Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Visualisierung großer Straßengraphen mittels Vulkan

Sabri Bektas

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
Beginn am:	7. Mai 2019
Beendet am:	7. November 2019

Abstract

Das Ziel dieser Forschung ist es einen dreidimensionalen Straßengraphen in Globusdarstellung auf Basis der Vulkan Grafikkarten API zu erstellen. Die Vulkan API ist eine Computergrafikschnittstelle, die einen plattformunabhängigen Zugriff auf moderne GPUs ermöglicht. Als Grundlage für die Straßendaten dient die Geodatenbank OpenStreetMap, welche aus Längen - und Breitengrad aufgebauten Knotendaten liefert. Mithilfe dieser Datenform wird eine Globusdarstellung ermöglicht. Der auf Vulkan basierte Straßengraph bietet eine dynamische Änderung des angezeigten Graphen. Es können Straßenkategorien, wie zum Beispiel Autobahnen oder Landstraßen, ein- und ausgeblendet werden.

Zuerst werden verwandte Arbeiten zu dem Vulkan basierten Straßengraphen vorgestellt. Nach einem allgemeinen Überblicks auf Straßengraphen, werden die Grundlagen dieses Projekts nahegelegt. OpenStreetMap und ihre grundlegende Struktur der Daten bilden das erste Fundament der Grundlagen. Hier soll verdeutlicht werden, welche Daten für einen Straßengraphen von Bedeutung sind. Im Anschluss wird die Vulkan API analysiert, indem die spezifischen Vulkan Objekte erklärt werden. Nach dem vermittelten Grundwissen folgt die Offenlegung des Vulkan basierten Straßengraphen von der Entwicklungsumgebung über den verwendeten Bibliotheken bis hin zum Benchmarking des Programms.

Auf dieser Grundlage ist es empfehlenswert die Vulkan API als Basis für Straßengraphen zu wählen. Die hardwarenahe Programmierung erfordert ein stabiles Grundwissen, kann jedoch für eine gesteigerte Effizienz in der Darstellung sorgen. Weiterführende Implementierung könnte sich mit Text- oder Flächendarstellung beschäftigen oder eine Überführung zu mobilen Endgeräten realisieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface. 13

DÖS Deutschland-Österreich-Schweiz. 50

FMI Institut für Formale Methoden der Informatik. 13

FPS Frames Per Second. 51

GLEW OpenGL Extension Wrangler Library. 18

GLFW Graphics Library Framework. 5, 35

GLSL OpenGL Shading Language. 32

GPU graphics processing unit. 25

HLSL High Level Shading Language. 32

JSON JavaScript Object Notation. 21

KDE K Desktop Environment. 15

ODbL Open Data Commons Open Database License. 19

OSM OpenStreetMap. 5

OSMF OpenStreetMap Foundation. 19

PBF Protocolbuffer Binary Format. 20

SDK Source Development Kit. 35

SDL Simple DirectMedia Layer. 41

SPIR-V Standard Portable Intermediate Representation - V. 32

VbS Vulkan basierter Straßengraph. 35

VRAM Video Random Access Memory. 42

WSI Window System Interface. 36

XML Extensible Markup Language. 20

1 Einleitung

Motivation

Straßengraphen sind digitale Informationen, mit denen sich Positionen auf der Erdoberfläche bestimmen lassen. Es ist eine Zuweisung der räumlichen Lage, die durch Metadaten hinsichtlich eines Zeitbezugs oder der Entstehung genauer definiert werden.

In den Forschungsarbeiten am Institut für Formale Methoden der Informatik (FMI) müssen oft große Straßengraphen visuell dargestellt werden. Visualisierungstechniken auf Basis des Webs erlauben überwiegend nur eine Anzeige mit wenig Details des Netzwerks. Aber auch das Pendant zur Vulkan basierten Variante, nämlich der in den verwandten Arbeiten vorgestellten „The Simplest Graph Renderer“ auf Basis von OpenGL, stellt große Straßengraphen nicht flüssig dar. Hierfür soll die Computergrafik-Programmierschnittstelle Vulkan Application Programming Interface (API), welche durch ihre Hardware nahe Schnittstelle eine effiziente Befehlsverarbeitung ermöglicht, Abhilfe schaffen.

Zielsetzung

Ziel dieser Bachelorarbeit ist die flüssige Darstellung von großen Straßengraphen. Der Deutschland Straßengraph hat beispielsweise über 25 Millionen Knoten und über 50 Millionen Kanten. Es soll mit einer effizienten Befehls- und Speicherverarbeitung eine flüssige Darstellung ermöglicht werden werden. Hierbei wird auf die Vulkan API zurückgegriffen. Vulkan ist eine neuere Computergrafik-Programmierschnittstelle, welche plattformübergreifende Anwendungen zulässt. Die Visualisierung basiert auf einer dreidimensionalen Globusdarstellung und ermöglicht eine Differenzierung zwischen den einzelnen Straßenkategorien, wie zum Beispiel Autobahnen oder Bundesstraßen, anhand der Farb- beziehungsweise Strichbreite. Darüber hinaus können diese Straßenkategorien ein- und ausgeblendet werden.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Verwandte Arbeiten: Das zweite Kapitel beschäftigt sich mit den verwandten Arbeiten. Es soll einen Überblick zu den aktuellen Straßengraphen geben, die vergleichbar mit dem Vulkan basierten Straßengraphen (VbS) sind.

Kapitel 3 – Grundlagen: In diesem Kapitel werden die Grundlagen behandelt. Zu ihnen zählen OpenStreetMap und Vulkan API. Nach Abschluss dieses Kapitels ist ein Grundwissen über die relevanten OpenStreetMap Daten und den Workflow von den spezifischen Vulkan Objekten vermittel worden, was als Basis für das darauf folgende Kapitel dient.

Kapitel 4 – Vulkan basierter Straßengraph: Das vierte Kapitel beschäftigt sich mit Implementierung und der Evaluation des Straßengraphen. Insbesondere werden hier die im vorherigen Kapitel vorgestellten Vulkan Objekte in ihrem Entstehungsprozess aufgezeigt. Darüber hinaus finden auch die verwendete Entwicklungsumgebung und Bibliotheken ihre Stellung.

Kapitel 5 – Zusammenfassung und Ausblick: Im letzten Kapitel werden die Ergebnisse der Arbeit zusammengefasst und Anknüpfungspunkte vorgestellt.

2 Verwandte Arbeiten

2.1 Marble

Marble ist eine einfache, übersichtliche und aufgabenorientierte virtuelle Globus- und Weltatlas-Software, welche dem Benutzer ermöglicht, zwischen den Planeten beispielsweise der Erde, dem Mond, der Venus, dem Mars und weiteren zu wählen, um diese als 3D-Modell darzustellen. Marble bietet eine einfach einzubettende Bibliothek für Entwickler, welche schnell und plattformübergreifend mit minimalen Hard- und Softwareanforderungen arbeitet.[01][27]

Entwickelt wurde Marble von der K Desktop Environment (KDE) Community. Die KDE ist eine internationale Community, welche freie und Open-Source-Software entwickelt.

Getreu eines realen Atlasses ermöglicht Marble sich frei über der Karte zu bewegen und Orte nachzuschlagen. Darüber hinaus wird eine Zoom Funktion geboten und viele verschiedene Ansichten der Erdoberfläche werden zur Verfügung gestellt. Marble ist sehr flexibel. Über das plattformübergreifende Design hinaus lassen sich die Kernkomponenten problemlos in andere Programme integrieren. Die Software ist so konzipiert, dass sie ohne Hardwarebeschleunigung läuft, sie kann jedoch um OpenGL erweitert werden. Eines der Ziele ist es, dass die Anwendung vergleichsweise schnell gestartet wird. Sie wird mit einem minimalen, aber nützlichen Offline-Datensatz geliefert.[02]

Unterstützung für Online-Mappingquellen wie OpenStreetMap wurden durch Mitwirkende ermöglicht. Marble bietet außerdem Möglichkeiten zur Routenplanung[03]. Ein Navigationsmodus namens MarbleToGo wurde im Rahmen des Google Summer of Code 2010 entwickelt.[04] Dieser wurde später teilweise neu geschrieben und in Marble Touch umbenannt.[05]

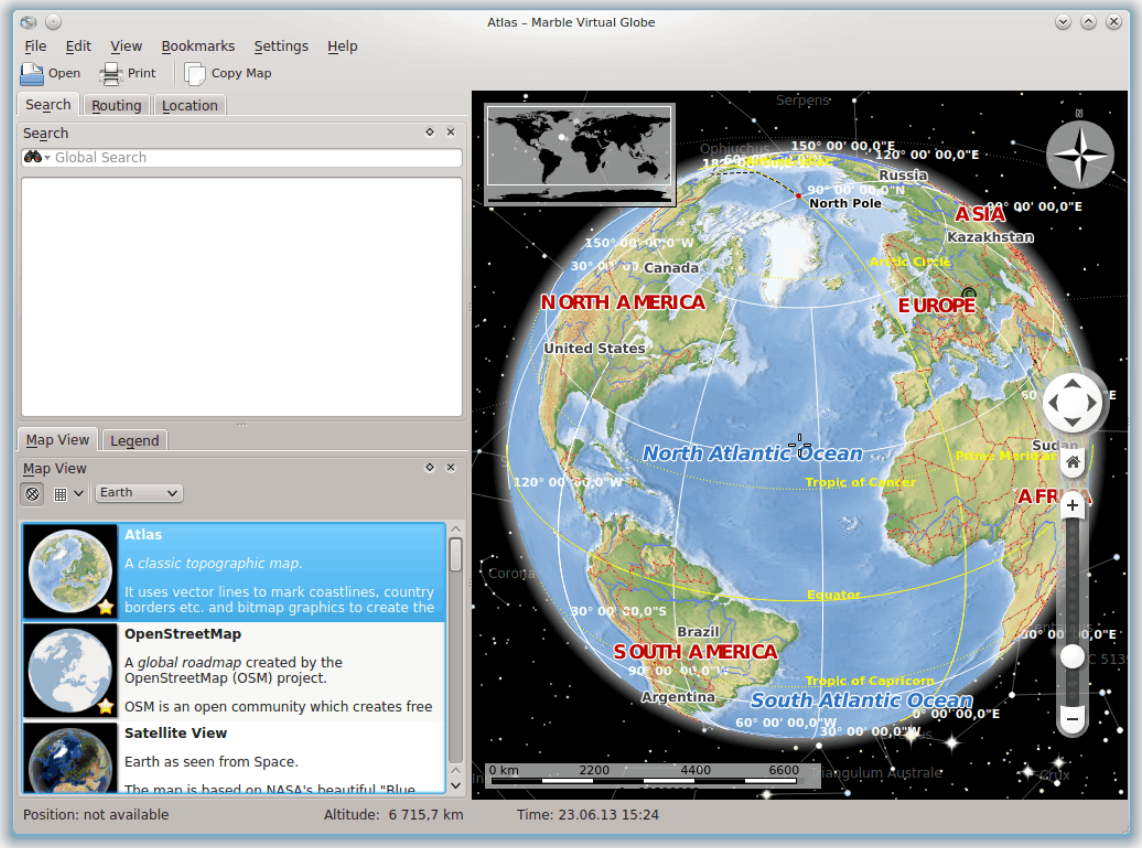


Abbildung 2.1: Marble virtueller Globus– siehe https://docs.kde.org/trunk5/en/kdeedu/marble/quick-1.png

2.2 Leaflet

Leaflet ist die führende Open-Source-JavaScript-Bibliothek für mobile, interaktive Karten. Die Verwendung von HTML5 und CSS3 sichert der Bibliothek die mehrheitliche Unterstützung von Browsern.[06]

Im Jahr 2010 entwickelte Vladimir Agafonkin Leaflet bei der Firma CloudMade unter dem Namen Web Maps API. Die Veröffentlichung der ersten Version erfolgte im Jahr 2011.[8] Leaflet wird seit 2013 über die Firma Mapbox unter der Leitung von Vladimir Agafonkin weiterentwickelt.[07]

Das Projekt wurde mit dem Fokus auf Einfachheit, Leistung und Benutzerfreundlichkeit entwickelt. Es funktioniert effizient auf allen wichtigen Desktop- und mobilen Plattformen und kann mit vielen Plugins erweitert werden.

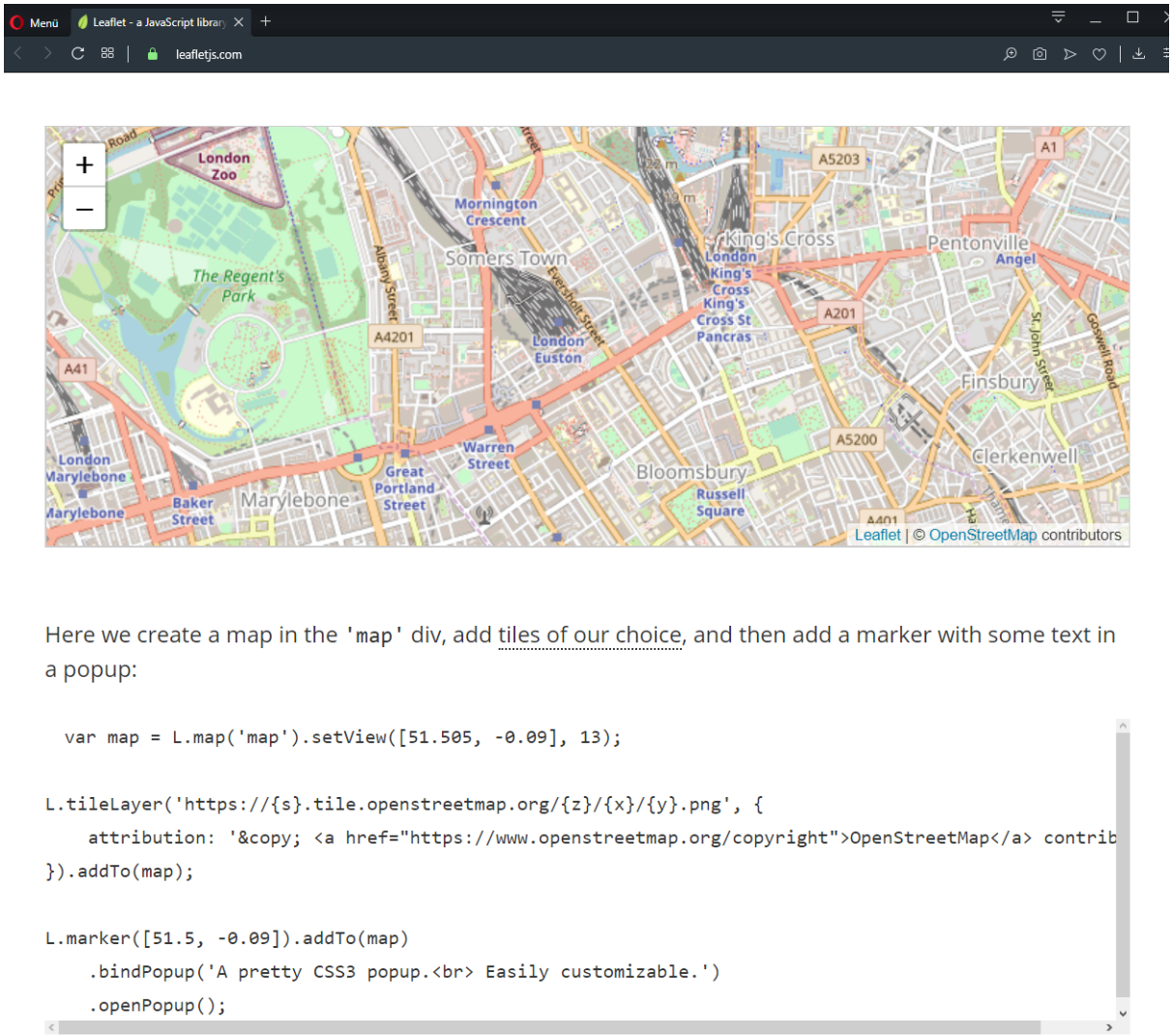


Abbildung 2.2: Leaflet OSM Browser– siehe https://leafletjs.com

Aufgrund der benutzerfreundlichen und gut dokumentierten API greifen viele Entwickler auf Leaflet zurück. Darüber hinaus kann mit wenig Aufwand eine OpenStreetMap Karte in eine Webseite eingebunden werden.[06]

2.3 The Simplest Graph Renderer

Im Jahr 2014 wurde The Simplest Graph Renderer von einem Mitarbeiter des FMI entwickelt. Mit Abhängigkeiten von OpenGL Extension Wrangler Library (GLEW) und GLFW ist es möglich OpenStreetMap Straßen auf einem Globus zu rendern. The Simplest Graph Renderer basiert auf OpenGL und benötigt einen Kartenausschnitt, um die gewünschten Straßen auf der Weltkugel darstellen zu können. Bei dem Rendering des kompletten Deutschland Straßennetzes mit einem System bestehend aus 16 GigaByte Arbeitsspeicher, einem Prozessor vergleichbar mit einer Ryzen 2600 und einer Grafikkarte vergleichbar mit AMD Radeon 580, ist die Bewegung im Raum nicht flüssig.

Der Simple Graph Renderer benutzt GLFW, um die Ergebnisse auf dem Bildschirm zu präsentieren. Der Vulkan basierte Streetviewer ist eine Homage an den Simple Graph Renderer, mit dem unterschied der verwendeten Grafikbibliotheken Vulkan und OpenGL.

3 Grundlagen

3.1 OpenStreetMap

3.1.1 Allgemeines

OSM ist ein Projekt, in dem freie geographische Daten und Kartenmaterial gesammelt werden. Im Jahr 2004 wurde das Projekt ins Leben gerufen, mit der Absicht eine Straßenkarte abzubilden. Mittlerweile ist OSM die größte freie Geodatenbank der Welt.[08]

Die immense Datenbank kann von jedem ergänzt werden. Am einfachsten funktioniert dies über ein GPS-Gerät, welches die Daten aufzeichnet. Ein GPS-Gerät ist jedoch nicht unbedingt notwendig, um etwas zu der OSM Datenbank beizusteuern. Weitere Möglichkeiten etwas dazu beizutragen, sind das Abzeichnen von Luftbildern von Bing oder das Zeichnen von Straßen mit einem Editor.[09][10]

Die Erstellung von Karten ist nur eine Funktionalität von OSM. Darüber hinaus ist die Navigation, aber auch die Erstellung von Kartenmaterial für Navigationsgeräte mit OSM möglich. Außerdem kann OSM umgebungs- und ortspezifische Suchanfragen beantworten, wie zum Beispiel „Wo befindet sich das nächste italienische Restaurant in meiner Nähe?“. [10]

Das Benutzen und die Weiterverarbeitung der Daten ist lizenkostenfrei, solange die OSM und seine Mitwirkenden angegeben werden.[11] Die Daten sind unter der Open Data Commons Open Database License (ODbL) von der OpenStreetMap Foundation (OSMF) lizensiert. OSMF ist eine internationale gemeinnützige Organisation, die das OpenStreetMap-Projekt unterstützt, aber nicht kontrolliert. Die OSMF widmet sich der Förderung des Wachstums, der Entwicklung und der Verbreitung kostenloser Geodaten und der Bereitstellung von Geodaten für jedermann.[12]

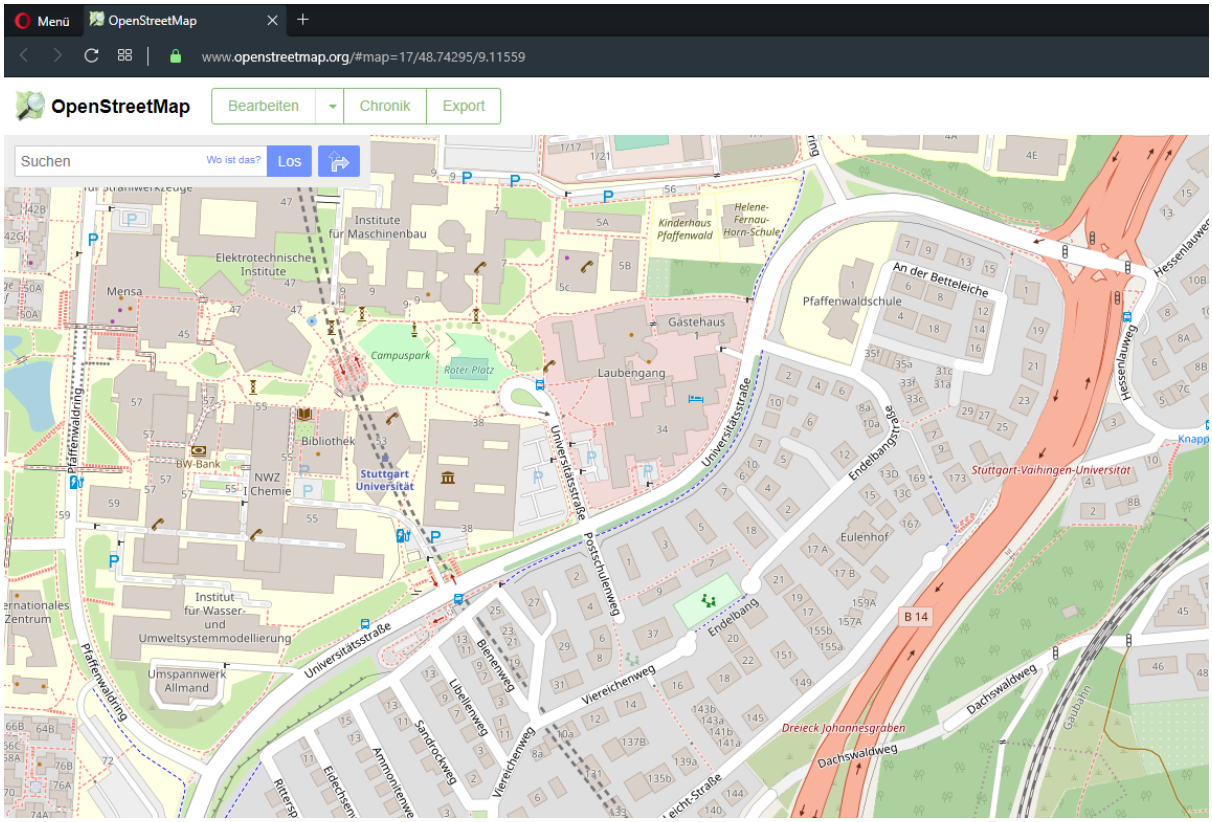


Abbildung 3.1: OpenStreetMap Kartenausschnitt der Universität Stuttgart-Vaihingen

3.1.2 OSM Daten

Formate der OSM Daten

Daten können in vielen verschiedenen Formaten gespeichert werden. Bei Bilddateien gibt es beispielsweise das JPEG, das JPG, das BMP Format und viele weitere Formate. So gibt es auch bei OSM Daten unterschiedliche Datenformate. Die gängigsten unter ihnen sind folgende: [13]

- Das OSM Extensible Markup Language (XML) Format

  XML ist ein sogenanntes Meta-Format. Mittels der Baumstruktur ist das XML-Formats übersichtlich und für den Menschen lesbar. Um die OSM Daten leicht über das Internet transferieren zu können, wurde das OSM XML Format entwickelt.[14]

- Das Protocolbuffer Binary Format (PBF) Format

  Das Binärformat, das auf der Kodierung der Protokollpuffer basiert, ist das kompakteste Format. Es sticht durch seine hohe Speichererffizienz und Lese- und Schreibgeschwindigkeit hervor.

- Das o5m Format

  Wie das PBF Format ist das o5m Format ebenfalls ein binäres Datenformat. Es ist nicht so speichereffizient, wie das PBF Format, jedoch lassen sich hiermit die Daten schneller verarbeiten.

- Das Overpass JavaScript Object Notation (JSON) Format

  Dieses Format ist eine Ausgabeform der Overpass API für OSM Daten. Mit der API lassen sich selektive Downloads aus der OSM Datenbank ableiten. Diese können im JSON für den Menschen lesbar eingesehen werden.

Für diese Arbeit wurde das PBF Format verwendet. Aufgrund der Tatsache, dass das PBF Format eine Komprimierung des OSM XML Formats darstellt und für den Menschen unlesbar ist, wird im folgenden das OSM XML Format hingehend seiner Struktur anhand einer Beispiel-Datei genauer analysiert.

Struktur der OSM Daten

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900"
    maxlon="12.2524800"/>
 <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO"
    uid="46882" visible="true" version="1" changeset="676636"
    timestamp="2008-09-21T21:37:45Z"/>
 <node id="1831881213" version="1" changeset="12370172" lat="54.0900666"
    lon="12.2539381" user="lafkor" uid="75625" visible="true"
    timestamp="2012-07-20T09:43:19Z">
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
 </node>
 ...
 <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO"
    uid="46882" visible="true" version="1" changeset="676636"
    timestamp="2008-09-21T21:37:45Z"/>
 <way id="26659127" user="Masch" uid="55988" visible="true" version="5"
    changeset="4142606" timestamp="2010-03-16T11:47:08Z">
  <nd ref="292403538"/>
  ...
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Strae"/>
 </way>
```

```
 <relation id="56688" user="kmvar" uid="56190" visible="true" version="28"
    changeset="6947637" timestamp="2011-01-12T14:23:49Z">
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="node" ref="364933006" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  <member type="node" ref="249673494" role=""/>
  <tag k="name" v="Kstenbus Linie 123"/>
  <tag k="network" v="VVW"/>
  <tag k="ref" v="123"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
 </relation>
 ...
</osm>
```

Im Wesentlichen sind die OSM Daten in drei elementare Gruppen unterteilt: den `Nodes`, den `Ways` und den `Relations`. Nodes sind Knoten oder vielmehr Punkte auf der Erdoberfläche und zeigen an, wo genau sich beispielsweise ein Straßenknoten oder ein Baum befindet. Straßen und Flüsse werden mithilfe von Ways dargestellt und entsprechen einer Folge von Nodes. Eine Relation ist multifunktional und kann zum Beispiel eine Busroute darstellen. Jedes Element kann Attribute und Tags zur genaueren Beschreibung enthalten.

Das Element Node

Die Nodes sind die Kernelemente des OSM Datenmodells. Sie stellen einen bestimmten Punkt auf der Erdoberfläche dar, der durch seinen Längen- und Breitengrad und seiner Node-ID definiert ist. Eine optionale Dimension der Höhe (engl. Elevation) kann ebenfalls durch den Key `ele` einbezogen werden.

Insbesondere lassen sich mit Nodes eigenständige Punkte definieren. Eine Node kann beispielsweise eine Parkbank oder einen Wasserbrunnen darstellen. Neben eigenständigen Punkten werden Nodes auch verwendet, um die Form eines Weges zu definieren. Mit Tags können den Nodes bestimmte Eigenschaften zugewiesen werden. Verkehrsampeln werden beispielsweise mit dem Tag `highway=traffic_signal` symbolisiert. [15]

Das Element Way

Ein Way besteht aus mindestens einer Linie. Die Verbindung aus zwei Nodes bezeichnen hierbei eine Linie. Es ist eine geordnete Liste aus Nodes, die sequentiell verbunden sind.

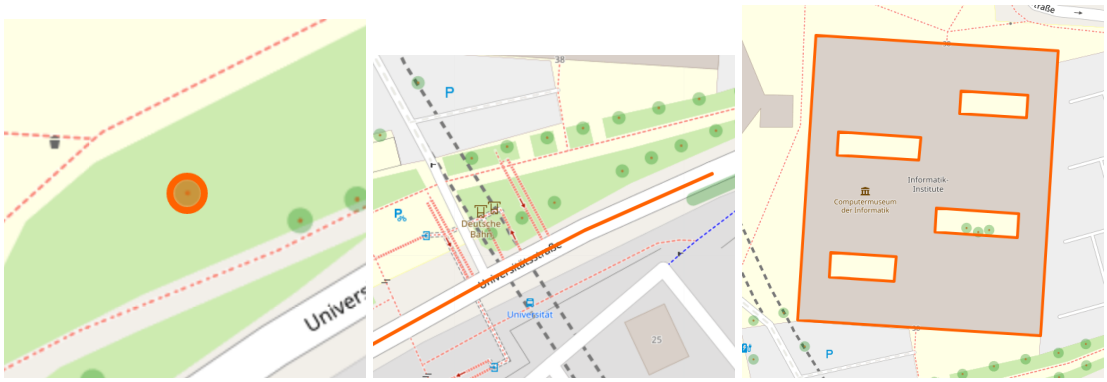
Wege können auch die Grenzen von Flächen wie Gebäude oder Wälder darstellen. Die Besonderheit in diesem Fall ist, dass der erste und letzte Knoten des Ways gleich sind. Dies wird als „closed way“ bezeichnet[16]. Kreisverkehre unter anderem sind auch closed ways, jedoch stellen diese keine Flächen dar. Entscheidend sind die Tags in solchen Fällen, die dann eine Straße symbolisieren. [16]

Das Element Relation

Eine Relation ist eine multifunktionale Datenstruktur, die eine Beziehung zwischen zwei oder mehreren Datenelementen dokumentiert. Beispiele sind unter anderem:

- Eine Routenbezeichnung für Wege, die beispielsweise einen Fahrradweg oder eine Busroute bilden.
- Ein Umkehrverbot, das das Umkehren beziehungsweise Wenden auf bestimmten Straßen verbietet.
- Ein Multi-Polygon, um Gebiete zu markieren, dessen Grenzen als „outer way“ und Teilgebiete im inneren als „inner way“ bezeichnet werden. Ein Beispiel hierfür ist ein See mit einer Insel im Inneren.

Erheblich für die Definition der Relation ist der Tag `Type`. Dieser beschreibt worum es sich in dieser Relation handelt. Jedes Element kann innerhalb der Rolle ebenfalls Tags enthalten, die zum Beispiel das Umkehrverbot genauer beschreiben mit den Tag-Mitgliedern „from“ und „to“.[17]



(a) Node **(b)** Way **(c)** Relation

Abbildung 3.2: Veranschaulichung der Grund Elemente in OpenStreetMap

Tags

Die Tags beschreiben die Bedeutung des jeweiligen Elements und können an alle Elemente, also den Nodes, Ways und Relations, gebunden werden.

Aufgebaut sind die Tags aus zwei Textfeldern, einem „key“ und einem „value“. Es handelt sich jeweils um Unicode-Strings mit bis zu 255 Zeichen. `highway=residential` ist ein Tag für Wohnstraßen, die den Zugang zu dem Häusern ermöglichen. Die Keys müssen eindeutig sein, das heißt zwei Tags dürfen nicht den selben Key haben.

Es gibt keine feste Richtlinien für Tags, aber es gibt viele Konventionen auf der OpenStreetMap Wiki. Mithilfe des „taginfo “ Systems (Link: https://taginfo.openstreetmap.org/) kann nach Tags gesucht werden, um deren Bedeutung für eine mögliche Anwendung zu verstehen.

Attribute

Die OSM Datenbank speichert Attribute für Nodes, Ways und Relations. Folgende Tabelle zeigt alle OSM Attribute auf.

Name	Value	Beschreibung
id	integer	Wird verwendet um ein Element eindeutig zu bestimmen. Jede Elementgruppe hat ihren eigenen ID Raum, so kann ein Node, als auch ein Way die id=10 haben.
user	character String	Zeigt den Usernamen des letzten Bearbeiters des Elements an.
uid	integer	Der numerische Identifier des letzten Bearbeiters.
timestamp	W3C standard date and time formats	Zeitpunkt der letzten Modifikation (z.B. „2016-12-31T23:59:59.999Z“)
visible	„true“ oder „false“	Gibt an, ob das Objekt in der Datenbank gelöscht wird oder nicht. Wenn visible=„false“, dann sollte das Objekt nur von History-Aufrufen zurückgegeben werden.
version	integer	Die Bearbeitungsversion des Objekts. Neu erstellte Objekte beginnen bei Version 1 und der Wert wird vom Server erhöht, wenn ein Client eine neue Version des Objekts hochlädt.
changeset	integer	Die Nummer des Changesets, in dem das Objekt erstellt oder aktualisiert wurde. Ein Changeset besteht aus einer Gruppe von Änderungen, die ein einzelner Benutzer über einen kurzen Zeitraum hinweg vorgenommen hat. Ein Änderungssatz kann beispielsweise das Hinzufügen neuer Elemente und Tags, das Ändern von Tag-Werten von Elementen, das Löschen von Tags und auch das Löschen von Elementen beinhalten.

Tabelle 3.1: Attributtabelle [18]

3.2 Vulkan API

3.2.1 Khronos Group

Die Khronos Group ist ein offenes Industriekonsortium von über 150 führenden Hard- und Softwareunternehmen, die fortschrittliche, lizenzfreie Beschleunigungsstandards für 3D-Grafiken, Augmented und Virtual Reality, Vision und Machine Learning entwickeln. Zu den Khronos-Standards gehören OpenGL®, OpenGL® ES, OpenGL® SC, WebGL™, SPIR-V™, OpenCL™, SYCL™, OpenVX™, NNEF™, COLLADA™, OpenXR™, 3D Commerce™ und glTF™. Die Vulkan API ist ebenfalls ein Standard, das von der Khronos Group entwickelt wird.

Die Mitglieder von Khronos sind in der Lage, zur Entwicklung der Khronos-Spezifikationen beizutragen. Außerdem sind sie befugt, in verschiedenen Phasen vor dem öffentlichen Einsatz eines Projekts abzustimmen. Des Weiteren können sie die Bereitstellung ihrer hochmodernen beschleunigten Plattformen und Anwendungen durch einen frühen Zugang zu Spezifikationsentwürfen und Konformitätstests beschleunigen. Der Konformitätstest ist ein Testverfahren, mit dem festgestellt wird, ob eine Implementierung mit den Spezifikationen des Standards übereinstimmt. [26]

Abbildung 3.3: Standards der Khronos Group [26]

3.2.2 Entstehung der Vulkan API

Viele Grafik APIs sind in einer Zeit entstanden, in der die Grafikhardware auf konfigurierbar feste Funktionen beschränkt war. Vertex-Daten mussten von Programmierern in einen Standardformat eingebunden werden und waren in Bezug auf Beleuchtungs- und Schattierungsmöglichkeiten von dem Grafikprozessor (engl. graphics processing unit (GPU)) abhängig.

In Folge der stetigen Weiterentwicklung in der Grafikkartenarchitektur wurden immer mehr programmierbare Funktionen angeboten. Dadurch ergab sich die Notwendigkeit, bereits bestehende APIs um neuere Funktionen zu erweitern. Auf der Grafiktreiberseite musste den Anforderungen und Absichten der Programmierer nachgekommen werden. Aufgrund dessen gibt es häufige Treiber Updates zur Verbesserung der Leistung. Darüber hinaus gab es in den letzten Jahren einen Anstieg von mobilen Geräten mit leistungsfähiger Grafikhardware. Je nach

Energie- und Platzbedarf haben diese mobilen GPUs unterschiedliche Architekturen. Das Tiled Rendering ist ein Beispiel für so eine Architektur. In ihr ist der Prozess der Unterteilung eines Computergrafikbildes durch ein Raster und das getrennte Rendern dieser Rasterabschnitte beschrieben. Die Menge an Speicher und Bandbreite im Vergleich zu Systemen, in welchen die gesamten Frames auf einmal gezeichnet werden, ist stark reduziert. Eine weitere Einschränkung, die sich aus dem Alter dieser APIs ergibt, ist die begrenzte Multi-Threading-Unterstützung, die zu einem Engpass auf der CPU-Seite führen kann.

Vulkans Entwicklung basiert seit dem Entstehungsprozess auf modernen Grafikarchitekturen. Der Treiberaufwand wird durch die umfangreiche API verringert, indem der Programmierer seine Absichten spezifizieren kann. Vulkan verwendet ein standardisiertes Bytecode Format mit einem einzigen Compiler. Dadurch werden Inkonsistenzen bei der Shader Kompilierung vermieden. Schließlich erkennt es die universellen Verarbeitungsmöglichkeiten moderner Grafikkarten an, indem es die Grafik- und Rechenfunktionen in einer einzigen API vereint. [1919]

3.2.3 Vulkan Objekte

Vulkan Objekte werden mit dem Präfix `Vk` gekennzeichnet, wohingegen Vulkan Funktionen mit dem Präfix klein `vk` beschrieben sind. Die Objekt Typen sind wie Handles zu behandeln und werden von Funktion zu Funktion weiter gereicht. Wenn die Objekte nicht mehr benötigt werden, werden sie zerstört.[2419]

Das folgende Diagramm in Abbildung 3.4 ist in in drei Bereiche unterteilt. Jeder Bereich hat ein Haupt-Objekt: Die `VkInstance`, die `VkPhysicalDevice` und die `VkDevice`, in dem Diagramm sind diese rot hinterlegt. Diese Bereiche spiegeln die logische und reale Hardwareebene wieder. Objekte mit grünem Hintergrund haben keine eigenen Typen, sondern werden durch einen numerischen Index vom Typ `uint32_t` in ihrem darüber liegenden Objekt dargestellt.

Die Reihenfolge der Erstellung wird durch die Abfolge von durchgehenden Linien mit Pfeilen dargestellt. Beispielsweise wird die Queue-Familie vor der `VkQueue` erzeugt. Kompositionen beziehungsweise Abhängigkeiten werden durch Linien mit Rauten dargestellt. Diese Objekte müssen nicht zusätzlich erstellt werden, weil sie im Elternobjekt enthalten sind. Das `VkPhysicalDevice` Objekte wird beispielsweise aus einem `VkInstance` Objekt abgeleitet. Durch gestrichelte Linien werden andere Beziehungen dargestellt wie zum Beispiel das Senden verschiedener Befehle an einen `VkCommandBuffer`.[25]

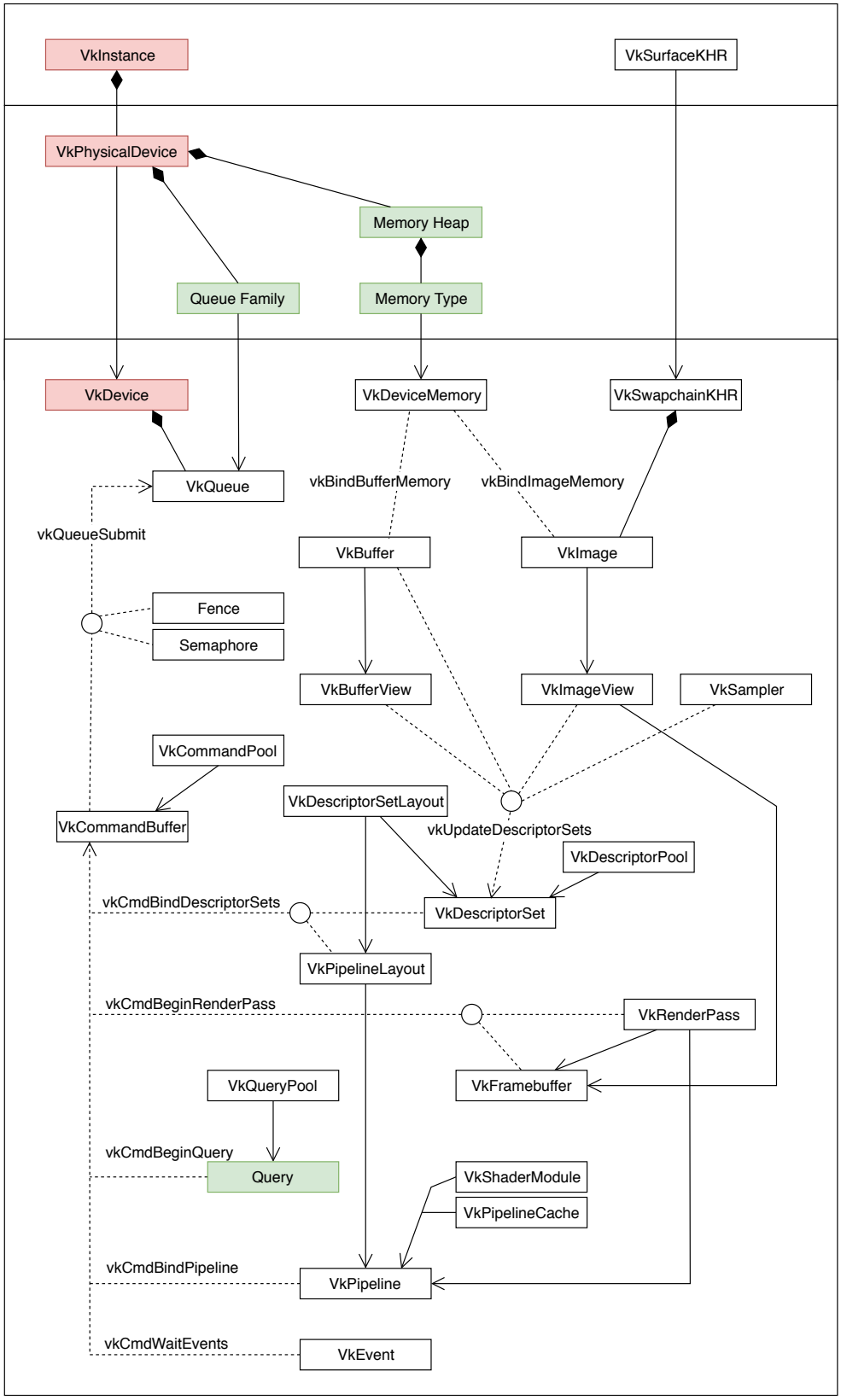


Abbildung 3.4: Vulkan Objekte [25]

VkInstance

Die `VkInstance` ist das erste zu erstellende Objekt. Es stellt die Verbindung von der Anwendung und der Vulkan Laufzeit dar und ist daher nur einmal in der Anwendung vorhanden. Zusätzlich werden anwendungsspezifische Informationen für die Verwendung von Vulkan gespeichert. Deshalb ist es erforderlich, beim Anlegen einer Instance, alle Erweiterungen und Layer anzugeben, um diese zu aktivieren.

Wie bereits erwähnt ist die Vulkan API auf einen minimalen Treiberaufwand ausgelegt. Dieses Ziel wird unter anderem durch eine begrenzte Fehlerprüfung erreicht. Fehler, wie das Übergeben von Nullzeigern an erforderliche Parameter, werden nicht explizit behandelt. Das Abstürzen des Programms oder ein Fehlverhalten sind die Folgen.

Allerdings bedeutet das nicht, dass keine Fehlerüberprüfungen hinzugefügt werden können. Hierfür bietet Vulkan das System der „Validation Layer“ an. Die Aktivierung der Validation Layer erfolgt durch die obige Angabe bei der Erstellung der Instance. Validation Layer sind optionale Komponenten, die in Funktionsaufrufe eingebunden werden, um zusätzliche Operationen ausführen zu können. Gängige Operationen sind:

- Überprüfen, ob Parameterwerte der Spezifikation entsprechen.
- Verfolgung der Erstellung und Zerstörung von Objekten, um Ressourcenverluste zu finden.
- Protokollieren von Aufrufen und deren Parameter.[25][2419]

VkPhysicalDevice

Die `VkPhysicalDevice` beschreibt eine Vulkan-kompatible Hardware Komponente des zugrundeliegenden Systems. Diese werden aus VkInstance aufgelistet und können nach ihren Eigenschaften abgefragt werden, wie die Hardware-ID oder den unterstützten Features.

Darüber hinaus kann die VkPhysicalDevice verfügbare Typen der Queue-Familie aufzählen. Die bedeutsamste unter ihnen ist die „Graphics“ Queue, außerdem gibt es noch die „Compute“ und „Transfer“ Queue-Familien Typen. Jede Queue kann nur bestimmte Arten von Operationen ausführen. Graphic Queues können Grafikpipelines ausführen, die durch `vkCmdDraw`-Befehle gestartet werden. Compute Queues können Compute Pipelines ausführen, die durch `vkCmdDispatch` gestartet wurden. Transfer Queues können Übertragungsvorgänge von `vkCmdCopy` aus durchführen.

Zudem kann die VkPhysicalDevice Memory-Heaps und Speichertypen auswerten. Ein Memory-Heap repräsentiert einen bestimmten Bereich im Arbeitsspeicher. Es kann den System Arbeitsspeicher auf dem Motherboard oder einen bestimmten Speicherplatz auf im Video Arbeitsspeicher auf einer dedizierten Grafikkarte abstrahieren. Bei der Zuweisung von Speicher muss der Speichertyp angegeben werden.[25][2419]

VkDevice

Die `VkDevice` ist als logisches Gerät zu betrachten. Sie ist das Fundament und stellt ein initialisiertes Vulkan Gerät dar. Während der VkDevice Erstellung werden die zu verwendenden Features spezifiziert und die Queues mit ihren Queue-Familien und ihrer Quantität angegeben.[25][2419]

VkQueue

Eine Warteschlange von Befehlen wird Queue genannt. Die Hauptarbeit, die der Grafikprozessor leistet, wird durch Zuweisungen von Operationen an den `VkCommandBuffer` und die anschließende Übermittlung an die `VkQueue` vollbracht. Werden mehrere Queue Family Typen vewendet, wie zum Beispiel eine Compute Queue und eine Graphics Queue, so lassen sich verschiedene CommandBuffer zuweisen. Auf diese Weise lassen sich bei richtiger Anwendung durch die asynchrone Ausführung der Befehle die Prozesse wesentlich beschleunigen.[25][2419]

VkCommandPool und VkCommandBuffer

Die Zuweisung von `VkCommandBuffer` werden von `VkCommandPool` durchgeführt. Sie stehen im Zusammenhang mit einer bestimmten Queue-Familie.

Die VkCommandBuffer repräsentieren einen Puffer mit diversen Befehlen, die von einem VkDevice ausgeführt werden. Auf einem VkCommandBuffer werden `vkCmd` Funktionen angewandt. Sie legen die Reihenfolge, den Typen und die Parameter der Anwendungen fest, die ausgeführt werden, wenn die VkCommandBuffer einer VkQueue zugewiesen wird.[25][2419]

VkSampler

`VkSampler` Objekte stellen den Status eines Bild-Samplers dar, mit dem Bilddaten gelesen und Filter und andere Transformationen für den Shader angewendet werden. Sampler sind durch eine Reihe von Zustandsparameter beschrieben, wie zum Beispiel der Filtermodus (Bsp. linear oder nearest) oder der Adressiermodus (Bsp. clamp-to-edge oder clamp-to-border).[25][2419]

VkBuffer und VkImage

Der `VkBuffer` und die `VkImage` sind zwei Ressourcentypen, die den Gerätespeicher belegen. Der VkBuffer ist ein Container für die Größe der binären Daten in Bytes, wohingegen die VkImage eine Menge von Pixeln darstellt. Für die Erstellung einer VkImage sind unter anderem die Dimensionen, ob 1D, 2D oder 3D, das Pixelformat, wie `R8G8B8A8_UNORM` oder

`R32_SFLOAT` anzugeben. Darüber hinaus kann es aus einzelnen Bildern zusammengesetzt sein, da es mehrere Array Layer oder Mipmap Level haben kann. Mipmaps sind vor kalkulierte und verkleinerte Versionen eines Bildes. Jedes Bild ist halb so groß in der Länge und Breite wie das vorherige Bild. Objekte, die von der Kamera aus weiter entfernt stehen, werden ihre Texturen von den kleineren Mip Bildern beziehen. Das Nutzen von kleineren Texturen erhöht die Rendergeschwindigkeit und verhindert Effekte wie den Moiré-Effekt.[25][2419]

VkDeviceMemory

Das Erstellen eines VkBuffers mit einer bestimmten Länge oder das Erstellen einer VkImage mit einer definierten Dimension weist nicht automatisch den Objekten Speicher zu. Hierfür muss ein `VkDeviceMemory` Objekt erstellt werden. Es ist ein Speicherblock eines bestimmten Typs, der auch von der VkPhysicalDevice unterstützt wird. VkBuffer und VkImage werden durch die Funktionen `vkBindBufferMemory` und `vkBindImageMemory` zusammengebunden.[25][2419]

VkSurfaceKHR

Das Konzept der Swapchain ermöglicht das Präsentieren der finalen Bilder auf dem Bildschirm oder im Anwendungsfenster. Das Erstellen einer Swapchain ist plattformabhängig. Nachdem ein Fenster über eine System API initialisert wurde, wird ein `VkSurfaceKHR` angelegt. Dazu wird das VkInstance Objekt sowie einige systemabhängige Paramter benötigt. Unter Windows sind diese zum Beispiel der Instance Handle „HINSTANCE„ und der Fensterhandle „HWND“. Das VkSurfaceKHR Objekt ist die Vulkan Repräsentation eines Fensters.[25][2419]

VkSwapchainKHR

Weiterhin wird die `VkSwapchainKHR` erstellt, die eine Reihe von Bildern darstellt. Die Swapchain ist eine Warteschlange von Bildern, die auf dem Bildschirm präsentiert werden sollen. In Vulkan muss diese explizit angelegt werden. Ihre Aufgabe ist es sicherzustellen, dass das nächste Bild vor ihrer Präsentation fertig gerendert wurde. Immer wenn ein Bild gezeichnet werden soll, muss die Swapchain ein Image zur Verfügung stellen, auf welches dann gerendert wird. Sobald das Image fertig bearbeitet wurde, wird sie an die Swapchain zurück gegeben und anschließend an einem bestimmten Zeitpunkt auf dem Bildschirm angezeigt.[25][2419]

VkBufferView und VkImageView

Über dem VkBuffer und dem VkImage befindet sich die Views Ebene. Ein `VkBufferView` ist ein Objekt das in Abhängigkeit eines bestimmten VkBuffer erstellt wird. Hier kann der Offset und der Bereich während der Erstellung übergeben werden, um die Ansicht auf die VkBuffer

Daten zu beschränken. Analog ist auch die `VkImageView` ein Satz von Parametern, die sich auf ein bestimmtes Bild beziehen. Hier können beispielweise Pixeldaten in ein anderes Format interpretieren werden oder die Ansicht auf einen bestimmten Bereich von Mip Ebenen oder Arrayschichten beschränken.[25][2419]

Deskriptoren

Die Shader greifen auf die Ressourcen VkBuffer, VkImages und VkSampler über die Deskriptoren zu. Deskriptoren sind in Deskriptor Sets zusammengefasst. Zunächst wird ein `VkDescriptorSetLayout` erstellt, das als Vorlage für ein Descriptor Set fungiert. Ein VkDescriptorSetLayout Objekt wird durch ein Array von null oder mehr Desktriptor Bindungen definiert. Jede Deskriptor Bindung wird durch einen Deskriptortyp, eine Anzahl der Deskriptoren in der Bindung, eine Reihe von Shader Stufen und eine Reihe von Sampler Deskriptoren spezifiert.

Des Weiteren wird ein `VkDescriptorPool` Objekt angelegt, wodurch Deskriptor Sets zugeordnet werden. Beim Anlegen eines Deskriptorenpools müssen die maximale Anzahl von Deskriptorsets und Deskriptoren verschiedener Typen angegeben werden.

Zuletzt werden die `VkDescriptorSet` allokiert. Hierfür werden die erstellten VkDescriptorPool und VkDescriptorSetLayout benötigt. Das VkDescriptorSet symbolisiert Speicher, der die aktuellen Deskriptoren beinhaltet. Es kann so konfiguriert werden, dass ein Deskriptor auf einen bestimmten Puffer, eine bestimmte Pufferansicht, ein Bild oder einen bestimmten Sampler zeigt. Dies wird durch die Funktion `vkUpdateDescriptorSet` realisiert.

Mehrere VkDescriptoSets können in einen VkCommandBuffer gebunden werden, um dann durch Rendering-Befehle Anwendung zu finden. Dazu wird die Funktion `vkCmdBindDescriptorSets` ausgeführt. Für die Exekution wird jedoch noch ein weiteres Objekt benötigt, das `VkPipelineLayout`. Das Objekt dient dazu Vulkan mitzuteilen, wie viele verschiedene VkDescriptorSets verwendet werden. Es stellt eine Konfiguration für die Render Pipeline dar, welche Arten von VkDescriptorSet an den VkCommandBuffer überreicht werden. Dies geschieht über einen Array bestehend aus VkDescriptorSetLayout Objekten.[25][2419]

VkRenderPass

In den gewöhnlichen Grafik APIs wird das gerendert, was als nächstes auf der Liste steht. Wohingegen in Vulkan das Rendern eines Frames geplant wird und in Passes und Subpasses organisiert wird. Ein Subpass stellt eine Phase des Renderings dar, die eine Teilmenge der Anlagen in einem Renderpass liest und schreibt. Rendering-Befehle werden in einem bestimmten Subpass einer Renderpassinstanz aufgezeichnet.

Ein `VkRenderPass` stellt eine Sammlung von Attachments (Anlagen), Subpasses und Abhängigkeiten zwischen Subpasses dar. Außerdem beschreibt es wie Attachments im Subpass Vorgang verwendet werden.

Attachments sind in Vulkan Renderziele, die ein Bild als Ausgabe des Renderings verwenden. Durch Attachments werden die Formate der Bilder beschrieben, beispielsweise kann ein VkRenderPass ein Farb Attachment in dem Format `R8G8B8A8_UNORM` und ein Depth-Stencil Attachment in dem Format `D16_UNROM` verwenden. Darüber hinaus wird zu Beginn der VkRenderPass angegeben, ob der Inhalt behalten oder gelöscht werden soll.[25][2419]

VkFramebuffer

Der `VkFramebuffer` ist eine Anbindung an VkImages, die als Attachments verwendet werden. Das VkFramebuffer Objekt wird durch die Angabe von VkRenderPass und VkImageViews erstellt. Dabei ist die Übereinstimmung mit den Spezifikationen der VkRenderPass nicht zu vernachlässigen. Framebuffer stellen eine weitere Ebene über den VkImages dar und gruppieren VkImageViews, sodass diese als Attachments beim Rendern in VkRenderPasses gebunden werden. Sobald das Rendering eines VkRenderPasses beginnt wird die Funktion `vkCmdBeginRenderPass` aufgerufen und der Framebuffer übergeben.[25][2419]

VkPipeline

Die `VkPipeline` stellt die Sammlung der Mehrheit vorher definierter Objekte dar. Ein Attribut der VkPipeline ist die `VkPipelineLayout`, was das Layout der Deskriptoren definiert. Pipelines sind in zwei Arten unterteilt: die Grafik Pipeline und die Compute Pipeline. Die Compute Pipeline beschäftigt sich mit reinen Rechenvorgängen, wohingegen die Grafik Pipeline aus mehreren Shader Stufen, mehreren fest funktionalen Pipelinestufen und einem Pipelinelayout besteht und dadurch komplexer ist. Für jeden einzelnen Parametersatz, der beim Rendern benötigt wird, muss eine neue Pipeline erstellt werden. Diese wird dann als aktuelle aktive Pipeline in einem VkCommandBuffer festlegt, indem die Funktion `vkCmdBindPipeline` aufgerufen wird.[25][2419]

VkShaderModule

Die Shader-Kompilation ist ein mehrstufiger Prozess in Vulkan. Erstens unterstützt Vulkan keine hochrangige OpenGL Shading Language (GLSL) oder High Level Shading Language (HLSL). Stattdessen akzeptiert Vulkan ein Format namens Standard Portable Intermediate Representation - V (SPIR-V), das jede übergeordnete Sprache ausgeben kann. Ein Puffer mit SPIR-V Daten wird verwendet, um ein `VkShaderModul` zu erstellen. Bei der Erstellung der Pipeline für jede Shaderstufe, die verwendet wird (Vertex, Tessellationskontrolle, Tessellationsauswertung, Geometrie, Fragment oder Berechnung), wird das ShaderModule und der Namen

der Einstiegspunktfunktion (wie „main“) angegeben. Die SPIR-V ist eine Sprache zur Definition von Shadern. Es ist beabsichtigt, ein Compiler-Ziel aus einer Reihe von verschiedenen Sprachen zu sein. Dadurch hat der Programmierer die Freiheit Shader flexibel zu schreiben.[25][2419]

VkPipelineCache

Mit dem Hilfsobjekt `VkPipelineCache` kann die Erstellung der Pipeline beschleunigt werden. Es ist ein Objekt, das optional bei der Erstellung der Pipeline übergeben werden kann. Die Cache trägt dazu bei, die Performance durch geringeren Speicherverbrauch und die Kompilierungszeit der Pipelines zu verbessern. Der Zustand eines PipelineCache-Objekts kann gespeichert und in einen Puffer mit binären Daten geladen werden, um es auf der Festplatte abzulegen. Bei der nächsten Ausführung kann diese dann verwendet werden.[25][2419]

Query

Query ist ein weiterer Objekttyp in Vulkan. Es kann verwendet werden, um bestimmte numerische Werte, die von der GPU geschrieben wurden, zurückzulesen. Es gibt verschiedene Arten von Abfragen wie Occlusion, ob einige Pixel gerendert wurden oder Timestamp (ein Zeitstempelwert von einem GPU-Hardwarezähler). Query hat keinen eigenen Typ, da es sich innerhalb eines QueryPools befindet und nur durch einen `uint32_t` Index repräsentiert wird. QueryPool kann durch Angabe von Typ und Anzahl der zu enthaltenden Abfragen erstellt werden. Sie können dann verwendet werden, um Befehle an den Befehlspuffer auszugeben, wie `vkCmdBeginQuery`, `vkCmdEndQuery` oder `vkCmdWriteTimestamp`.[25][2419]

Synchronisation

Schließlich gibt es noch Objekte, die für die Synchronisation verwendet werden: Fence, Semaphore und Event. Eine Fence signalisiert dem Host, dass die Ausführung einer Aufgabe abgeschlossen ist. Fence hat keine eigene Befehlsfunktion, wird jedoch beim Aufruf von `vkQueueSubmit` übergeben. Sobald die eingereichte Queue abgeschlossen ist, wird die entsprechende Fence signalisiert.

Mithilfe einer Semaphore kann eine Abhängigkeit zwischen Warteschlangenoperationen oder zwischen einer Warteschlangenoperation und dem Host eingefügt werden. Eine Semaphore wird ohne Konfigurationsparameter erstellt. Es kann verwendet werden, um den Ressourcenzugriff über mehrere Queues hinweg zu steuern. Nach Abschluss der Ausführung einer Queue Operation kann eine Semaphore signalisiert werden und es kann eine Queue Operation warten lassen, bis eine Semaphore signalisiert wird.

Ein Event wird auch ohne Parameter erzeugt. Es lassen sich Abhängigkeiten zwischen Befehlen, die an dieselbe Warteschlange gesendet werden oder zwischen dem Host und einer

Warteschlange eingefügt werden definieren. Sie haben zwei Zustände signalisiert und nicht signalisiert. Eine Anwendung kann ein Ereignis entweder auf dem Host oder auf dem Gerät signalisieren oder aufheben. Eine Gerät kann warten, bis ein Ereignis signalisiert wird, bevor sie weitere Operationen ausführt. Mithilfe der Methoden `vkCmdSetEvent`, `vkCmdResetEvent` und `vkCmdWaitEvent` werden diese Funktionlitäten ausgeführt. [25][2419]

4 Vulkan basierter Straßengraph

4.1 Entwicklungsumgebung und verwendete Bibliotheken

Diese Sektion soll einen kurzen Überblick über die Entwicklungsumgebung mit Vulkan schaffen. Hierbei werden die verwendeten Komponenten und ihr Nutzen für die Arbeit offen gelegt. Der Vulkan basierter Straßengraph (VbS) in dieser Arbeit wurde auf einer Windows Plattform mit der Visual Studio Entwicklungsumgebung programmiert, kann jedoch in einer beliebigen Umgebung entwickelt werden.

4.1.1 Vulkan SDK

Die bedeutsamste Komponente für die Entwicklung ist das Vulkan Source Development Kit (SDK). Es beinhaltet die Header Dateien, Standard Validierungslayer, Debugging Tools und einen Loader für die Vulkan Funktionen. Der Loader verarbeitet die Funktionen zur Laufzeit. Die SDK kann von der https://vulkan.lunarg.com Seite heruntergeladen werden. Bevor Vulkan SDK heruntergeladen und installiert wird, ist die Kompatibilität mit der Grafikkarte zu überprüfen und eventuelle Treiber Updates zu laden.

Unter Windows wird die Vulkan-Installer.exe Datei ausgeführt, um die Installation zu starten. Nach einer erfolgreichen Installation kann unter dem `Bin` Ordner im Installationsverzeichnis die `vkcube.exe` Demo ausgeführt werden.

Auf einem Linux Betriebssystem wird die Vulkan SDK zunächst entpackt beispielsweise durch das Kommando `tar -xzf vulkansdk-linux-x86_64-xxx.tar.gz`, wenn eine .tar.gz Datei heruntergeladen wurde. Die Beispiele in der SDK hängen von der XCB-Bibliothek ab. Dies ist eine C-Bibliothek, die für die Schnittstelle mit dem X Window System verwendet wird. Es kann in Ubuntu aus dem Paket `libxcb1-dev` installiert werden. Um die Demo unter Linux zu starten müssen vorher die Beispiele mit dem Kommando `./build_examples.sh` gebaut werden. Nach der erfolgreichen Kompilierung kann die `vkcube` Executable im Verzeichnis `examples/build` ausgeführt werden.

Abbildung 4.1: vkcube.exe

4.1.2 GLFW

Die Graphics Library Framework (GLFW) wird verwendet, da Vulkan eine plattformunabhängige API ist und keine Werkzeuge zur Erstellung einer Fensteranzeige enthält. Aufgrund der Tatsache, dass sowohl die Vulkan API als auch GLFW plattformunabhängig sind, werden beide oft zusammen verwendet. [20]

Für die Interaktion mit dem Fenster Manager wird eine standardisierte Window System Interface (WSI) Erweiterung gebraucht. Die Oberfläche ist das Medium, worauf das Bild am Ende angezeigt wird. Die Instantiierung erfolgt über eine Referenz auf das native Fensterhandle, zum Beispiel das Handle Window (HWND) unter der Windows Plattform. GLFW hat eine integrierte Funktion, welche solche plattformspezifischen Details behandelt.

Darüber hinaus unterstützt GLFW Tastatur, Maus, Gamepad, Zeit und Ereignis Inputs. Dies wird verwendet um sich auf der Welt beziehungsweise Globusoberfläche frei zu bewegen. Das Framework kann aus der offiziellen Webseite (https://www.glfw.org) geladen werden.

4.1.3 GLM

GLM steht für OpenGL Mathematics und ist eine C++ Bibliothek für Grafiksoftware. Diese Bibliothek funktioniert mit OpenGL, stellt aber auch die Interoperabilität mit anderen Bibliotheken von Drittanbietern und SDKs sicher. Es ist ein guter Kandidat für Software-Rendering (Raytracing / Rasterisierung), Bildverarbeitung, physikalische Simulationen und jeden Kontext, der eine simple und praktische Mathematikbibliothek erfordert.[21]

Bei dem VbS wird die GLM Bibliothek für Vektoren verwendet. Außerdem werden die Matrizen für Model, View und Projection berechnet. Die Model Matrix bildet Koordinaten in einem Model auf Koordinaten eines bestimmten Raumes ab. Abhängig sind diese Raum Koordinaten von den Koordinaten, dem Maßstab, und der Rotation des zu zeichnenden Models. In dem entwickelten Straßenviewer sind bereits alle Koordinaten in Weltkoordinaten angegeben, wodurch die Model Matrix auf die Indentitätsmatrix gesetzt wird. In der 3D-Programmierung werden Bewegungs- und Blickwinkeländerungen von der View Matrix registriert. Es werden die passende Position und die Blickrichtung nach jeder Interaktionen berechnet. Nachdem die Welt mit Hilfe der View-Transformation ausgeführt wurde, kann die Projektion-Transformation angewendet werden. Es wird erwartet, dass die Koordinaten innerhalb eines bestimmten Bereichs liegen und jede Koordinate, die außerhalb dieses Bereichs liegt, abgeschnitten wird. Die abgeschnittenen Koordianten werden verworfen, hieraus leitet sich der Name Clip-Raum ab. Die Projektions Matrix transformiert die Knotenkoordinaten von der View in einen Clip Raum. [22]

4.1.4 Boost Spirit X3

Boost Spirit X3 ist ein Parser für C++. Es ermöglicht Grammatiken ähnlich der Erweiterten-Backus-Nauer-Form (EBNF) zu definieren. Diese Grammatikspezifikationen können mit anderem C++ Code verbunden werden, dadruch lässt dich der Code effizient gestalten. Außerdem muss Boost Spirit X3 keinen zusätzlichen Übersetzungsschritt vom EBNF-Quellcode in den C++-Code durchführen. Da die Ziel-Input-Grammatiken vollständig in C++ geschrieben sind, werden keine separaten Tools zum Kompilieren, Vorverarbeiten oder Integrieren in den Build-Prozess benötigt.

In dem VbS werden durch diese Bibliothek vorbereitete Straßendaten eingelesen und beim Einleseprozess direkt in kartesische Koordinaten umgewandelt.

4.2 Graphdaten

4.2.1 Relevante Daten aus OSM

Der VbS stellt die Straßen eines übergebenen Kartenausschnitts dar. Die relevanten Daten für einen Straßenviewer sind die Ways, welche mit dem Tag-Key `highway` versehen wurden. Es werden folgende Values in Betracht gezogen:

- motorway
- trunk
- primary
- secondary
- teritary
- unclassified
- residential
- motorway_link
- trunk_link
- primary_link
- secondary_link
- teritary_link
- living_street
- service
- road
- turning_circle

Trägt ein Way einer dieser 16 highway Tags, so wird es in die Liste der zu zeichnenden Straßen aufgenommen. Die Graphdaten werden mittels einer im FMI entwickelten Software zu einer kompakten Straßendaten-Datei verarbeitet. Hierzu wird ein Kartenausschnitt mit der Dateiendung `.osm.pbf` übergeben. Die Ausgabe ist eine Datei im `.fmi` Format.

In der Datei werden zunächst die ganzen Straßenknoten (Vertex oder Node) aufgezählt. Pro Zeile steht ein Vertex mit den Attributen NodeID1, NodeID2, Längen- und Breitengrad und Elevation (de. Höhe) hinterlegt. Danach folgen die Edge Daten, die zeigen welche Vertexe miteinander verbunden sind, um schließlich eine Straße zu bilden. Jede Zeile stellt hier eine Verbindung zwischen zwei Vertex Punkten dar, indem die Indexe der vorher aufgelisteten Vertexe als Start und Ziel gesetzt werden. Die Edges haben demnach folgende Attribute: Startindex, Zielindex, Kosten, Typ und Maximalgeschwindigkeit.

4.2.2 Import und Einarbeitung in den Vertex Buffer

Im Vulkan basierten Straßengraphen gibt es zwei Varianten, wie Daten eingelesen und verarbeitet werden. Primär wurde diese Arbeit für die Deutschlandkarte konzipiert. Aus diesem Grund startet der VbS ohne Parameterübergabe direkt mit Einarbeitung der zugeschnittenen Deutschlanddaten. Diese Variante funktioniert mit der vorher vorgestellten Boost Spirit X3 Bibliothek. Später wurde die Unterstützung anderer Kartenausschnitte mithilfe der Standard C++ fstream Bibliothek implementiert. Bei dieser zweiten Variante wird der Pfad eines Kartenausschnitts in dem `.fmi` Format als Startparamater übergeben und Zeile für Zeile ausgewertet.

Boost Spirit X3 Variante

Die erste Variante hat speziell angefertigte Daten. Hier sind die zugeschnittenen Deutschlanddaten in 20 Dateien unterteilt, wobei jeweils zwei Dateien einen Straßentypen definieren. Pro Straßentyp sind es demnach zwei Dateien: Die Vertex-Datei und die Edge-Datei. In den Vertex-Dateien sind die Knotenpunkte der Straßen enthalten, dagegen schließen die Edge-Dateien die Indizes der zu verbindenden Vertexe ein. Diese 20 Dateien wurden mit der FMI Software erstellt und haben eine Gesamtgröße von 3,04 Gigabyte. Die Bibliothek Boost Spirit X3 wird verwendet. Für die Vertex und Index Dateien gibt es jeweils eine separate Parser Grammatik:

```
bool r = phrase_parse(first, last,
        // Begin grammar
        // id1, id2, lat, lon, elev
        (
            (float_ > float_ > float_[get_lat] > float_[get_lon] > float_) % eol
        ),
        // End grammar
        blank);
```

Der erste Code Abschnitt parst die Vertex Daten, indem nach einer Abfolge von fünf `float` Zahlen und einer end-of-line `eol` gefiltert wird. Wurde eine Zeile identifiziert, so können die Vertex Daten generiert werden. Hierbei werden die Funktionen in den eckigen Klammern `[ ]` ausgeführt, die vorher definiert wurden. Die eckigen Klammern symbolisieren eine Funktion, die auf den Wert vor den Klammern angewandt werden soll. Es werden pro Zeile die `get_lat` Funktion und die `get_lon` Funktion aufgerufen, um den Längen- und Breitengrad zu entnehmen und anschließend in kartesische Koordinaten umzuwandeln. Die umgewandelten Werte werden danach in die passenden VertexBuffer gepusht. Im zweiten Code Abschnitt werden analog die Edge Daten eingelesen und in die IndexBuffer geschrieben. Diese Methode ist dank der Boost Library effizient und schnell.

```
bool r = phrase_parse(first, last,
        // Begin grammar
        // srcId, trgId, cost, type, ms
        (
                (int_[get_src] > int_[get_trg] > int_ > int_ > int_) % eol
        ),
        // End grammar
        blank);
```

Die Erstellung der Vertex- und IndexBuffer in der Variante, werden durch die eingelesen Dateien definiert. Da diese Dateien in Straßenkategorien unterteilt sind, geben sie vor in welche Buffer die Daten gespeichert werden müssen.

fstream Variante

In der zweiten Variante handelt es sich nur um eine Datei, die eingelesen wird. Sowohl Vertex als auch die Index Daten sind in dieser Datei enthalten. Es wird zeilenweise gelesen und wie in der ersten Variante beim Lesen die Vertex und Index Daten in die jeweiligen Buffer

geschrieben. Hierfür wird die Standard C++ fstream Bibliothek verwendet. Diese Variante ist deutlich langsamer als die erste Variante mit der Boost Bibliothek. Es werden zunächst die ganzen Nodes gelesen und konvertiert in einen VertexMaster geschrieben.

```
...
for (int i = 0; i < nodeQuantity; i++)
{
        std::getline(file, line);
        std::stringstream lineStream(line);
        double lat, lon;
        // Read an integer at a time from the line
        /*1te Zahl ist NodeID*/ lineStream >> placeholder;
        /*2te Zahl ist NodeID2*/ lineStream >> placeholder;
        /*3te Zahl ist LATITUDE*/ lineStream >> lat;
        /*4te Zahl ist LONGITUDE*/ lineStream >> lon;
        /*5te Zahl ist elevation*/ lineStream >> placeholder;
        lat = convertDegreeToRadian(lat);
        lon = convertDegreeToRadian(lon);
        x = radius * cos(lat) * sin(lon);
        y = radius * sin(lat);
        z = -radius * cos(lat) * cos(lon);
        vertexMaster.push_back({ { x, y, z } });
}
...
```

Nachdem die Nodes fertig abgearbeitet wurden, kommen die Edge Zeilen und werden nach ihren Typ gefiltert. Jede Straßenkategorie hat ihren eignen Typen. Dieser ist bedeutsam für die Eingliederung in die Vertex- und Index Buffer. Je nach Typ werden die Daten aus dem VertexMaster in die einzelnen Vertexe für die Straßenkategorien eingespeist.

```
while (std::getline(file, line)) {
        std::stringstream lineStream(line);
        lineStream >> srcIDX;
        lineStream >> trgIDX;
        lineStream >> placeholder;
        lineStream >> type;
        if (type == 1 || type ==2)
        {
                Vertex pbSRC = { { vertexMaster[srcIDX].pos[0], vertexMaster[srcIDX].pos[1],
                    vertexMaster[srcIDX].pos[2] }, {0, 0, 1} };
                Vertex pbTRG = { { vertexMaster[trgIDX].pos[0], vertexMaster[trgIDX].pos[1],
                    vertexMaster[trgIDX].pos[2] }, {0, 0, 1} };
                verticesStreetTyp1.push_back(pbSRC);
                verticesStreetTyp1.push_back(pbTRG);

                indicesStreetTyp1.push_back(count1);
                indicesStreetTyp1.push_back(count1 + 1);
                count1 += 2;
        }
        ...
}
```

4.3 Die Implementierung des Vulkan basierten Straßengraphen

In dem Kapitel Vulkan Objekte wurde der Workflow von Vulkan erklärt. Der Worklflow wird in dem VbS nun anhand von Codeausschnitten ausgeführt. Das Kapitel ist folgendermaßen aufgebaut: Als erstes wird ein Fenster mit GLFW initialisiert. Daraufhin werden die fundamentalen Vulkan Objekte in ihrem Entstehprozess gezeigt. Sind GLFW und Vulkan vollständig aufgesetzt, wird anschließend die Haupschleife des Programms dargestellt. Zu guter Letzt werden die gesamten Objekt aufgeräumt.

4.3.1 Initialisierung von GLFW

Eine grafische Oberfläche kann mithilfe von nativen Plattform APIs oder Bibliotheken wie GLFW und Simple DirectMedia Layer (SDL) erstellt werden. In dieser Arbeit wird GLFW verwendet. Bevor die GLFW-Bibliothek verwendet werden kann, muss sie initialisiert werden. Dies geschieht über die `glfwInit()` Methode.[23]. Danach werden GLFW Window Hinweise gesetzt, welche für den `glfwCreateWindow` Aufruf von Bedeutung sind. „GLFW_CLIENT_API “ und „GLFW_NO_API “ signalisieren, dass es keine Client API gibt, die GLFW bekannt ist.

Nachdem ein Fenster kreiert wurde, stellt die `glfwSetFramebufferSizeCallback` sicher, dass bei einer Größenänderung des Fensters eine Benachrichtigung versandt wird. Dies ist später für das neu erstellen der Swap Chain.

```
void initWindow() {
        glfwInit();
        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
        glfwSetWindowUserPointer(window, this);
        glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
    ...
}
```

4.3.2 Erstellung der Vulkan Objekte

Im Folgenden werden die fundamentalen Schritte zur Erstellung des VbS aufgezeigt. Hierbei werden die im Kapitel Grundlagen vorgestellten Vulkan Objekte mit Hinsicht auf ihre Erstellung anhand von kurzen Code Abschnitten gezeigt.

Schritt 1: Instance und PhysicalDevice Auswahl

Jede Vulkan Anwendung findet ihren Anfang mit der Einrichtung der Vulkan API über eine VkInstance. Eine Instance wird erstellt, indem die Anwendung und alle Erweiterungen beschrieben werden. Nahezu bei jeder Objekterstellung in Vulkan muss eine CreateInfo beschrieben werden. Die CreateInfo beinhaltet alle Information die für das Anlegen des Objekts notwendig sind. Der `sType` beispielsweise gibt an, welcher Struktur Typ mit der CreateInfo definiert wird. Die erwähnten ValidationLayer und Erweiterungen werden auch bei der CreateInfo gesetzt.

Listing 4.1: CreateInfo der VkInstance

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;

auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
createInfo.ppEnabledExtensionNames = extensions.data();

if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
}else {
        createInfo.enabledLayerCount = 0;
}
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create instance!");
}
```

Nachdem die Instanz erstellt wurde, kann nach Vulkan unterstützter Hardware gesucht werden. Ist die passende Hardware gefunden, werden ein oder mehrere VkPhysicalDevices für den Betrieb ausgewählt. Die Auswahl des präferierten Gerätes kann von Eigenschaften wie Video Random Access Memory (VRAM) Größe oder Fähigkeiten des Geräts abhängen.

Schritt 2: Logical Device und Queue-Familien

Daraufhin folgt die Erstellung eines VkDevices beziehungsweise eines logischen Geräts. Grundbausteine hierfür sind die vorher ersuchten Eigenschaften und Fähigkeiten des Geräts. Mit ihnen werden die VkPhysicalDeviceFeatures genauer beschrieben, zum Beispiel das Multi-Viewport Rendering, die Verwendung von 64-Bit-Floats oder die Anisotropische Filterung.

Listing 4.2: CreateInfo der VkDevice

```
VkDeviceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.pEnabledFeatures = &deviceFeatures;
```

```
createInfo.enabledExtensionCount = static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();

if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
}else {
        createInfo.enabledLayerCount = 0;
}

if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
        throw std::runtime_error("failed to create logical device!");
}
```

Operationen, wie Zeichenbefehle und Speicheroperationen werden in Vulkan asynchron ausgeführt, indem sie eine Queue übertragen werden, die VkQueue. Dafür ist die Wahl einer Queue Family bedeutsam. Queues werden aus Queue Families zugewiesen, wobei jede Queue Family einen bestimmten Satz von Operationen in ihren Queues unterstützt. Für Grafik-, Rechen- und Speichertransferoperationen beispielsweise könnte es separate Queue Families geben. Die Verfügbarkeit von Queue Families könnte auch als Unterscheidungsmerkmal bei der Auswahl physikalischer Geräte verwendet werden.

Listing 4.3: Mögliche Queue-Familien finden

```
QueueFamilyIndices PhysicalDeviceSelection::findQueueFamilies(VkPhysicalDevice device, VkSurfaceKHR
    surface) {
        QueueFamilyIndices indices;
        uint32_t queueFamilyCount = 0;
        vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);
        std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
        vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());
        int i = 0;
        for (const auto& queueFamily : queueFamilies) {
                if (queueFamily.queueCount > 0 && queueFamily.queueFlags &
                    VK_QUEUE_GRAPHICS_BIT) {
                        indices.graphicsFamily = i;
                }
                VkBool32 presentSupport = false;
                vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);
                if (queueFamily.queueCount > 0 && presentSupport) {
                        indices.presentFamily = i;
                }
                if (indices.isComplete()) {
                        break;
                }
                i++;
        }
        return indices;
}
```

Schritt 3: Surface und Swapchain

Nachdem ein Fenster mit GLFW erstellt wurde, kann die VkSurfaceKHR surface angelegt werden. Dies geschieht mit der `glfwCreateWindowSurface` Funktion von GLFW. Der KHR-Postfix symbolisiert Objekte einer Vulkan Erweiterung.

Listing 4.4: Surface Erstellung

```
void createSurface() {
        if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
                throw std::runtime_error("failed to create window surface!");
        }
}
```

Für die Erstellung der VkSwapchainKHR sind neben der gewohnten sType die Oberfläche (Surface), auf der die Swapchain Bilder präsentiert werden. Darüber hinaus wird durch das minImageCount Element die Mindestanzahl an darstellbaren Bildern der Anwendung beschrieben. Nachfolgend wird für die Erstellung der Swapchain Bilder das imageFormat Element verwendet. Der imageColorSpace-Wert gibt an, wie die Swapchain die Bilddaten interpretiert und das imageExtent Element dient zur Definition der Pixelgröße der Swapchain Bilder. Nachstehend definiert die imageArrayLayer Komponente die Anzahl der Ansichten auf die Oberfläche. In diesem Fall stellt der Wert „1 “ eine nicht 3D Ansicht dar. Danach gibt das imageUsage Element mit dem Wert VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT an, dass die Image verwendet werden kann, um für den FrameBuffer eine ImageView zu erstellen. Das sharingMode Glied mit dem Wert VK_SHARING_MODE_CONCURRENT besagt, dass mehrere Queue-Families auf diese Image Zugriff haben. Andernfalls mit dem Wert VK_SHARING_MODE_EXCLUSIVE hat nur eine Queue-Family Zugang zu der Image. Der queueFamilyIndexCount Wert beschreibt, wie viele Queue-Families zugriff auf ein Image haben. Weiter definiert das preTransform Element die Ausrichtung des Bildes vor der Präsentation. Zusätzlich gibt das compositeAlpha Element an, ob das Alpha Blending beachtet wird bei mehreren Surfaces. Dann folgt das presentMode Glied, das den Präsentationsmodus der Swapchain beinhaltet. Hiermit wird bestimmt, wie eine Präsenationsanfrage verarbeitet wird. Schließlich gibt die clipped Komponente an, ob die Vulkan-Implementierung Rendering-Operationen verwerfen darf, die Bereiche der Oberfläche betreffen, die nicht sichtbar sind.

Listing 4.5: CreateInfo der VkSwapchainKHR

```
VkSwapchainCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = surface;
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1;
createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
QueueFamilyIndices indices = pds->findQueueFamilies(physicalDevice, surface);
uint32_t queueFamilyIndices[] = { indices.graphicsFamily.value(), indices.presentFamily.value() };
if (indices.graphicsFamily != indices.presentFamily) {
```

```
            createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
            createInfo.queueFamilyIndexCount = 2;
            createInfo.pQueueFamilyIndices = queueFamilyIndices;
}
else {
            createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
}
createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;

if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
            throw std::runtime_error("failed to create swap chain!");
}
```

Schritt 4: Image Views und Framebuffers

Das Zeichen eines Images aus der Swapchain erfordert das Einbinden in einen VkImageView und in einen VkFramebuffer. Die VkImageView ist, wie der Name preisgibt, die Ansicht in ein Bild. Es beschreibt, wie auf das Bildes zugegriffen werden soll, zum Beispiel als 2D Texture. Des Weiteren werden die SubResources angeben, die einen Teilbereich der Image beschreiben. Im VbS ist das der Tiefen- und der Farbaspekt. Zusätzlich wählen SubResources den Satz von Mipmap-Ebenen und Array-Ebenen, auf die die ImageView zugreifen kann.

Listing 4.6: CreateInfo der ImageView

```
VkImageViewCreateInfo viewInfo = {};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = image;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = format;
viewInfo.subresourceRange.aspectMask = aspectFlags;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;
VkImageView imageView;
if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
            throw std::runtime_error("failed to create texture image view!");
}
```

Nachdem umschließen des Images in einen VkImageView wird diese durch den VkFramebuffer referenziert. Framebuffer stellen eine Sammlung von Speicheranhängen dar, die von einer Render Pass Instanz gebraucht werden. Beispiele hierfür sind Farb- und Tiefenanhänge.

Listing 4.7: CreateInfo des FrameBuffers

```
VkFramebufferCreateInfo framebufferInfo = {};
framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
framebufferInfo.renderPass = renderPass;
```

```
framebufferInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
framebufferInfo.pAttachments = attachments.data();
framebufferInfo.width = swapChainExtent.width;
framebufferInfo.height = swapChainExtent.height;
framebufferInfo.layers = 1;
if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, &swapChainFramebuffers[i]) != VK_SUCCESS) {
        throw std::runtime_error("failed to create framebuffer!");
}
```

Schritt 5: RenderPass

Die RenderPasses beschreiben die Art der Bilder, wie sie verwendet werden und wie ihr Inhalt behandelt werden soll. Um ein RenderPass Objekt zu erstellen werden die Attachments und Subpasses angegeben. Jede RenderPass enthält mindestens eine Subpass. Die Subpass stellt eine Phase des Renderings dar. In dem VbS sind es die Tiefen und Farbattachments, die in dem Subpass verwendet werden. Die dependency gibt an, welche Subpasses zusammenhängen und in welcher Reihenfolge sie zusammengestetzt werden.

Listing 4.8: CreateInfo des RenderPasses

```
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
renderPassInfo.pAttachments = attachments.data();
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 1;
renderPassInfo.pDependencies = &dependency;
if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {
        throw std::runtime_error("failed to create render pass!");
}
```

Schritt 6: Graphics Pipeline

Durch das Erstellen eines VkPipeline Objekts wird die Grafikpipeline in Vulkan eingerichtet. Hiermit werden die konfigurierbaren Zustände der Grafikkarte beschrieben. Die Größe des Ansichtsfensters durch das ViewPort Element oder der Tiefenpufferbetrieb durch den DepthStencilState sind Beispiele hierfür, aber auch programmierbare Zustände über das VkShaderModule. Die VkShaderModule-Objekte werden aus Shader-Byte-Code erstellt, dadurch wird der Treiber informiert, welche Renderziele in der Pipeline verwendet werden. Die VertexInputState, die InputAssembly und der RasterizerState geben an wie die Vertex Daten aufgebaut sind und wie sie innerhalb der Pipeline verarbeitet werden sollen.

```
VkGraphicsPipelineCreateInfo pipelineInfo = {};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = 2;
pipelineInfo.pStages = shaderStages;
pipelineInfo.pVertexInputState = &vertexInputInfo;
```

```
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pDepthStencilState = &depthStencil;
pipelineInfo.layout = pipelineLayout;
pipelineInfo.renderPass = renderPass;
pipelineInfo.subpass = 0;
pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;
```

Schritt 7: Command Pools und Command Puffer

Wie bereits erwähnt müssen Operationen in eine Warteschlange gestellt werden. Die Warteschlangen werden von einem VkCommandBuffer aufgezeichnet, bevor sie verarbeitet werden können. Diese Command Buffer werden von einer CommandPool zugewiesen, welche in Assoziation mit der Queue Family steht. Das Level Element mit VK_COMMAND_BUFFER_LEVEL_PRIMARY bedeutet, dass es der erste zu allokierende CommandBuffer ist. Weil das Bild im Framebuffer von dem Bild in der Swapchain abhängt, wird der Command Buffer für jedes mögliche Bild aufgezeichnet und muss zur richtigen Zeit gezeichnet werden.

```
VkCommandBufferAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocInfo.commandPool = commandPool;
allocInfo.commandBufferCount = 1;
VkCommandBuffer commandBuffer;
vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
```

Schritt 8: Hauptschleife

Anschließend an den CommandBuffer wird die Hauptschleife definiert. Als erstes wird mit der vkAcquireNextImageKHR Funktion ein Bild aus der Swapchain entnommen. Der zu dem Bild passende Command Buffer wird ausgewählt und mit vkQueueSubmit ausgeführt. Daraufhin wird das Bild wieder zurück an die Swapchain gegeben, um es zuletzt mit dem Befehl vkQueuePresentKHR auf dem Bildschirm zu präsentieren.

Vulkan hat einen asynchronen Ausführungsvorgang in seinen Queues. Es ist daher essenziell, dass Synchronisationsobjekte verwendet werden. Semaphoren sind solche Objekte, die korrekte Reihenfolge in der Ausführung gewährleisten. Um zu verhindern, dass ein Rendervorgang für ein Bild gestartet wird, obwohl es noch für die Präsentation auf dem Bildschirm gelesen wird, ist die Ausführungsfolge des Draw CommandBuffers von großer Wichtigkeit. Eine zweite Semaphore wird verwendet, um den Aufruf der vkQueuePresentKHR zu kontrollieren. Das Rendering muss abgeschlossen sein, bevor diese Funktion aufgerufen wird.

```
VkResult result = vkAcquireNextImageKHR(device, swapChain, std::numeric_limits<uint64_t>::max(),
    imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);
...
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
VkSemaphore waitSemaphores[] = { imageAvailableSemaphores[currentFrame] };
VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffers[imageIndex];
VkSemaphore signalSemaphores[] = { renderFinishedSemaphores[currentFrame] };
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
vkResetFences(device, 1, &inFlightFences[currentFrame]);
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
        throw std::runtime_error("failed to submit draw command buffer!");
}
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = signalSemaphores;
VkSwapchainKHR swapChains[] = { swapChain };
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = swapChains;
presentInfo.pImageIndices = &imageIndex;
result = vkQueuePresentKHR(presentQueue, &presentInfo);
```

Schritt 9: Das Aufräumen aller Objekte

Bevor eine Anwendung beendet wird, sollten alle erstellen Objekte zerstört und der Speicher freigegeben werden, um alle während oder nach der Initialisierung zugewiesenen Ressourcen freizugeben.[23]

```
cleanupSwapChain();
vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);
vkDestroyBuffer(device, indexBuffer, nullptr);
vkFreeMemory(device, indexBufferMemory, nullptr);
...
vkDestroyBuffer(device, vertexBuffer, nullptr);
vkFreeMemory(device, vertexBufferMemory, nullptr);
...
for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
}
vkDestroyCommandPool(device, commandPool, nullptr);
vkDestroyDevice(device, nullptr);
```

```
if (enableValidationLayers) {
        DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
}
vkDestroySurfaceKHR(instance, surface, nullptr);
vkDestroyInstance(instance, nullptr);
glfwDestroyWindow(window);
glfwTerminate();
```

Zusammenfassung

Die hier dargestellte Abfolge beinhaltet nicht alle Schritte, wie das Zuweisen von Vertex Buffern oder das Erstellen homogener Buffer. Die Schritte zusammengefasst:

- Kreieren eines VkInstances
- Auswählen einer passenden Grafikkarte (VkPhysicalDevice)
- Erstellen von VkDevices und VkQueues für das Zeichnen und das Präsentieren
- Anlegen von Window, Window Surface und Swapchain
- Einbinden der Swapchain Images in den VkImageView
- Spezifizieren eines Render Passes, welches die Render Ziele und den Gebrauch bestimmt
- Aufsetzen der Frame Buffer für den Render Pass
- Erstellen der Graphics Pipeline
- Allokieren und Befüllen des CommandBuffers mit den Draw Commands für jede mögliche Swapchain image
- Zeichnen der Frames durch beschaffen der Images, übermitteln des richtigen Draw Command Buffers und die Images wieder zurück an die Swapchain führen
- Aufräumen der gesamten Anwendung

4.4 Ausführung des Programms unter Linux

Um das Programm auf einer Linux Umgebung auszuführen zu können, müssen zunächst die fehlenden Bibliotheken und das Vulkan SDK installiert werden. Dieses Vorgehen wurde in dem Abschnitt „Entwicklungsumgebung und verwendete Bibliotheken“ erläutert. Nachstehende Schritte müssen danach ausgeführt werden:

1. Entpacken des VbS mit dem Befehl: `unzip vulkan-based-streetviewer.zip`.
2. Navigation in das entpackte Verzeichnis, in der sich das Makefile befindet, und Ausführung des Makefiles für die Compilierung mit dem Befehl: `make`.

3. Nach erfolgreicher Compilierung befindet sich eine Executable im Verzeichnis, diese mit dem Befehl: `./vulkan-based-streetviewer XXX` ausführen. Das XXX ist hierbei ein Platzhalter für den Pfad Straßengraphen, welcher zu visualisieren ist.

4.5 Bedienung

Die Bedienung des Programms ist ähnlich wie bei den meisten Computerspielen. Mit den W, A, S, D oder den Pfeiltasten lässt sich die Kamera nach links, rechts, vorne und hinten bewegen. Um die Ausrichtung der Kamera zu verändern wird eine Maus benötigt. Hiermit lässt sich dann durch Gedrückthalten der linken Maustaste und der gleichzeitigen Bewegung der Maus, die Kameraausrichtung in die gewünschte Richtung realisieren.

Darüber hinaus gibt es noch weitere Funktionalitäten für das Beschleunigen der Fortbewegung. Zum Einen können die Tasten: `Shift`, `STRG` und `ALT Left` in Kombination mit den Richtungstasten gedrückt werden, um den Geschwindigkeitsfaktor in der Fortbewegung zu erhöhen. Hierbei ist Shift der kleinste und Alt der größte Faktor. Zum Anderen kann mithilfe des Mausrads eine schnelle vor und zurück Bewegung ausgeführt werden.

Außerdem ist das dynamische Ändern der angezeigten Straßen möglich. Mit den Zahlentasten von 1 bis 10 lassen sich die unterschiedlichen Kategorien beim Betätigen ein- oder ausblenden. Die nachstehende Tabelle 4.1 zeigt, welche Straßenkategorien an die Tasten gebunden sind.

Taste	Straßenkategorie	Farbe
1	Autobahnen und ihre Anschlussstellen	Blau
2	Autobahnähnliche Straßen (auch „gelbe Autobahn‘ genannt)	Orange
3	Bundesstraßen und ihre Anschlussstellen	Gelb
4	Landesstraßen und ihre Auf- und Abfahrten	Gold Gelb
5	Kreisstraßen und ihre Auf- und Abfahrten	Weiß
6	Öffentlich befahrbare Nebenstraßen mit einfachem Ausbauzustand	Grau
7	Straßen an und in Wohngebieten	Rot
8	Verkehrsberuhigter Bereich	Pink
9	Erschließungswege und Straßen mit unbekannter Klassifikation	Schwarz
0	Wendestellen und Wendeschleifen	Violette

Tabelle 4.1: Straßenkategorien

4.6 Benchmarking

In diesem Kapitel wird das Benchmarking vom VbS anhand von vier verschiedenen Straßengraphen analysiert. Die Straßengraphen Deutschland-Österreich-Schweiz (DÖS), Deutschland, Baden-Württemberg und Mecklenburg-Vorpommern wurden hierbei von dem FMI zur Verfügung gestellt.

Das Benchmarking wurde auf folgendem System ausgeführt:

- Prozessor: Ryzen 5 2600X
- Grafikkarte: AMD RX 580 mit 8GB
- Arbeitsspeicher: 16 GB
- Betriebssystem: Windows 10

Die zehn Straßenkategorien aus Tabelle 4.1 auf Seite 50 wurden hierfür in 3 Gruppen zusammengefasst:

Gruppe	Straßenkategorien
1	Autobahnen und Schnellstraßen bestehend aus den ersten fünf Kategorien (Kategorien 1-5)
2	Straßen in Wohngebieten und Verkehrsberuhigte Bereiche (Kategorien 7,8 und 10)
3	Nebenstraßen, Einschließungswege und Straßen mit unbekannter Klassifikation (Kategorien 6 und 9)
Alle	Alle Straßenkategorien (Kategorien 1-10)

Tabelle 4.2: Gruppierung der Straßenkategorien

Für die Tests wurde der Globus um seine eigne Achse rotiert und parallel dazu wurde eine Abfolge von Bewegungssequenzen ausgeführt. Er wurde bezüglich der Frames Per Second (FPS) und der Speicherauslastung vom Arbeitsspeicher, als auch dem dedizierten Grafikspeicher getestet. Das Liniendiagramm in Abbildung 4.2 zeigt die FPS der unterschiedlichen Straßengraphen mit einem Bezug auf die unterteilten Straßenkategorien. Die Punkte in der Grafik sind mit der Anzahl der Nodes beschriftet, die der Straßengraph in einer bestimmten Gruppe hat. Zum Beispiel hat der Straßengraph DÖS nur mit Einblendung der ersten Gruppe 112 FPS bei 34.784.464 Knoten.

Aus dem Liniendiagramm wird deutlich, dass bei steigender Knotenanzahl die angezeigten FPS abnehmen. Ausgenommen sind hier die Straßengraphen von Baden-Württemberg und Mecklenburg-Vorpommern, die einen hohen FPS Wert haben und durch systemabhängige Prozesse leicht manipuliert werden können. Die Anzeige von 155.754.166 Knoten beim DÖS Graphen erfolgt mit knappen 30 FPS und liefert somit ein relativ flüssiges Ergebnis.

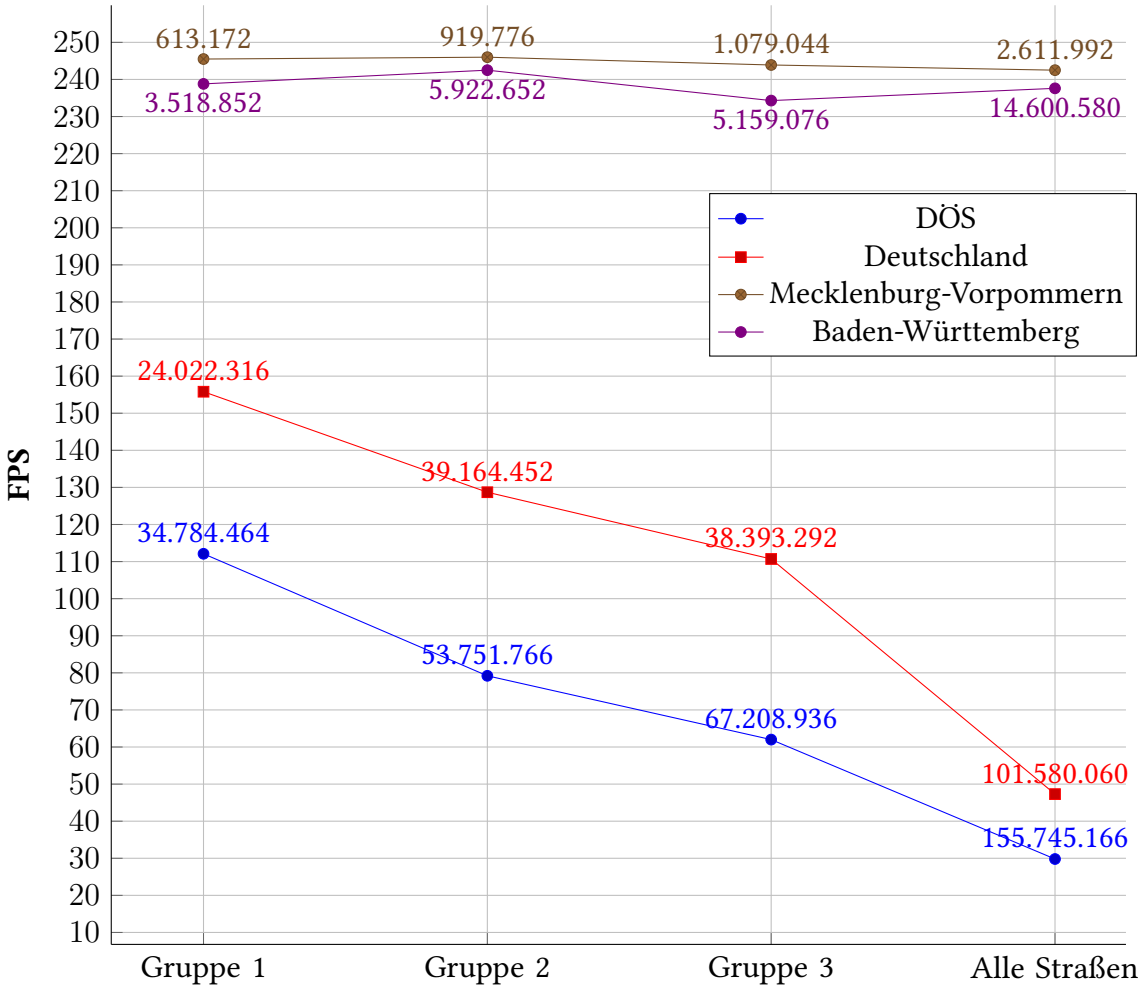


Abbildung 4.2: FPS Test

Zu bedenken ist hierbei, dass pro gezeichnete Kanten doppelt soviel Knoten benötigt werden. Dies hängt damit zusammen, dass die verwendete Topologie eine Line List ist, die zwei aufeinander folgende Knoten miteinander verbindet und anschließend die nächsten zwei Knoten in der Liste abhandelt. Außerdem werden bei dem VbS, Knoten die in mehreren Straßentypen vorkommen, auch mehrfach in den Vertex Puffer gelegt. Aus diesen Gründen hat die durch den VbS gezeichnete Deutschlandkarte insgesamt 101.580.060 Knoten, wobei die Germany.fmi Datei im Vergleich 25.115.477 Knoten und 50.790.030 Kanten hat.

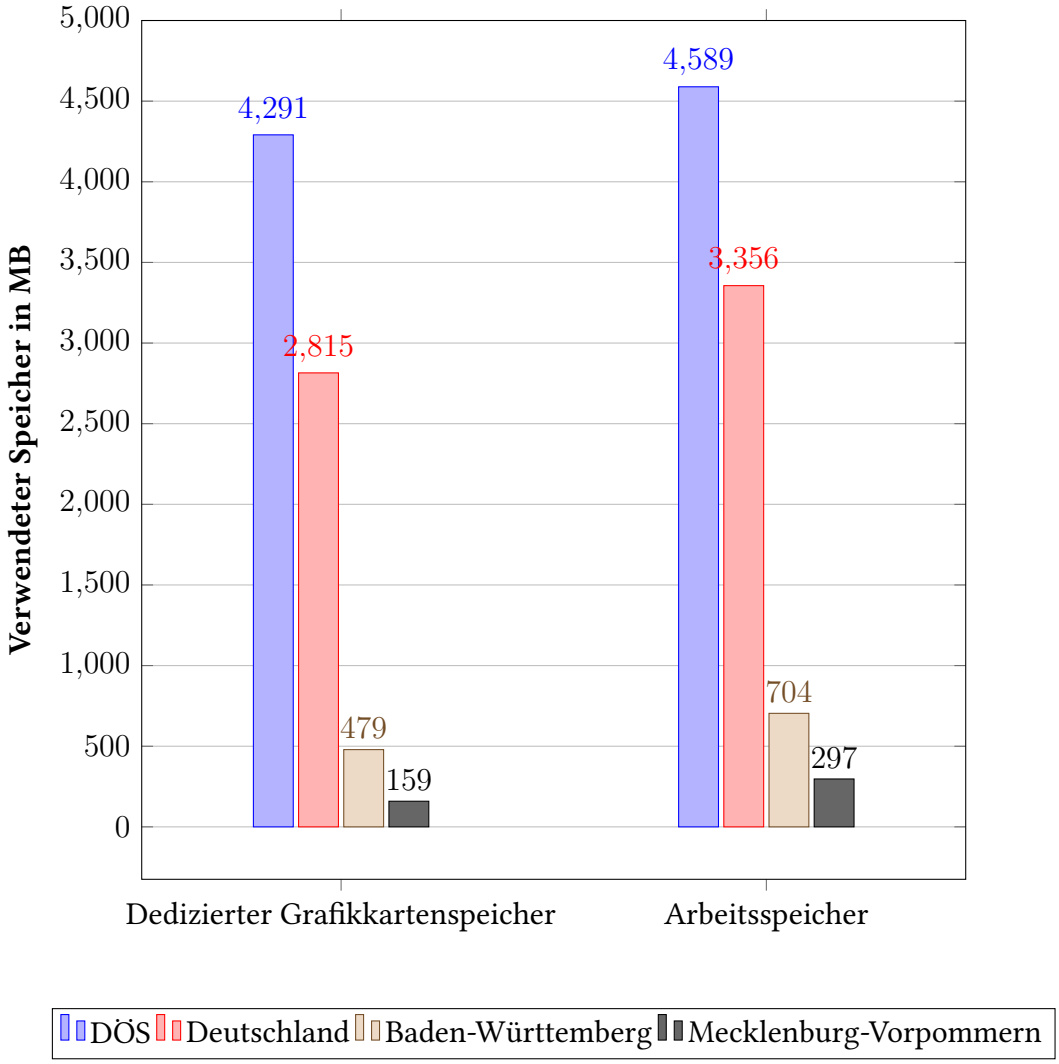


Abbildung 4.3: Speicherverbauch

Der Speicherverbauch vom VbS ist in Abbildung 4.3 dargestellt. Der VbS arbeitet zunächst auf dem Arbeitsspeicher, um die ganzen Vertex und Edge Daten einzulesen und zu verarbeiten. Anschließend werden die Vertex- und Indexdaten vom Arbeitsspeicher auf den dedizierten Grafikspeicher abgebildet. Dadurch kommt der nahezu identische Speicherverbauch zustande. Möchte man mit dem VbS einen Straßengraphen abbilden, ist darauf zu achten, dass genügend Speicher sowohl auf der Graffikkarte als auch auf dem Arbeitsspeicher vorhanden ist. Der verbrauchte Speicher entspricht ungefähr der Dateigröße des übergebenen Straßengraphen. Beispielsweise hat die Deutschlandkarte.fmi Datei eine Größe von 2.85 Gigabyte, im Vergleich hierzu werden 2.81 Gigabyte Speicher auf der Grafikkarte benötigt.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Vulkan basierter Straßengraph entwickelt und beschrieben. Hierfür wurden zunächst die Grundlagen erläutert, um die umfangreiche Vulkan API und die dem zugrundeliegende OSM Basis näher kennenzulernen. OSM ist die Datenquelle der VbS, und daher ist die Struktur der Daten erheblich für das Verständnis und die Integration in den Viewer.

Daraufhin wird der VbS in seine Elemente herunter gebrochen und die verwendeten Bibliotheken GLFW, GLM und Boost Spirit aufgezeigt.

Die Vulkan API ermöglicht einen Low-Level-Zugriff auf die PC-Hardware und bietet dem Programmierer damit viele Möglichkeiten seine Anwendung effizient zu entwickeln. Aufgrund der hardwarenahen Implementierung muss der Programmierer viele Komponenten genau definieren, zum Beispiel das Allokieren von Speicher oder die Bestimmung der zu verwendenden Queues auf der Grafikkarte. Einige der Komponenten wurden in dieser Bachelorarbeit anhand von Codeauszügen näher beschrieben.

Darüber hinaus wird erläutert, wie die VbS ausgeführt und bedient werden kann. Vorerst müssen alle Abhängigkeiten installiert werden, um dem Programm eine erfolgreiche Kompilierung zu gewähren. Im Anschluss kann der Viewer mit Maus und Tastatur verwendet werden.

Zuletzt werden Benchmarktests mit Bezug auf die Anzahl der FPS und die Höhe des Speicherverbauchs durchgeführt. Hier soll der VbS auf seine Effizienz und Leistung überprüft werden.

Ausblick

Die Zukunft der Vulkan API lässt sich schwer vorhersehen. Mit einem Blick auf andere gegenwärtige Grafik APIs, wie der DirectX 12 API von Microsoft oder dem Metal Grafikapi von Apple, unterscheidet sich Vulkan nicht besonders. Alle genannten APIs sind Hardware nah und beabsichtigen einen geringe CPU Auslastung durch die Minderung der Anzahl an Draw-Aurufen. Ein Draw-Aufruf enthält alle Informationen, die den Grafikprozessor über Texturen, Zustände, Shader, Rendering-Objekte, Puffer und weitere Komponenten informieren. Diese Informationen sind gekapselt als CPU-Arbeit, die Zeichenressourcen für die Grafikkarte vorbereitet. Doch in einem erheblichen Punkt unterscheidet sich Vulkan von den anderen Grafik APIs: Die Plattformunabhängigkeit.

Vulkan ist auch Gegenstand des vorgestellten Streetviewers. Hier ermöglicht die API mit einer effizienten Befehls- und Speicherverarbeitung eine flüssige Darstellung von Straßengraphen. Die Visualisierung basiert auf einer dreidimensionalen Globusdarstellung. Eine weiterführende Implementierung könnte sich mit Text- oder Flächendarstellung beschäftigen oder eine Überführung zu mobilen Endgeräten realisieren. Letzteres ist ein auch aktuelles Thema in der Spieleentwicklung für mobile Endgeräte, da Vulkan ebenfalls Android Geräte unterstützt. Die hardwarenahe Programmierung erfordert ein stabiles Grundwissen, kann jedoch für eine gesteigerte Effizienz in der Darstellung sorgen.

Literaturverzeichnis

[01] *Marble*. https://marble.kde.org. Eingesehen am 11.10.2019. URL: https://marble.kde.org (zitiert auf S. 15).

[02] KDE. *The Marble Handbook; Chapter 1. Introduction*. https://docs.kde.org/trunk5/en/kdeedu/marble/introduction.html. Eingesehen am 11.10.2019. URL: http://docs.kde.org/ (zitiert auf S. 15).

[03] Nienhüer, Dennis. *Worldwide and Offline Routing*. https://web.archive.org/web/20100729005700/http://nienhueser.de/blog/?p=137. Eingesehen am 11.10.2019 (zitiert auf S. 15).

[04] Srivastava, Siddharth. *KDE/Google Summer of Code 2010 (Part 1 of 2)*. https://dot.kde.org/2010/12/14/kdegoogle-summer-code-2010-part-1-2. Eingesehen am 11.10.2019. URL: http://docs.kde.org/ (zitiert auf S. 15).

[05] Nienhüer, Dennis. *Introducing Marble Touch*. https://web.archive.org/web/20111129025522/http://nienhueser.de/blog/?p=352. Eingesehen am 11.10.2019. URL: http://docs.kde.org/ (zitiert auf S. 15).

[06] *Leaflet*. Eingesehen am 11.10.2019. URL: https://leafletjs.com (zitiert auf S. 16, 17).

[07] *Vladimir Agafonkin GitHub Profile*. Eingesehen am 11.10.2019. URL: https://github.com/mourner (zitiert auf S. 16).

[08] *OpenStreetMap*. Eingesehen am 11.10.2019. URL: https://www.openstreetmap.de (zitiert auf S. 19).

[09] *OpenStreetMap - Deutschland FAQs*. Eingesehen am 11.10.2019. URL: https://www.openstreetmap.de/faq.html (zitiert auf S. 19).

[10] *Willkommen bei OpenStreetMap*. Eingesehen am 11.10.2019. URL: https://wiki.openstreetmap.org/wiki/Willkommen_bei_OpenStreetMap (zitiert auf S. 19).

[11] *OpenStreetMap Copyright and License*. Eingesehen am 11.10.2019. URL: https://www.openstreetmap.org/copyright/en (zitiert auf S. 19).

[12] *OpenStreetMap Foundation*. Eingesehen am 11.10.2019. URL: https://wiki.osmfoundation.org/wiki/Main_Page (zitiert auf S. 19).

[13] *OSM file formats*. Eingesehen am 12.10.2019. URL: https://wiki.openstreetmap.org/wiki/OSM_file_formats (zitiert auf S. 20).

[14] *LearnOSM - Dateiformate*. Eingesehen am 12.10.2019. URL: https://learnosm.org/de/osm-data/file-formats (zitiert auf S. 20).

[15] *Node*. Eingesehen am 18.10.2019. URL: https://wiki.openstreetmap.org/wiki/Node (zitiert auf S. 22).

[16] *Way*. Eingesehen am 18.10.2019. URL: https://wiki.openstreetmap.org/wiki/Way (zitiert auf S. 23).

[17] *Relation*. Eingesehen am 18.10.2019. URL: https://wiki.openstreetmap.org/wiki/Relation (zitiert auf S. 23).

[18] *Elements*. Eingesehen am 18.10.2019. URL: https://wiki.openstreetmap.org/wiki/Elements (zitiert auf S. 24).

[1919] A. Overvoorde. „Vulkan Tutorial“. In: 2019. Kap. Overview, S. 10 (zitiert auf S. 26).

[20] *GLFW*. Eingesehen am 30.10.2019. URL: https://www.glfw.org (zitiert auf S. 36).

[21] *OpenGl Mathematics*. Eingesehen am 30.10.2019. URL: https://glm.g-truc.net/0.9.9/index.html (zitiert auf S. 37).

[22] *Matrices*. Eingesehen am 30.10.2019. URL: https://open.gl/transformations (zitiert auf S. 37).

[23] *GLFW Documentation*. Eingesehen am 30.10.2019. URL: https://www.glfw.org/docs/3.2 (zitiert auf S. 41, 48).

[2419] T. K. V. W. Group. „Vulkan® 1.1.126 - A Specification (with all registered Vulkan extensions)“. In: 2019, S. 2110 (zitiert auf S. 26, 28–34).

[25] Sawicki, Adam. *Understanding Vulkan objects*. Eingesehen am 30.10.2019. URL: https://gpuopen.com/understanding-vulkan-objects/ (zitiert auf S. 26–34).

[26] *Khronos Group*. Eingesehen am 30.10.2019. URL: https://www.khronos.org (zitiert auf S. 25).

[27] *Marble Vision*. Eingesehen am 11.10.2019. URL: https://marble.kde.org/about.php (zitiert auf S. 15).

Alle URLs wurden zuletzt am 07. 11. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift