicht sehr aufwändig und bietet keinen visuellen Mehrwert. Es dient lediglich als Mittel zum Zweck, um anschließend ein Vergleich durchzuführen.
Ein Partikel wird durch folgendes *struct* dargestellt.

```
/*
 * A particle has a position and a velocity
 */
typedef struct particle
{
    vector3f *position;
    vector3f *velocity;
    vector3f *color;
    float age;
} particle;
```

Schaubild 3.1.1: Das struct particle

Ein Partikel hat demnach eine Position, eine Geschwindigkeit und Richtung, eine Farbe und eine Lebensspanne.
Wie in Abschnitt 2.1.2 beschrieben besteht ein Partikelsystem aus einem oder mehreren Emittern, die dann die Partikel halten. Schaubild 3.1.2 zeigt die zugehörigen *structs*. Hierbei hält ein Emitter eine Position zur Initialisierung der Partikel. Die Partikel sind als Array vorhanden. Das Partikelsystem hält dann ein Array für die Emitter.

Die Tatsache, dass es mehrere Emitter geben kann und diese verschiedene Positionen haben können kommt nur in der *CPU*-Variante (siehe Abschnitt 3.2) zum Tragen. In den

Varianten *OpenGL* und *Vulkan* wird z.B. jeweils nur eine Rücksetzposition eines Emitters angegeben. Das hat den Effekt, dass mehrere Emitter ggf. zu einem werden.

```
/*
 * An emitter has a position and contains an array of particles
 */
typedef struct emitter
{
    vector3f *position;
    particle **particles;
    int pamount;
} emitter;

/*
 * A particle system consists of one or more emitter
 */
typedef struct particle_system
{
    emitter **emitters;
    int eamount;
} particle_system;
```

Schaubild 3.1.2: Das struct emitter und particle_system

3.2 Implementierungsdetails

Das Schaubild 3.2.1 gibt einen Überblick über die Softwarekomponenten.

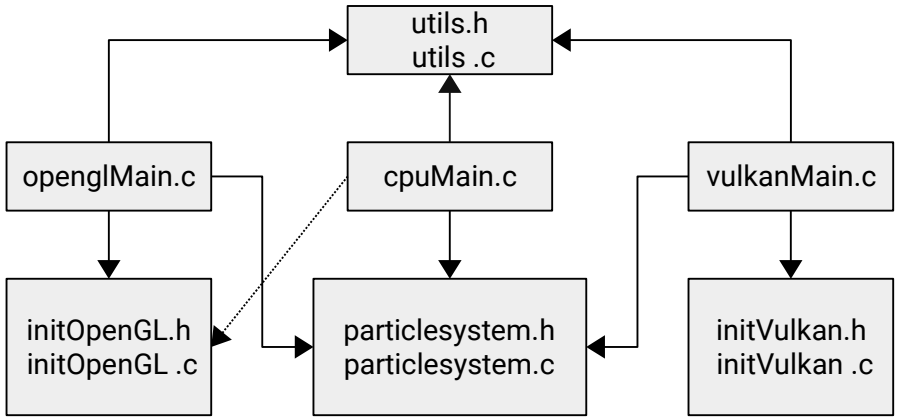

Schaubild 3.2.1: Softwarekomponenten

Es existieren drei Varianten:

i. CPU
ii. OpenGL
iii. Vulkan

Die erste Variante ist die Variante, bei der die Simulation auf der CPU durchgeführt wird und das Rendering mit OpenGL. Die OpenGL Variante verwendet OpenGL Compute-Shader zum simulieren der Partikel und Vertex- und Fragment-Shader zum rendern. Das gleiche gilt für Vulkan.

Des weiteren werden im *initOpenGL*-Modul OpenGL und GLFW initialisiert, sodass eine Verwendung möglich ist. Das *initVulkan*-Modul initialisiert Vulkan. Das Partikelsystem wird im Modul *particlesystem* beschrieben und außerdem befinden sich dort weitere Funktionen zum Verwalten des Partikelsystems.

3.2.1 CPU Variante

```
/************** RENDER LOOP *************/
double time, tFrame, tLast = 0;
while (!glfwWindowShouldClose(window))
{
    time = glfwGetTime();
    tFrame = time - tLast;
    tLast = time;

    /*** UPDATE ***/
    updateParticles((float) tFrame, ps, calcPos, calcCol);

    /*** RENDER ***/
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glfwGetFramebufferSize(window, &width, &height);

    emitter *e;
    particle *p;
    vector3f *pos;
    for (int j = 0; j < ps->eamount; j++)
    {
        e = (ps->emitters)[j];
        for (int i = 0; i < e->pamount; i++)
        {
            p = (e->particles)[i];
            pos = p->position;

            glColor3f(p->color->x, p->color->y, p->color->z);
            glBegin(GL_POINTS);
            glVertex3f(pos->x, pos->y, pos->z);
            glEnd();
        }
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Schaubild 3.2.1.1: cpuMain Render Loop

Hierbei wird zuerst die Funktion *updateParticles()* aufgerufen, um die Partikel zu simulieren. Um diese Funktion generisch zu halten, können Funktionszeiger injiziert werden, die auf eine Funktion zeigen, die dann die Position und Farbe des Partikels berechnen. Hierbei werden alle Partikel in einer Schleife aktualisiert. Partikel deren Lebenszeit abgelaufen ist oder die sich außerhalb des Bildschirms befinden werden

zurückgesetzt und neu initialisiert. Nach dem Simulationsschritt werden die Partikel dann mit *OpenGL* gerendert.

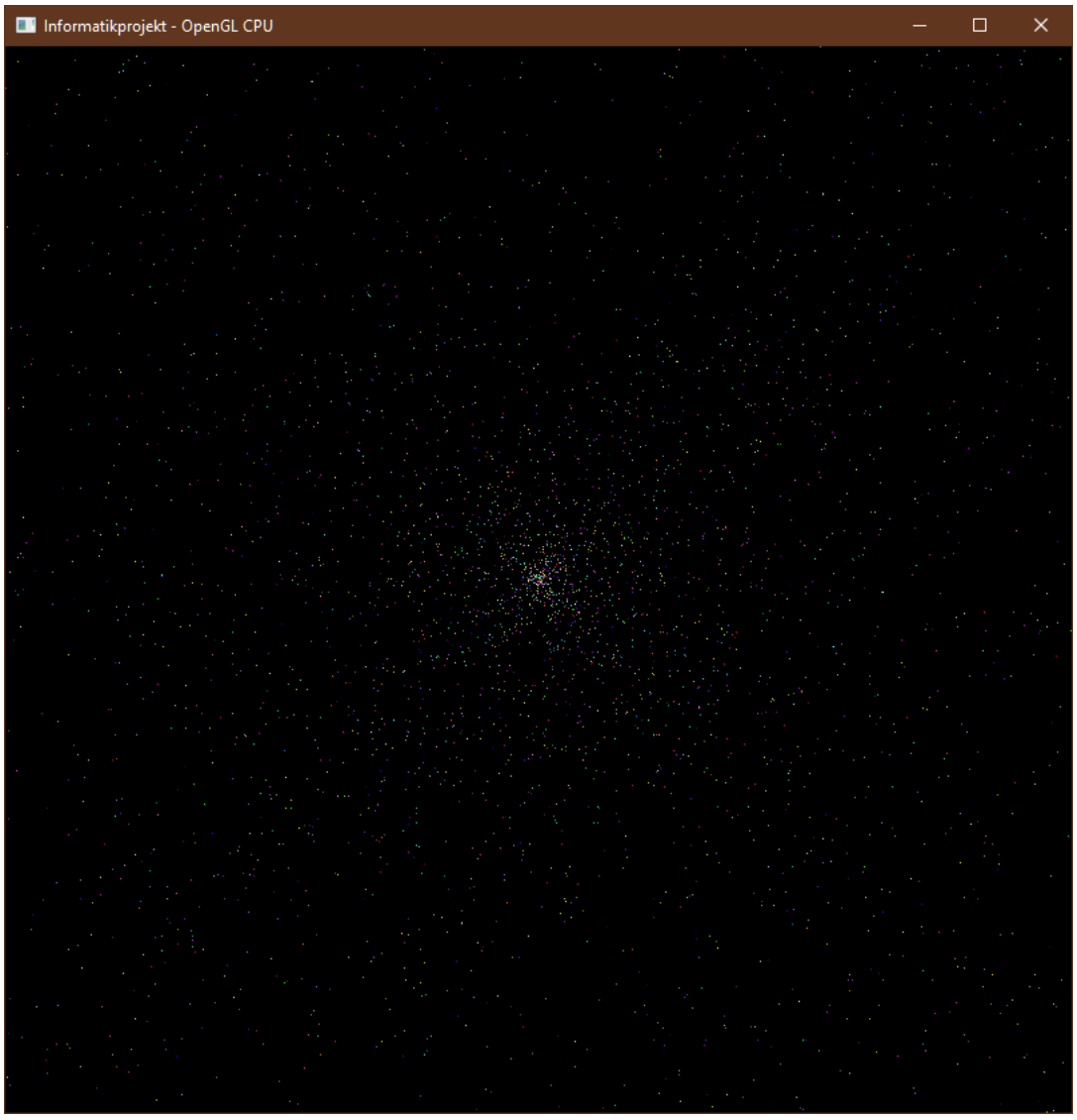

Schaubild 3.2.1.2: Die CPU Variante mit 10.000 Partikeln

Schaubild 3.2.1.2 zeigt die Ausführung dieser Variante mit 10.000 Partikeln.

3.2.2 OpenGL Variante

Eine weitere Variante ist die Variante mit *OpenGL*. Hierfür werden drei *Shader* verwendet:

- *Compute-Shader*
- *Vertex-Shader*
- *Fragment-Shader*

Der *Vertex-* und der *Fragment-Shader* sind hierbei trivial und geben lediglich die ankommenden Daten direkt weiter.

```
#version 460

layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 colIn;

layout(location = 0) out vec3 colV;

void main(void)
{
    colV = colIn;
    gl_Position = vec4(pos, 1);
}
```

Schaubild 3.2.2.1: Vertex-Shader der OpenGL-Variante

```
#version 460

layout(location = 0) in vec3 colV;
layout(location = 0) out vec4 colOut;

void main(void)
{
    colOut = vec4(colV, 1);
}
```

Schaubild 3.2.2.2: Fragment-Shader der OpenGL-Variante

Schaubild 3.2.2.1 zeigt den *Vertex-Shader*. Die Position und die Farbe eines Partikels werden hierbei in die Eingabe-Variablen geschrieben. Die Farbe wird dann einer eigenen Ausgabe-Variable und die Position in die eingebaute *OpenGL* Ausgabevariable *gl_Position* gespeichert.

Der *Fragment-Shader* in Schaubild 3.2.2.2 nimmt die Farbe entgegen, gibt diese allerdings direkt weiter, ohne dass diese wirklich geändert wird.

Die eigentliche Simulation der Partikel findet im *Compute-Shader* statt.

```
struct particle
{
    float px, py, pz;
    float vx, vy, vz;
    float cx, cy, cz;
    float age;
};

layout(std430, binding = 0) buffer particles
{
    particle p[];
};

layout(location = 0) uniform float dt;
layout(location = 1) uniform vec3 resetPos;
layout(location = 2) uniform uint maxParticles;
```

Schaubild 3.2.2.3: Input-Variablen für den Compute-Shader

Schaubild 3.2.2.3 zeigt die Eingabevariablen für den Compute-Shader. Ein *struct*, das das Partikel-*struct* aus dem Modul *particlesystem* direkt abbildet wird für ein Partikel verwendet. Des weiteren sind werden durch drei *uniforms* weitere Informationen hineingegeben. r*esetPos* ist die Position, an die ein Partikel zurückgesetzt werden soll und *maxParticles* ist die Gesamtanzahl der Partikel, die simuliert werden. Diese beiden *uniforms* werden am Anfang der Anwendung gesetzt. In der Renderschleife wird dann jedes Mal das *dt* gesetzt. *dt* ist die Zeit, die seit dem letzten Frame verstrichen ist.

In Schaubild 3.2.2.4 ist schließlich der Inhalt der *main()* des Compute-Shaders zu finden. OpenGL bietet einige Funktionen an, die bereits im Voraus implementiert sind. Eine Zufallsfunktion gehört allerdings nicht dazu. Die *noise()* Funktion generiert zwar auch etwas Zufall, aber es ist nicht sichergestellt, dass diese überall implementiert ist. Somit wurde eine eigene Pseudozufallsfunktion basierend auf den *Xorshift-PRNG*-Algorithmus von [Marsaglia 2003, S. 4f] implementiert. Dieser fordert einen *Seed* und da jedes

einzelne Partikel zufällig vor generiert wurde, werden die alten Werte aus Position, Geschwindigkeit und Richtung und Farbe, verwendet. Diese werden in einem schnellen *Hash*-Verfahren gehasht und als *Seed* verwendet.

```
void main()
{
    uint gid = gl_GlobalInvocationID.x;

    if (gid <= maxParticles)
    {
        particle part = p[gid];

        uint hash1 = hash(uvec3(uint(part.px * FLOAT_MAX), uint(part.cy * FLOAT_MAX), uint(part.vz * FLOAT_MAX)));
        uint hash2 = hash(uvec3(uint(part.vx * FLOAT_MAX), uint(part.py * FLOAT_MAX), uint(part.cz * FLOAT_MAX)));
        uint hash3 = hash(uvec3(uint(part.cx * FLOAT_MAX), uint(part.vy * FLOAT_MAX), uint(part.pz * FLOAT_MAX)));

        if (part.age < 0 || part.px > 1 || part.py > 1 || part.pz > 1 || part.px < -1 || part.py < -1 || part.pz < -1)
        {
            part.px = resetPos.x;
            part.py = resetPos.y;
            part.pz = resetPos.z;

            part.age = rand(hash(uvec3(hash1, hash2, hash3))) % (250 - 60 + 1) + 60;

            part.vx = foreSign(hash1) * float(rand(hash2)) * FLOAT_FACTOR;
            part.vy = foreSign(hash3) * float(rand(hash1)) * FLOAT_FACTOR;
            part.vz = foreSign(hash2) * float(rand(hash3)) * FLOAT_FACTOR;

            part.cx = float(rand(hash1 ^ hash2)) * FLOAT_FACTOR;
            part.cy = float(rand(hash2 ^ hash3)) * FLOAT_FACTOR;
            part.cz = float(rand(hash3 ^ hash1)) * FLOAT_FACTOR;
        }
        else
        {
            part.px += part.vx * dt;
            part.py += part.vy * dt;
            part.pz += part.vz * dt;

            part.age -= 0.01f;
        }

        p[gid] = part;
    }
}
```

Schaubild 3.2.2.4: Die main() des Compute-Shader für OpenGL

Dadurch ist sichergestellt, dass jedes Partikel einzigartig bleibt. Das hinzuziehen von externen *Seeds* hätte dafür gesorgt, dass sich diesen *Seed* sämtliche Partikel teilen und sich somit angleichen.

Für den *Compute-Shader* als Eingabe dient ein *SSBO* (*Shader Storage Buffer Object)*. Das ist ein *float*-Array, das aus dem Partikelsystem generiert wurde. Dieses Array wird als *SSBO* in den Grafikspeicher geladen, dem *Compute-Shader* als Quelle zugeführt und anschließend als Zeichenquelle genutzt. Das bedeutet der *Vertex-Shader* verwendet das *SSBO* als *Vertex-Array* und zeichnet die Partikel.

```
/************* RENDER LOOP *************/
double time, tFrame, tLast = 0;
while (!glfwWindowShouldClose(window))
{
    time = glfwGetTime();
    tFrame = time - tLast;
    tLast = time;

    /*** RENDER ***/
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    glfwGetFramebufferSize(window, &width, &height);
    glBindVertexArray(vertexArray);
    glUseProgram(renderShaderProgram);
    glDrawArrays(GL_POINTS, 0, PARTICLE_AMOUNT);
    glBindVertexArray(0);

    /*** UPDATE ***/
    glUseProgram(computeShaderProgram);
    glUniform1f(dtUniformLocation, tFrame);
    glDispatchCompute(PARTICLE_AMOUNT / WORKGROUP_SIZE_X, 1, 1);
    glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT | GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Schaubild 3.2.2.5: Die Renderschleife von der OpenGL Variante

Schaubild 3.2.2.5 zeigt die Renderschleife der *OpenGL*-Variante. Zuerst wird mit den *Vertex-* und *Fragment-Shader* das *Rendering* durchgeführt und anschließend mit dem *Compute-Shader* die Simulation. Da die Partikel vorinitialisiert wurden, wird mit dem Rendern begonnen.

Schaubild 3.2.2.6 zeigt die Ausführung der *OpenGL* Variante mit 10.000.000 Partikeln.

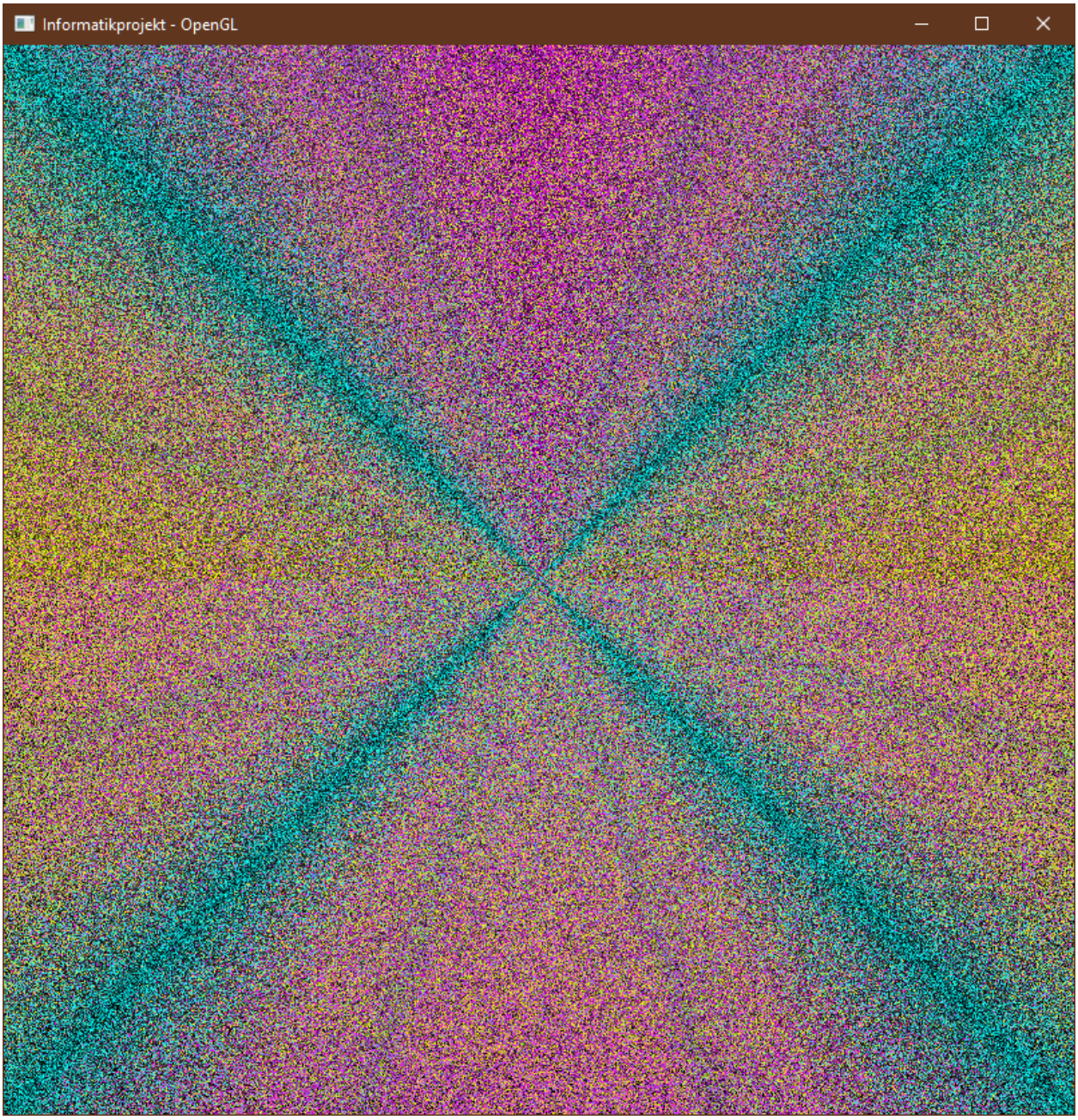

Schaubild 3.2.2.6: Die OpenGL Variante mit 10.000.000 Partikeln

3.2.3 Vulkan Variante

Für die *Vulkan* Variante werden im wesentlichen die gleichen *Shader* verwendet, wie für *OpenGL*. Diese werden lediglich so angepasst, damit sie erfolgreich in das *SPIR-V*-Fomat übersetzt werden können.

Für die *Vulkan*-Variante werden zwei *Vulkan-Pipelines* erstellt. Einmal eine *Compute-Pipeline* für die Simulation, in der dann der *Compute-Shader* verwendet wird und einmal eine *Graphics-Pipeline*, die dann für das *Rendering* zuständig ist. Für beide *Pipelines* sind unterschiedliche *Command-Buffer* und auch *Queues* im Einsatz. Des Weiteren werden z.T. unterschiedliche Konfigurationen benötigt. Deshalb gibt es für den *Computing*-Teil ein eigenes *struct* (Schaubild 3.2.3.1) und für den *Rendering*-Teil ein eigenes (Schaubild 3.2.3.2).

```
typedef struct compute {
    VkInstance instance;

    VkPhysicalDevice physicalDevice;
    VkDevice device;

    VkPipeline pipeline;
    VkPipelineLayout pipelineLayout;
    VkShaderModule shaderModule;

    VkCommandPool commandPool;
    VkCommandBuffer commandBuffer;

    VkDescriptorSetLayout particleBufferDescriptorSetLayout;
    VkDescriptorPool particleBufferDescriptorPool;
    VkDescriptorSet particleBufferDescriptorSet;
    VkBuffer particleBuffer;
    VkDeviceMemory particleBufferMemory;
    uint32_t particleBufferSize;

    VkDescriptorSetLayout dtUniformBufferDescriptorSetLayout;
    VkDescriptorPool dtUniformBufferDescriptorPool;
    VkDescriptorSet dtUniformBufferDescriptorSet;
    VkBuffer dtUniformBuffer;
    VkDeviceMemory dtUniformBufferMemory;
    uint32_t dtUniformBufferSize;

    VkDescriptorSetLayout staticInUniformBufferDescriptorSetLayout;
    VkDescriptorPool staticInUniformBufferDescriptorPool;
    VkDescriptorSet staticInUniformBufferDescriptorSet;
    VkBuffer staticInUniformBuffer;
    VkDeviceMemory staticInUniformBufferMemory;
    uint32_t staticInUniformBufferSize;

    VkQueue queue;
    uint32_t queueFamilyIndex;

    VkSemaphore semaphore;
} Compute;
```

Schaubild 3.2.3.1: Das struct Compute der Vulkan-Variante

```
typedef struct graphics {
    VkInstance instance;

    VkPhysicalDevice physicalDevice;
    VkDevice device;

    VkSurfaceKHR surface;
    VkSwapchainKHR swapChain;

    VkImageView *imageViews;
    uint32_t imageViewsSize;

    VkRenderPass renderPass;
    VkFramebuffer *framebuffers;

    VkPipeline pipeline;
    VkPipelineLayout pipelineLayout;
    VkShaderModule vertexShaderModule;
    VkShaderModule fragmentShaderModule;

    VkCommandPool commandPool;
    VkCommandBuffer *commandBuffers;

    VkBuffer particleBuffer;
    uint32_t particleBufferSize;

    VkQueue queue;
    uint32_t queueFamilyIndex;

    VkSemaphore renderComplete;
    VkSemaphore presentComplete;
    VkSemaphore semaphore;
} Graphics;
```

Schaubild 3.2.3.2: Das struct Graphics der Vulkan-Variante

Während das *Graphics-struct* vor allem die Teile zum Rendern beinhaltet, ist im *Compute-struct* zusätzlich die *Buffer*-Verwaltung untergebracht, da die *Buffer* für z.B. Partikel hauptsächlich von diesem *struct* verwendet werden. Der Partikel-*Buffer* wird im *Graphics-struct* dann lediglich zum auslesen verwendet.

Schaubild 3.2.3.3 zeigt zudem die Funktions-Prototypen der *Vulkan*-Variante, zum initialisieren der einzelnen *structs* bzw. deren *Member*.

```
// Shutdown
void shutdownGLFW(GLFWwindow *window);
void shutdownComputeVulkan(Compute *compute);
void shutdownGraphicsVulkan(Graphics *graphics);

// General
void createInstance(Compute *compute, Graphics *graphics);
void findPhysicalDevice(Compute *compute, Graphics *graphics);
void createDevice(Compute *compute, Graphics *graphics);
void createParticleBuffer(Compute *compute, Graphics *graphics);

// Compute
void createComputeBuffers(Compute *compute);
void createComputeDescriptorSetLayouts(Compute *compute);
void createComputeDescriptorSets(Compute *compute);
void createComputePipeline(Compute *compute);
void fillComputeBuffers(Compute *compute, float *particles, Dt *dtData, StaticIn *staticInData);
void createComputeCommandBuffer(Compute *compute);

// Graphics
void createGraphicsSurface(Graphics *graphics, GLFWwindow *window);
void createSwapchain(Graphics *graphics);
void createGraphicsPipeline(Graphics *graphics);
void createFramebuffer(Graphics *graphics);
void createGraphicsCommandBuffers(Graphics *graphics);

// ELse
void mapBufferMemory(Compute *compute, VkDeviceMemory memory, void *inputData, uint32_t dataSize);
void createSemaphore(VkDevice device, VkSemaphore *semaphore);
```

Schaubild 3.2.3.3: Weitere Vulkan-Funktionen der Vulkan-Variante

Die *Shutdown*-Funktionen dienen hierbei zum Herunterfahren oder Zerstören bestimmter Objekte am Ende des Programmablaufs.
Die bei *General* gruppierten Funktionen initialisieren die *Vulkan-Instance*, das *Vulkan-Device* mit den *Queues* und den Partikel-*Buffer*.
Die *Compute*-Funktionen initialisieren den Rest des *Compute-structs* und führen dann eine Aufzeichnung des *CommandBuffers* aus.
Das Gleiche gilt für die *Graphics*-Funktionen.

Nach dem Aufrufen dieser Funktionen ist *Vulkan* weitestgehend bereit die Partikel zu simulieren und das Ergebnis auf den Bildschirm zu zeichnen.

```
// Loop
double time, tLast = 0;
Dt tFrame = {};
uint32_t imageIndex;
while (!glfwWindowShouldClose(window))
{
    time = glfwGetTime();
    tFrame.dt = (float) (time - tLast);
    tLast = time;

    /*** RENDER ***/
    ASSERT_VK( f: vkAcquireNextImageKHR(graphics.device, graphics.swapChain, UINT64_MAX,
            graphics.presentComplete, VK_NULL_HANDLE, &imageIndex))

    graphicsSubmitInfo.pCommandBuffers = &(graphics.commandBuffers[imageIndex]);
    ASSERT_VK( f: vkQueueSubmit(graphics.queue, 1, &graphicsSubmitInfo, VK_NULL_HANDLE))

    graphicsPresentInfo.pImageIndices = &imageIndex;
    ASSERT_VK( f: vkQueuePresentKHR(graphics.queue, &graphicsPresentInfo))

    /*** UPDATE ***/
    // Update dt
    mapBufferMemory(&compute, compute.dtUniformBufferMemory, &tFrame, sizeof(Dt));
    ASSERT_VK( f: vkQueueSubmit(compute.queue, 1, &computeSubmitInfo, VK_NULL_HANDLE))

    glfwPollEvents();
}
```

Schaubild 3.2.3.4: Die Renderschleife der Vulkan-Variante

Schaubild 3.2.3.4 zeigt schließlich die Renderschleife der *Vulkan*-Variante. Auch bei dieser Variante wird mit dem Rendern begonnen, da die Partikel vorinitialisiert wurden. *GLFW*, welches in diesem Projekt in der *Vulkan*-Variante dazu verwendet wurde, um ein Fenster zu erstellen, nutzt die *KHR*-Erweiterungen. Darin enthalten sind u.a. *Surface*, in welchem die gerenderten Bilder dargestellt werden können und die *Swapchain*, welches letztlich ein Array von Bildern ist, die angezeigt werden können. Aus den *Framebuffern* werden Bilder erstellt, die dann der *Swapchain* zugeführt werden, die dann nach und nach in einem *Surface* angezeigt werden. Der erste Befehl im *Rendering*-Teil der Schleife holt das nächste Bild. Danach wird der *CommandBuffer* an die *Queue* übergeben. Die *Queue* führt die Befehle aus und das Bild wird gezeichnet.

Danach wird mit der *mapMemory*-Funktion die *Deltatime*, also die Zeit, die seit dem letzten Frame vergangen ist, an die GPU gesendet, da diese im *Compute-Shader* benötigt wird. Daran anschließend wird schließlich der *CommandBuffer* des *Computing*-Teils an eine *Queue* übergeben und ausgeführt.

```
ASSERT_VK( f: vkBeginCommandBuffer(compute->commandBuffer, &beginInfo))

vkCmdBindPipeline(compute->commandBuffer,  pipelineBindPoint: VK_PIPELINE_BIND_POINT_COMPUTE, compute->pipeline);
VkDescriptorSet descriptorSets[] = {
        compute->particleBufferDescriptorSet,
        compute->dtUniformBufferDescriptorSet,
        compute->staticInUniformBufferDescriptorSet
};
vkCmdBindDescriptorSets(compute->commandBuffer,  pipelineBindPoint: VK_PIPELINE_BIND_POINT_COMPUTE,
        compute->pipelineLayout,  firstSet: 0,  descriptorSetCount: 3,
        descriptorSets,  dynamicOffsetCount: 0,  pDynamicOffsets: NULL);

vkCmdDispatch(compute->commandBuffer,  groupCountX: PARTICLE_AMOUNT / WORKGROUP_SIZE_X, WORKGROUP_SIZE_Y, WORKGROUP_SIZE_Z);

ASSERT_VK( f: vkEndCommandBuffer(compute->commandBuffer))
```

Schaubild 3.2.3.5: Aufzeichnen des Compute-Command-Buffers

Die *CommandBuffer*, die in der Renderschleife benutzt werden, wurden im Vorfeld einmalig aufgezeichnet. Schaubild 3.2.3.5 zeigt diesen Prozess für den *Compute-Command-Buffer*. Mit *vkBeginCommandBuffer* wird die Aufzeichnung begonnen.
Die ersten Befehle, die der *CommandBuffer* aufzeichnen soll, sind die Befehle zum binden der *Compute-Pipeline*. Die *Compute-Pipeline* ist so konfiguriert, dass sie die gewünschten Schritte, darunter auch der *Compute-Shader*, ausführt.
Danach werden die *Deskriptoren* gebunden. Die *Deskriptoren* enthalten die Informationen über die im *Compute-Shader* verwendeten *Buffer*, wie Größe und Bindungspunkte im *Shader*.
Nachdem nun die Befehle zum Binden der Pipeline und der *Buffer*-Beschreibungen im *Command-Buffer* aufgezeichnet wurden, wird die Ausführung der *Compute-Pipeline* mit dem *Compute-Shader* aufgezeichnet.
Die Aufzeichnung schließt mit *vkEndCommandBuffer*. Der *Command-Buffer* enthält nun alle Befehle, die *Queue* ausführen muss.

Die *Vulkan*-Variante ausgeführt mit 10.000.000 Partikeln ist in Schaubild zu sehen.

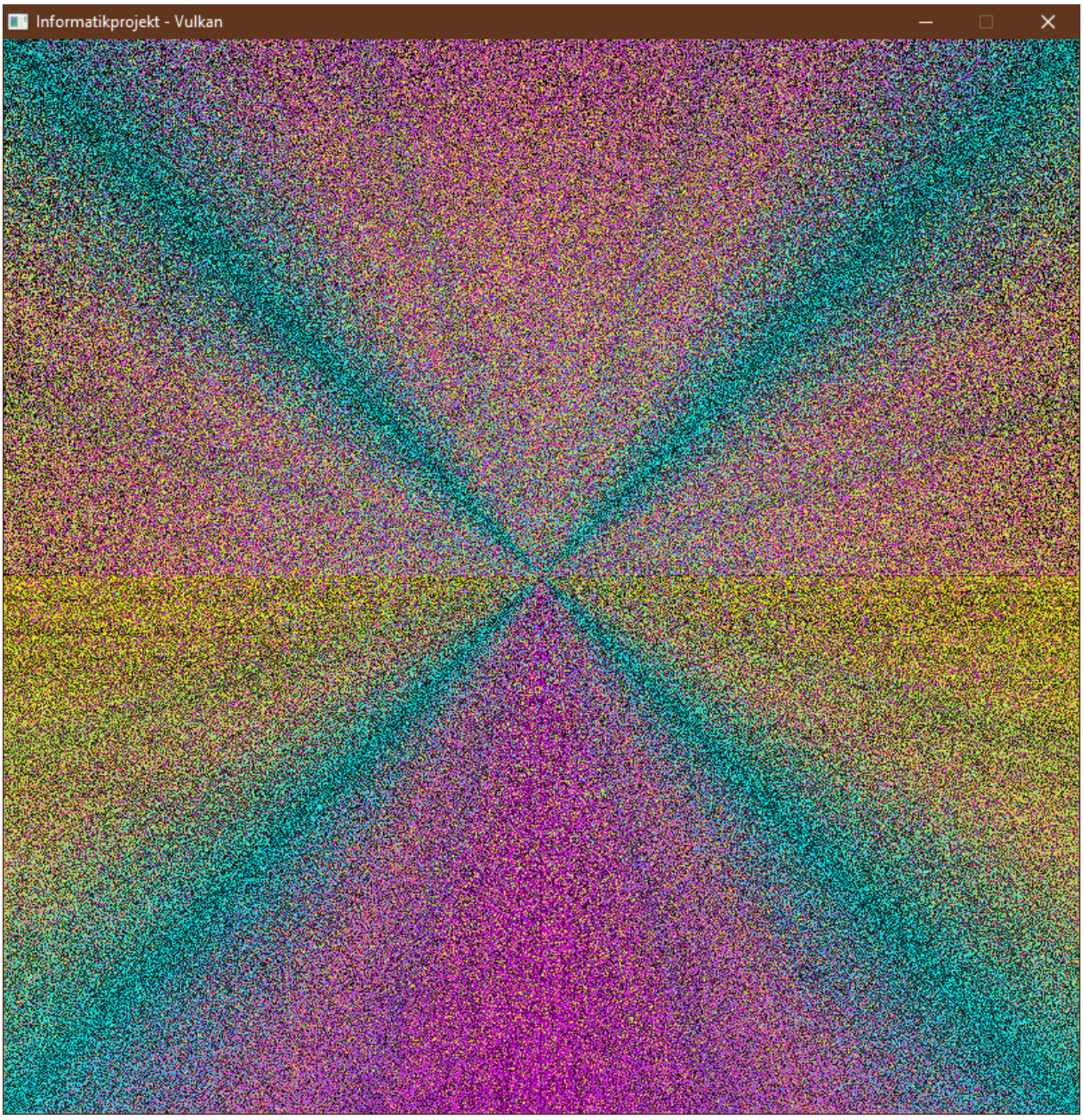

Schaubild 3.2.3.6: Die Vulkan-Variante mit 10.000.000 Partikeln

4 Performancevergleich und Aufwand

Im folgenden Abschnitt soll nun ein Performancevergleich und Aufwandsvergleich stattfinden. Beim Aufwandvergleich wird der Umsetzungeaufwand gemessen und beim Performancevergleich die Performance bei der Ausführung der Simulation und des Rendering.

4.1 Aufwand

Der Implementierungsaufwand wird in diesem Abschnitt etwas näher betrachtet.
Der Aufwand lässt sich u.a auf zwei folgende Weisen messen:

1. benötigte Zeit
2. *Lines of Code* (*LOC*) [Park 1992, S. 1]

4.1.1 Benötigte Zeit

Die genaue Zeit ist hierbei ein weniger geeignetes Mittel, da der Autor sowohl im Umgang mit *OpenGL* als auch *Vulkan* nicht erfahren war. Aspekte, wie *Shader*-Programmierung oder *Buffer*-Verwaltung, spielten erst in diesem Projekt eine Rolle. Außerdem war der Entwicklungszeitraum stetig unterbrochen und genaue Zeitmessungen liegen nicht vor.
Doch eine Abschätzung kann an dieser Stelle getroffen werden. *OpenGL* wird hauptsächlich von Treibern implementiert. Nur ein kleinerer Teil der Spezifikation bleibt übrig, den Entwickelnde kennen müssen. Dagegen kommen bei *Vulkan* nur kleine Treiber zum Einsatz und die Entwickelnden müssen deutlich mehr der Spezifikation kennen, um eine funktionierende Anwendung entwickeln zu können [NVIDIA]. Das bedeutet u.a., dass deutlich mehr Quellcode geschrieben werden muss, den sonst ein Treiber implementiert hätte. Das lässt an dieser Stelle die Schlussfolgerung zu, dass mehr geschriebener Quellcode und das Verstehen mehr Aspekte gegenüber *OpenGL* die Entwicklungszeit verlängert.
Diese Feststellung lies sich, auch wenn keine genauen Zeitmessungen vorliegen, auch bei diesem Projekt beobachten. Die Umsetzung mit *Vulkan* hat tendenziell länger gedauert als mit *OpenGL*.

4.1.2 Lines of Code

Die Größe bzw. der Umfang eines Projektes lässt sich u.a. auch an den Anzahl der Codezeilen festmachen. *LOC* dient zwar als Überbegriff für sämtliche Arten von *LOC*, meint aber auch speziell alle Codezeilen, also auch die Leerzeilen und Kommentare. Eine Art von *LOC* ist die *Source Lines of Code*, eine Art die *Logical Lines of Code*, wobei letzteres auch unter *Number of Statements* (*NOS*) bekannt ist.
SLOC sind die Anzahl der Quellcodezeilen ohne Leerzeilen und Kommentare, wohingegen die *LLOC* für die Anzahl der Anweisungen steht. Dieser Unterschied soll folgendes Beispiel verdeutlichen:

```
1  int main(void)
2  {
3      for (int i = 0; i < 100; i++)
4      {
5          // Addiere 1 zu i dazu
6          i = i + 1;
7      }
8
9      return 0; // Gib 0 zurueck
10 }
```

Schaubild 4.1.2.1: Beispielcode mit Zeilenangaben

Schaubild 4.1.2.1 zeigt ein Quellcodebeispiel mit zehn *LOC*. Also die Leerzeilen und Zeilen mit ausschließlich Kommentaren wird hierbei mitgezählt. Dieses Beispiel hat allerdings nur acht *SLOC*. Die Zeilen 5 und 8 werden nicht mitgezählt, da Zeile 5 nur ein Kommentar enthält und Zeile 8 eine Leerzeile ist. Als valide Werte für die *LLOC* können drei oder fünf angesehen werden. Je nachdem, wie Anweisungen definiert werden. Die *for*-Schleife kann als eine Anweisung gesehen werden, oder aber aufgeteilt werden in die drei Anweisungen innerhalb der Klammern. Für dieses Projekt werden die *LLOC* allerdings nicht erfasst. Tabelle 4.1.2.1 gibt nochmals einen Überblick über die erfassten *LOC*.

Tabelle 4.1.2.1: LOC-Tabelle des Beispielquellcodes

Quellcode	LOC	SLOC	LLOC
Schaubild 4.1.2.1	10	8	5

Tabelle 4.1.2.2 gibt nun einen Überblick über die *LOC* der einzelnen Projektdateien.

Tabelle 4.1.2.2: LOC-Tabelle der einzelnen Projektdateien

Quellcode	LOC	SLOC
cpuMain.c	89	64
openglMain.c	115	84
initOpenGL.c	115	95
initOpenGL.h	19	14
ComputeShader.glsl	88	71
VertexShader.glsl	12	9
FragmentShader.glsl	9	7
vulkanMain.c	145	108
initVulkan.c	896	702
initVulkan.h	146	113
ComputeShader.comp	96	77
VertexShader.vert	13	10
FragmentShader.frag	9	7
runCompiler.bat	3	3
particlesystem.c	211	145
particlesystem.h	118	41
utils.c	36	29
utils.h	9	6

Diese Ergebnisse können noch nach Modul bzw. Variante zusammengefasst werden. Diese Zusammenfassung ist in Tabelle 4.1.2.3 zu sehen. Hierbei sei angemerkt, dass die *CPU*-Variante wenige Teile aus der *OpenGL*-Variante nutzt, die in dieser Tabelle nicht berücksichtigt wurden.

Tabelle 4.1.2.3: LOC nach Variante bzw. Modul zusammengefasst

Variante / Modul	LOC	SLOC
CPU	89	64
OpenGL	358	280
Vulkan	1308	1020
particlesystem	329	196
utils	45	35

Sowohl bei den *LOC*, als auch bei den *SLOC* ist die Tendenz zu erkennen, dass die *Vulkan*-Variante etwas mehr als drei Mal aufwändiger, in Bezug auf *LOC*, ist, als die *OpenGL*-Variante. In Diagramm 4.1.2.2 ist diese Tendenz nochmals visuell dargestellt.

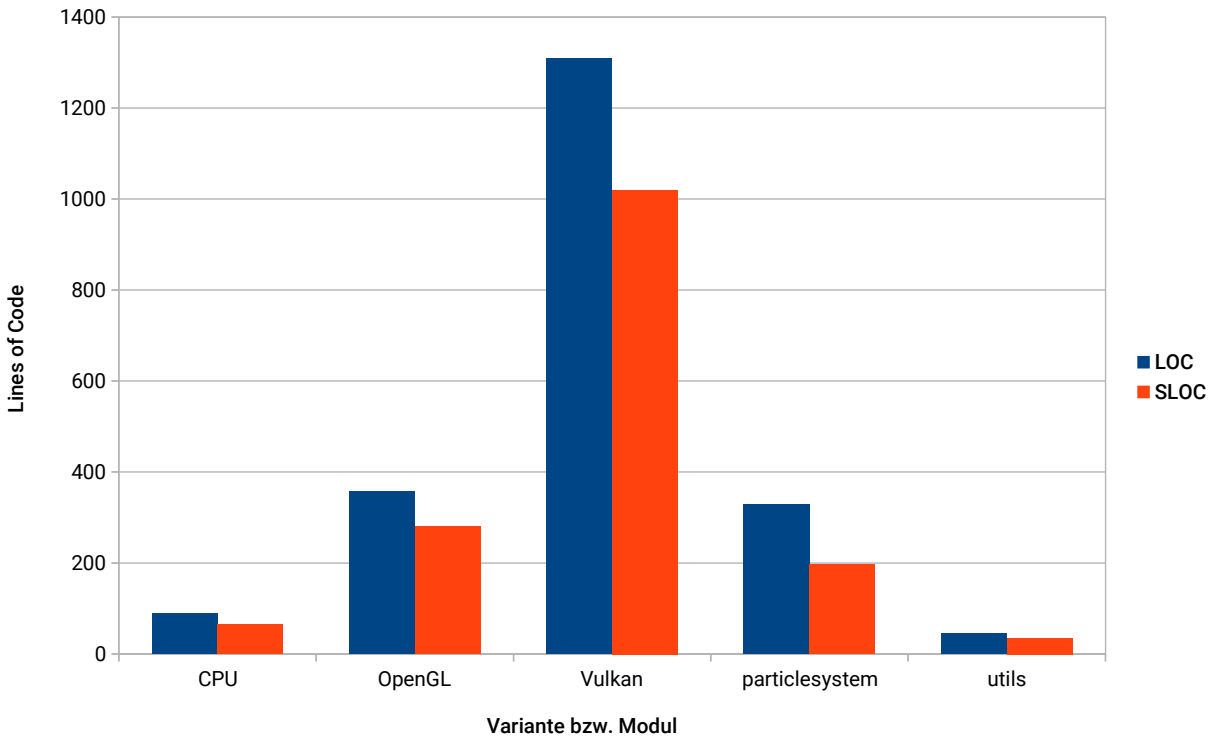

Schaubild 4.1.2.2: Balkendiagramm mit den LOC zum visuellen Vergleich

Des Weiteren lässt sich erkennen, dass alleine die *Vulkan*-Variante knapp zwei Drittel (genauer 64%) der gesamten *SLOC* ausmacht oder 78% im direkten Vergleich zur *OpenGL*-Variante. Die Anteile sind in den Diagrammen 4.1.2.3 und 4.1.2.4 dargestellt.

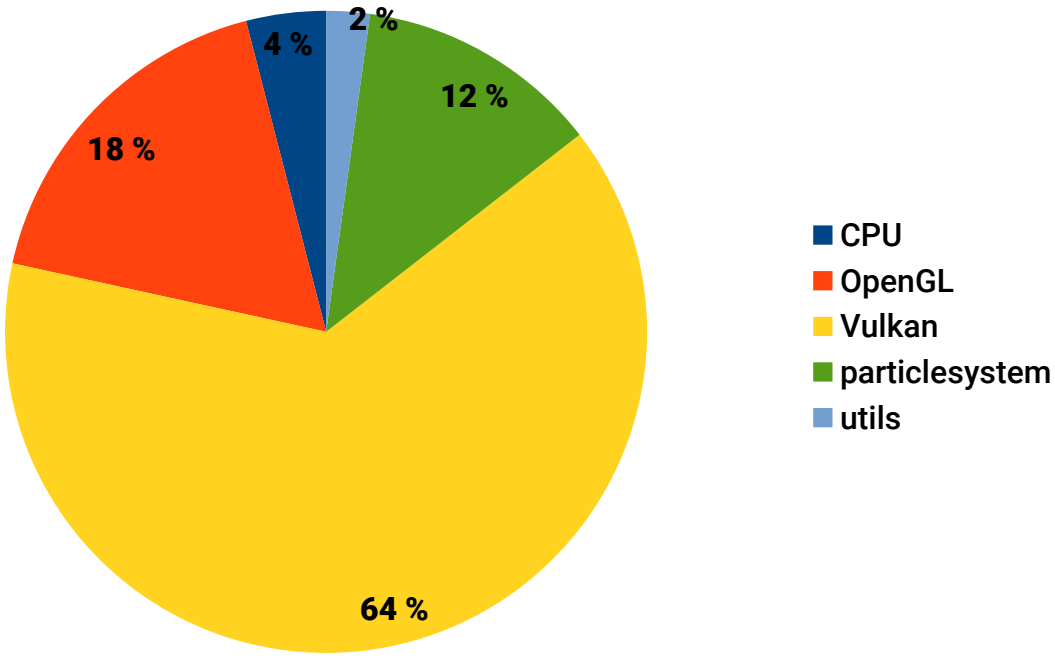

Schaubild 4.1.2.3: Anteile SLOC am Gesamtprojekt

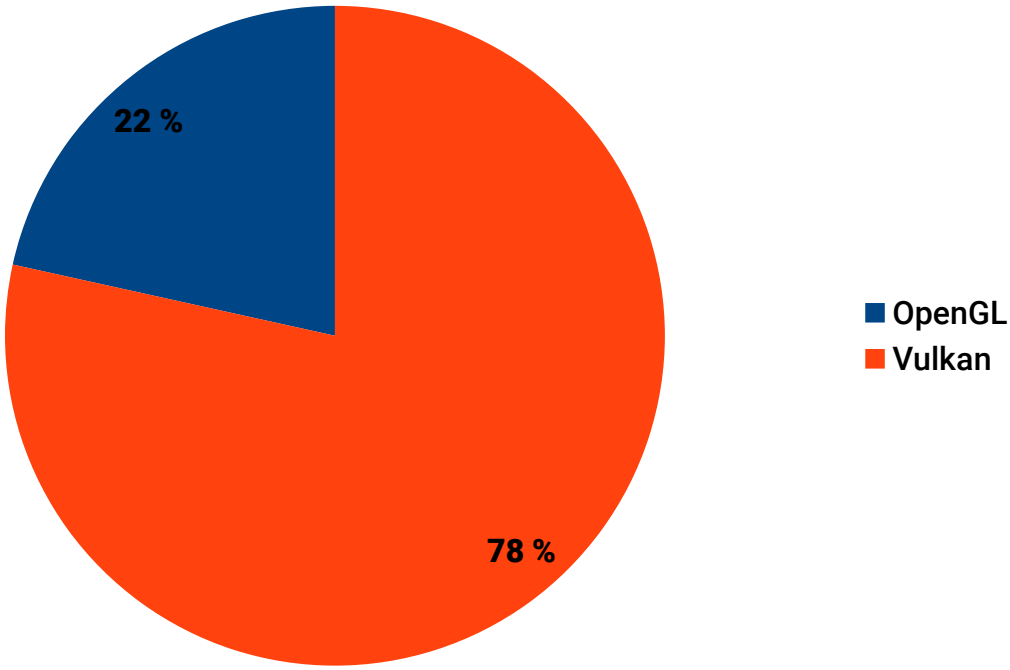

Schaubild 4.1.2.4: Anteil SLOC von Vulkan im direkten Vergleich zu OpenGL

Diese Diagramme und Zeitabschätzungen beziehen sich allerdings vor allem auf die Initialisierung der einzelnen Komponenten, bis das Simulieren und Rendern ausgeführt werden kann. Während bei *OpenGL* der Zustand jedes Mal neu gesetzt werden muss, so

ist der Zustand bei *Vulkan* u.a. in den *Pipelines* festgelegt und die *Command-Buffer* existieren zur Ausführung ebenso. Das bedeutet, einmal eine *Pipeline* erstellt und einen *Command-Buffer* befüllt, kann dieser genauso wiederverwendet werden und der Zustand muss nicht jedes Mal neu gesetzt, wie es bei *OpenGL* der Fall wäre. Dadurch dass jede *Pipeline* und *Command-Buffer* für sich alleine steht, kann es auch nicht vorkommen, dass ein Zustand nicht mitbedacht wird oder vergessen wird zu setzen. Für gleiche Aufgaben ist nach der Initialisierung also weniger Code erforderlich, um *Vulkan* arbeiten zu lassen.

4.2 Performance

Performancemessungen in der Computergrafik werden oft mit den Anzahl der gerenderten Bilder (*FPS*) in der Sekunde gemessen. Die *FPS* stehen direkt in Verbindung zur *Frametime*. Die *Frametime* ist die Zeit, die seit dem letzten *Frame* vergangen ist. Ein Durchlauf der *Renderschleife* entspricht einem *Frame*, somit ist die *Frametime* die Zeit, die benötigt wird um die *Renderschleife* einmal zu durchlaufen. Der Zusammenhang zu *FPS* ergibt sich unmittelbar, da die mittlere *Frametime* der Kehrwert eines *FPS* ist.

Beispiel an einer 60 *FPS* Anwendung: $\frac{60}{s} = \frac{1\,s}{60} = 0{,}01\overline{6}\,s$

Eine 60 *FPS* Anwendung hat somit eine mittlere *Frametime* von 16,$\overline{6}$ ms. Allerdings trifft ein *FPS* nur eine Aussage über die Anzahl der Bilder pro Sekunde. Aber 60 *FPS* kann bedeuten, dass ein Teil der Bilder mit langsamerer *Frametime* und der andere Teil schnellerer *Frametime* gerendert wurden. Im Mittel sind dies dann trotzdem 60 *FPS*.
Die Ausführungszeit eignet sich gut als Metrik um eine Performancemessung durchzuführen [Brunst & Mueller 2011, S. 36]. Für dieses Projekt wird zudem nur die mittlere *Frametime* betrachtet und keine Vermutungen darüber angestellt, weshalb die *Frametimes* u.U. an einer Stelle deutlich schneller oder langsamer in Erscheinung treten.

Für die *CPU*-Variante wird eine Messung durchgeführt, während für die *OpenGL*- und *Vulkan*-Variante jeweils drei Messungen durchgeführt werden. Die erste Messung mit 100.000 Partikeln, die zweite mit 1 Millionen Partikeln und de dritte mit 10 Millionen Partikeln. Jede Messung erzielt dabei 100 Messwerte, die über einen Zeitraum von

1.000 *Renderschleifen*-Durchläufen ermittelt werden. Das bedeutet, dass pro Messung jede zehnte *Frametime* gespeichert wird. Der Grund für die Anzahl der 100 Messwerte ist, dass die Darstellung durch einen Graphen dadurch übersichtlicher ist und statt über einen kurzen Zeitraum, wird ein längerer Zeitraum von 1.000 *Frames* beobachtet, um punktuell auftretende Performanceschwankungen durch andere Systemlast auszugleichen. Die Anzahl der Partikel im Projekt ist auf etwas mehr als 60 Millionen Partikeln nach oben begrenzt, da ansonsten die zulässige *Workgroup*-Größe der für die Messung verwendeten Grafikkarte überschritten wird.

4.2.1 Erste Messung

Die erste Messung wird mit 100.000 Partikeln durchgeführt. Für diese Messung wird außerdem die *CPU*-Variante mit betrachtet.
Die *CPU*-Variante hat eine mittlere *Frametime* von 59,11 ms bei 100.000 Partikeln. Das macht eine mittlere Anzahl von 17 *FPS*, was schon nicht mehr so gut ist.
Diagramm 4.2.1.1 zeigt die Messung der *CPU*-Variante im Detail.

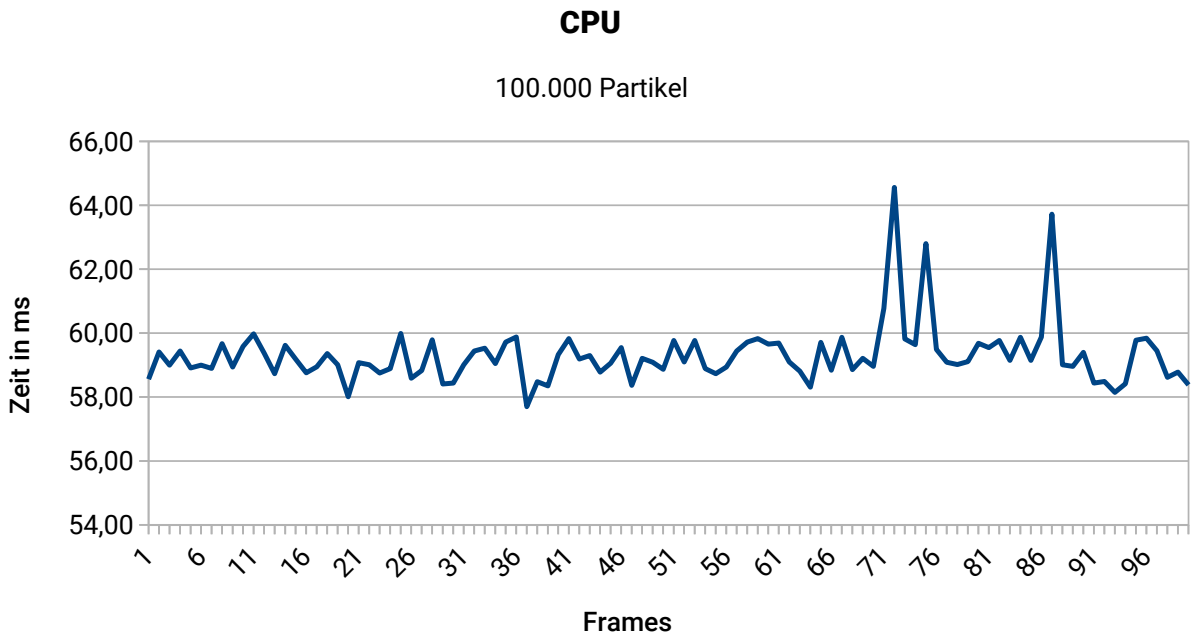

Schaubild 4.2.1.1: Mittlere Frametimes der CPU-Variante bei 100.000 Partikeln

Viel besser dagegen schneiden die *OpenGL*- und die *Vulkan*-Variante ab. Die *OpenGL*-Variante hat eine mittlere *Frametime* von 0,33 ms, was 3.030 *FPS* entspricht. *Vulkan* hat eine mittlere *Frametime* von 0,25 ms. Das sind 4.000 *FPS*. Die *Vulkan*-Variante ist damit rund 25% schneller, als die *OpenGL*-Variante. In Diagramm 4.2.1.2 sind die Graphen zu dieser Messung zu sehen.

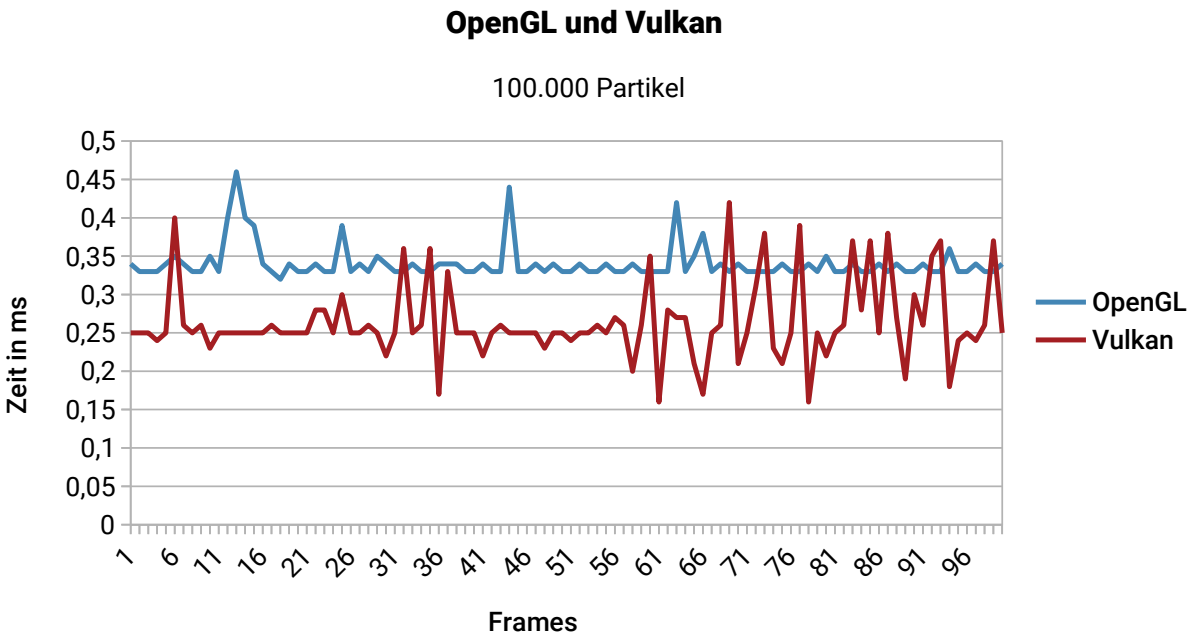

Schaubild 4.2.1.2: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 100.000 Partikeln

4.2.2 Zweite Messung

Die zweite Messung findet mit 1 Millionen Partikeln statt. Die *CPU*-Variante wird hiefür nicht mehr verwendet. Aus dieser Messung ergeben sich durchschnittlich 0,76 ms (1.316 *FPS*) für *OpenGL* und 0,58 ms (1724 *FPS*) für *Vulkan*. Hier ist der Performancevorteil von *Vulkan* bei 24%.

Die Messergebnisse lassen sich aus Diagramm 4.2.2.1 ablesen.

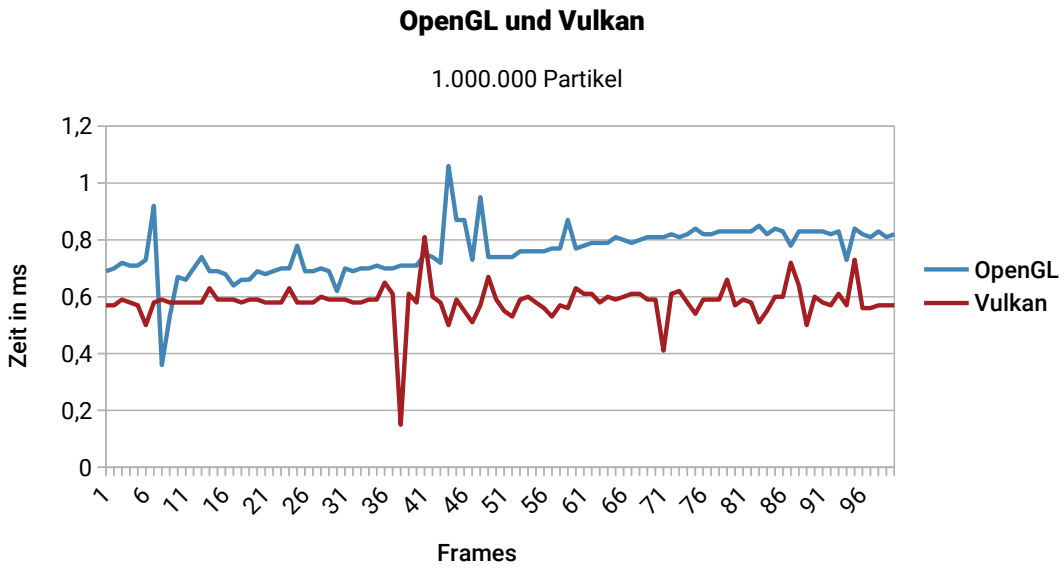

Schaubild 4.2.2.1: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 1.000.000 Partikeln

4.2.3 Dritte Messung

Die dritte Messung wird mit 10.000.000 Partikeln durchgeführt. Hierbei ist die mittlere *Frametime* von *OpenGL* 5,92 ms (169 *FPS*) und von *Vulkan* 4,41 ms (227 *FPS*). Bei 10 Millionen Partikeln ist der Vorsprung von *Vulkan* bei 26%.

Diagramm 4.2.3.1 zeigt auch hier die Messergebnisse im Detail.

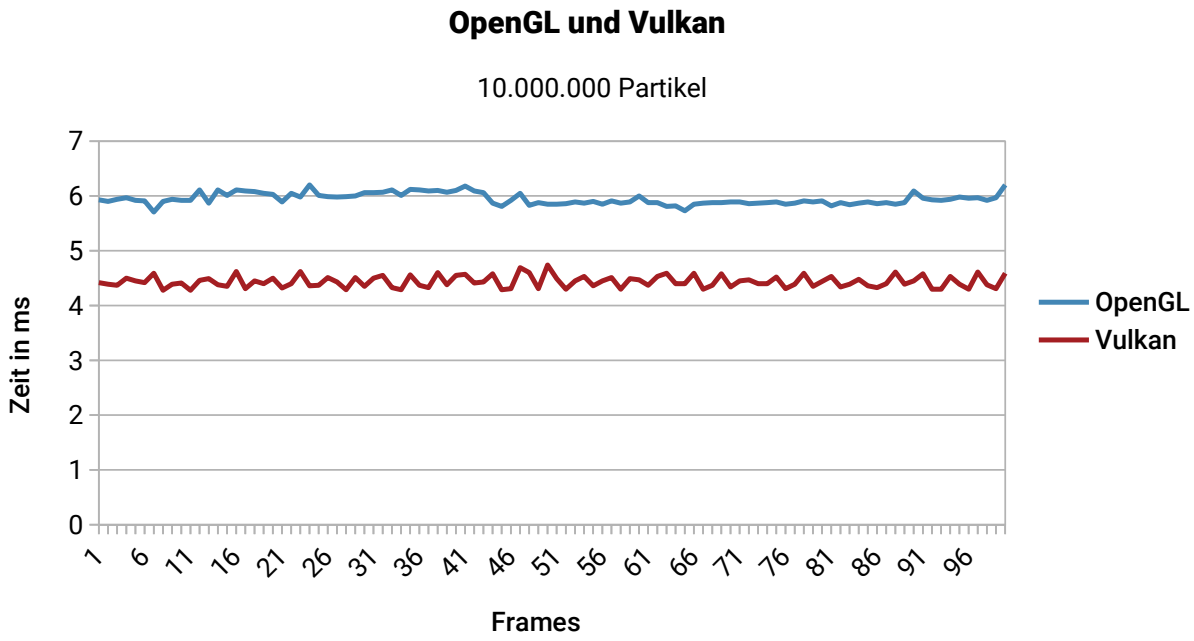

Schaubild 4.2.3.1: Mittlere Frametimes der OpenGL- und Vulkan-Variante bei 10.000.000 Partikeln

4.2.4 Messergebnisse

Die Messergebnisse der Messung sind Tabelle 4.2.4.1 nochmals zusammen dargestellt.

Tabelle 4.2.4.1: Die Messergebnisse zusammengefasst

Partikel	OpenGL		Vulkan		CPU	
	Mittlere Frametime (ms)	Mittlere FPS	Mittler Frametime (ms)	Mittlere FPS	Mittler Frametime (ms)	Mittlere FPS
100.000	0,33	3.030	0,25	4.000	59,11	17
1.000.000	0,76	1.316	0,58	1.724	-	-
10.000.000	5,92	169	4,41	227	-	-

In jeder Messung erreicht die *Vulkan*-Variante eine durchschnittlich 25% bessere Performance gegenüber *OpenGL*. Dieser bessere Performance zeigt sich konstant von Messung 1 bis Messung 3 und auch weitere Messungen würden ähnliche Ergebnisse erzielen.
tbd

5 Fazit und Ausblick

Folgender Abschnitt beschäftigt sich mit dem Fazit des Autors und gibt einen Ausblick darauf, was weiter auf diese Arbeit aufbauen kann oder wie man diese Arbeit erweitern kann.

5.1 Fazit

Vulkan ist eine neue Grafik-API, die den Entwickelnden das Ruder in die Hand gibt. Im Gegensatz zu *OpenGL* sind die Entwickelnden dabei gerade am Anfang auf sich alleine gestellt und müssen Teile implementieren, die sonst der *OpenGL* Treiber erledigt hätte. Doch diese Freiheit stößt bei vielen auf Zuspruch, wenn man sich Artikel und Foren zu *Vulkan* durchliest. Zum Lernen ist *Vulkan* aufgrund der Freiheit nicht einfach. Ebenso benötigt man sehr viel mehr Quellcode, um erste Erfolge sehen zu können. Wer also ein kleines Projekt schnell umsetzen möchte, der ist mit *Vulkan* vermutlich schlechter dran, als mit *OpenGL*. Doch wer bessere Performance oder mehr Freiheit über die Grafikkarten haben möchte, der sollte unbedingt zu *Vulkan* greifen. In diesem Projekt wurde durchschnittlich eine 25% bessere Performance von *Vulkan* gegenüber *OpenGL* gemessen. Ob der Zusatzaufwand bei der Implementierung diese 25% rechtfertigen müssen die Entwickelnden selbst entscheiden. Je nach Projekt kann eine schnelle Umsetzung oder eben bessere Performance benötigt werden. Auch sei angemerkt, dass sich diese 25% natürlich auf Partikelsysteme im Speziellen, noch spezieller das hier in der Arbeit verwendete Partikelsystem, bezieht.
Auf jeden Fall steht fest, dass es sich lohnt Einblicke in die Welt von *Vulkan* zu erhalten.

5.2 Ausblick

Diese Arbeit bezog sich hauptsächlich auf Partikelsysteme und deren Simulation auf der Grafikkarte. Für die Simulation wurden *Compute-Shader* verwendet, während das *Rendering* klassisch mit der Grafikpipeline umgesetzt wurde. In dieser Arbeit wurde die Kombination aus *Computing* und *Rendering* gemessen, nicht jedoch nur der Teil, der sich auf *Computing* bezieht. Die reinen *Compute* Möglichkeiten oder die reine *Compute*

Performance von *Vulkan* kann aufbauend auf dieser Arbeit weiter untersucht werden. Hierbei kann entweder weiterhin *OpenGL* als Vergleichs-API dienen oder eine andere, wie *DirectX*. Ebenso kann erweiternd zu dieser Arbeit eine andere Grafik-API als *OpenGL* mit *Vulkan* verglichen werden.

Literaturverzeichnis

[CONITEC] Was ist ein Partikel-System?, http://www.conitec.net/german/gstudio/faq.php#was5, [Aufgerufen am 06.01.2020]

[VULKAN] Vulkan Tutorial - Introduction, https://vulkan-tutorial.com/, [Aufgerufen am 06.01.2020]

[Reeves 1983] William T. Reeves; Particle Systems A Technique for Modeling a Class of Fuzzy Objects , 1983

[Orlamünder & Mascolus 2004] Dieter Orlamünder, Wilfried Mascolus; Computergrafik und OpenGL - Eine systematische Einführung , 2004, Fachbuchverlag Leipzig im Carl Hanser Verlag, TU Dresden, ISBN: 3-446-22837-3

[Pätzold 2005] Philipp Pätzold; Entwicklung eines Partikelsystems auf Basis moderner 3D-Grafikhardware, 2005

[Brodtkorb u.a. 2013] André R. Brodtkorb, Trond R. Hagen, Christian Schulz, Geir Hasle; GPU computing in discrete optimization. Part I: Introduction to the GPU, 2013, EURO Journal on Transportation and Logistics, https://doi.org/10.1007/s13676-013-0025-1

[Segal & Akeley 2019] Mark Segal, Kurt Akeley; The OpenGL Graphics System: A Specification, 2019

[KHRONOS] About The Khronos Group, https://www.khronos.org/about/, [Aufgerufen am 31.03.2020]

[VULKANSPEC] Vulkan 1.2.135 - A Specification, https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf, []

[GPUOPEN] V-EZ brings “Easy Mode” to Vulkan, https://gpuopen.com/v-ez-brings-easy-mode-vulkan/, [Aufgerufen am 31.03.2020]

[Marsaglia 2003] George Marsaglia; Random Number Generators, 2003, Journal of Modern Applied Statistical Methods, Florida State University

[Park 1992] Robert E. Park, Software Size Measurement: A Framework for Counting Source Statements, 1992, Software Engineering Institute, Carnegie Mellon University

[NVIDIA] Transitioning from OpenGL to Vulkan, https://developer.nvidia.com/transitioning-opengl-vulkan, [Aufgerufen am 01.04.2020]

[Brunst & Mueller 2011] Holger Brunst, Matthias S. Mueller, Performance Analysis of Computer Systems: Requirements, Metrics, Techniques and Mistakes, 2011, Center for Information Services & High Performance Computing, TU Dresden