Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Erstellung eines interaktiven, 3D Höhenmodells des Mars auf Grundlage von MOLA Daten

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang Angewandte Informatik

1. Gutachter: Prof. Dr.-Ing. Thomas Jung
2. Gutachter: Michael Mario Droste

Eingereicht von Tim Oelkers

15. Juli 2021

Kurzbeschreibung

Inhaltsverzeichnis

Kurzbeschreibung i

1. Einführung 1
- 1.1. Motivation . . . 1
- 1.2. Zielstellung . . . 1
- 1.3. Aufbau der Arbeit . . . 2

2. Grundlagen 3
- 2.1. Datenquellen . . . 3
- 2.2. OpenGL . . . 3
- 2.3. Datenreduzierung . . . 3
  - 2.3.1. Datenmenge . . . 3
  - 2.3.2. Verlustfrei . . . 4
  - 2.3.3. Verlustbehaftet . . . 5
  - 2.3.4. Redundanzentfernung . . . 5

3. Methodologie 7

4. Anforderungsanalyse 8
- 4.1. Use Case . . . 8
- 4.2. Ist-Zustand . . . 8
- 4.3. Anforderungen . . . 8

5. Konzeption 9
- 5.1. Programmiersprache . . . 9
  - 5.1.1. Java . . . 9
  - 5.1.2. C++ . . . 10
  - 5.1.3. JavaScript/TypeScript . . . 10
  - 5.1.4. Fazit . . . 11
- 5.2. Bibliotheken . . . 11
  - 5.2.1. Webserver . . . 11
  - 5.2.2. Grafik/Mathematik . . . 12
- 5.3. Architektur . . . 12
- 5.4. User Interface . . . 13

6. Implementierung 14
- 6.1. Build System . . . 14
- 6.2. Visualisierung . . . 15
- 6.3. Kamerabewegung . . . 16

7. Evaluation 17

8. Fazit **18**

A. Appendix **19**
A.1. Redundanzberechnung 19

Abbildungsverzeichnis

2.1. Indexierung eines Raster-Modells 4

1. Einführung

1.1. Motivation

Der Mars als unser Nachbarplanet ist, aufgrund seines Klimas und der Nähe, der am Besten für eine Besiedelung geeignetste, erdähnliche Planet in unserem Sonnensystem. Auch ist er das Ziel unzähliger Forschungen, insbesondere zu der Frage ob Leben außerhalb der Erde existieren kann. Aktuell findet ein kleiner Wettlauf zum Mars statt. Der *Perseverance Rover* der NASA landete am 18.02.2021 und *Tianwen-1* aus China am 14.05.2021. Des Weiteren ist derzeit die Raumsonde *al-Amal* aus den Vereinigten Arabischen Emiraten in der Mars-Umlaufbahn, eine Landung ist allerdings nicht vorgesehen. Des Weiteren planen auch Japan und Indien eine Rover Mission in den nächsten Jahren. Allerdings sind Mars-Missionen kein einfaches Unterfangen, von den letzten 50 Mission schlugen 29 zumindest teilweise fehl[1].

Eine Visualisierung hat vor allem informative Gründe. Dieses Projekt versucht interessierten Menschen zum Beispiel die Probleme einer Mars-Besiedelung oder allein schon die Probleme einer Landung vor Augen zu führen. Vor allem, da zwar einige 2D Visualisierungen mit derselben Datengrundlage existieren, echte 3D Visualisierungen aber schwer zu finden sind. Die Desktop-Version von Google Earth zum Beispiel besitzt zwar ein etwas verstecktes 3D Mars-Modell, die Ansicht ist allerdings auf eine Betrachtung von oben herab beschränkt, sodass der 3D Effekt hier kaum zur Geltung kommt.

1.2. Zielstellung

Im Zuge dieser Arbeit soll ein Prototyp geschaffen werden, welcher die Oberflächenstruktur des Mars in höchstem Detailgrad als 3D Modell darstellt. Der Nutzer soll möglichst ungehindert die Oberfläche aus verschiedenen Winkeln und Zoomstufen betrachten können. Ein kleines User Interface soll Informationen über den aktuellen Ort darstellen und den Nutzer bei der Navigation unterstützen.

Die Frage, die diese Arbeit mit dem Projekt zu beantworten versucht, ist, welche technischen Möglichkeiten existieren um mit dem Problem der Datenmenge (siehe Abschnitt 2.3.1) fertig zu werden und ob eine so detailgetreue Darstellung aus Nutzersicht überhaupt sinnvoll ist. Vor allem, da eine Verbesserung des Details Grades natürlich immer mit einer Verschlechterung der Performance in Verbindung gesetzt werden muss. Hierbei sollen verschiedene Möglichkeiten, sowohl verlustfrei als auch verlustbehaftet, implementiert und durch quantitative als auch qualitative/empirische Methoden evaluiert werden.

[1] `https://de.wikipedia.org/wiki/Chronologie_der_Marsmissionen`

1.3. Aufbau der Arbeit

2. Grundlagen

2.1. Datenquellen

Die Datengrundlage für dieses Projekt sind Daten des Mars Orbiter Laser Altimeter (MOLA), einem Höhenmessgerät an Bord des

2.2. OpenGL

2.3. Datenreduzierung

2.3.1. Datenmenge

Der beschriebene MOLA-Datensatz besitzt eine Breite von 46080 Pixeln und eine Höhe von 23040 Pixeln. Jeder Pixel ist dabei 16 bit groß, sodass die Rohdaten an sich eine Größe von 1,98 GB besitzen[3]. Um daraus ein 3D-Modell zu erstellen, müssen neben den Höhenwerten natürlich auch die x- und z-Position erfasst werden, welche sich aus dem Rasterformat der Daten errechnen lassen. Erschwerend hinzu kommt, dass der kleinste numerische Datentyp in GLSL 32 bit groß ist[2, Abschnitt 4.1, S. 23] und sich die Höhendaten so nicht effizient speichern lassen. Eine entsprechende Erweiterung (*extension*) von GLSL um 16 bit Typen mit dem Namen EXT_shader_16bit_storage existiert, befindet sich allerdings erst in der Entwurfsphase und kann daher nicht genutzt werden. Die reinen Vertex-Daten besitzen also eine Größe von 11,88 GB. Weitere Daten pro Vertex (zum Beispiel Normalenvektoren) werden für dieses Projekt nicht benötigt. Allerdings können diese Vertex-Daten in dieser Form nicht an OpenGL übergeben werden, da erst Polygone (OpenGL Primitives) daraus erstellt werden müssen. Im einfachsten Fall sind dies für ein Rastermodell natürlich Vierecke (GL_QUAD), diese Form ist allerdings veraltet und sollten nicht mehr benutzt werden. Stattdessen müssen die Vertex-Daten Dreiecke (GL_TRIANGLE) beschreiben. Da in einem Rastermodell aus Dreiecken natürlich ein Vertex 6 mal wiederholt werden müsste, gibt es eine effizientere Beschreibung der Polygone: Indexierung. Dabei wird ein Array an OpenGL übergeben, welche die Reihenfolge der Vertices als deren Position im ursprünglichen Vertex-Array kodiert (siehe Abbildung 2.1).

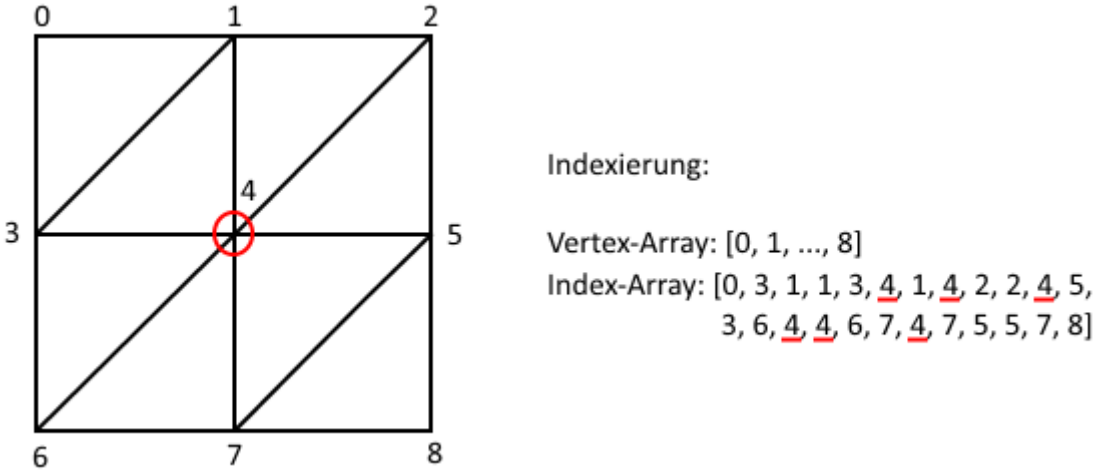

Abbildung 2.1.: Indexierung eines Raster-Modells

Durch diese Technik wird das Wiederholen eines Vertex verhindert, allerdings ist auch hier ein Speicherverbrauch zu berechnen. Der Index entspricht dabei einer Größe von 32 bit und die Anzahl lässt sich mit folgender Formel berechnen:

$$Anzahl = 6 * (Höhe - 1) * (Breite - 1) = 6.369.684.486$$

. Dadurch werden weitere 23,73 GB benötigt, wodurch der Gesamtspeicherverbrauch auf 35,61 GB steigt. Da dies, auch ohne nicht-funktionale Anforderungen definiert zu haben, nicht in einen durchschnittlichen Grafikspeicher passt, sind Maßnahmen zur Reduzierung der Daten zwingend erforderlich.

2.3.2. Verlustfrei

Das Durchführen von verlustfreien Methoden zur Reduzierung der Datenmenge ist natürlich immer den verlustbehafteten Methoden vorzuziehen, da ... Die einfachste Möglichkeit, ist die Entfernung von Redundanzen. Diese können auf Grund des Rasterformats der Daten vorhanden sein, da die Abstände zwischen den Datenpixeln laut Definition einem festen Wert entsprechen müssen. Die hohe horizontale Datenauflösung von 463m bei einer natürlichen Topographie führt zu der statistischen Annahme, dass dabei Redundanzen auftreten. Hier stellt sich die Frage, was eigentlich eine Redundanz im Kontext von 2D Höhenwerten ist. Ein Auffinden von gleichen Werten in einem 3x3 Raster ist der einfachste Fall. Wichtig hierbei ist, dass nicht etwa ein einfaches Wiederholen von Werten bereits eine Redundanz ist, da auch das Darstellen einer flachen Ebene eine wichtige Information ist. Allerdings können Redundanzen auch bei einer geneigten Ebene auftreten, bei der sich benachbarte Höhenwerte unterscheiden. Der generelle Fall zur Beschreibung ist das Vorhandensein einer linearen Abhängigkeit zwischen Werten in **allen** Spalten und Reihen eines mindestens 3x3 großen Rasters[4]. Wichtig hierbei ist, dass immer der 2D Kontext beachtet werden muss. Es können sich beliebig viele Werte in einer Reihe oder Spalte wiederholen, ohne das dies eine Redundanz darstellt, wenn auch nur eine Spalte oder Reihe im Raster diese lineare Abhängigkeit nicht aufweist. Um das Vorhandensein von Redundanzen im MOLA Datensatz zu beweisen, wurde ein Script

erstellt, welches im Abschnitt A.1 genauer beschrieben ist. Dies zeigt, dass insgesamt 68.264.527 Datenpixel redundant sind, was einem Prozentsatz von 6,43% entspricht. Dabei existieren 8.708.542 redundante Teilraster, das größte von ihnen besitzt immerhin eine Fläche von 277.922km².

Eine andere Möglichkeit eine verlustfreie Reduktion zu erreichen, ist die Daten zu entfernen, die vom Benutzer nicht gesehen werden können. Das betrifft explizit nicht den Punkt, dass Unterschiede auf Grund der Physiologie des menschlichen Auges nicht wahrgenommen werden können, da dies trotzdem einem theoretischen Informationsverlust entspricht.

2.3.3. Verlustbehaftet

Das Durchführen von verlustfreien Methoden allein reicht nicht aus, um den Speicherverbrauch auf ein akzeptables Niveau zu senken.

2.3.4. Redundanzentfernung

Bevor ein geeigneter Algorithmus gefunden werden kann, müssen einige Probleme mit der Entfernung von Redundanzen besprochen werden. Zum einen besteht das Problem, die gefundenen Lücken in einem Datenformat darzustellen. Normalerweise besteht immer der gleiche Abstand zwischen zwei benachbarten Datenpunkte, wenn jetzt einfach Vertices entfernt werden, dann ändern sich die Abstände und dies muss irgendwie kodiert werden. Eine Möglichkeit dies zu umgehen, ist die Entfernung zur Laufzeit durchzuführen und direkt bei der Erstellung redundante Vertices zu erkennen. Da der hier verwendete Algorithmus allerdings sehr kostenaufwendig ist, wurde dieser Ansatz verworfen. Stattdessen werden die Lücken durch vordefinierte Werte kodiert, die nicht in den originalen Daten vorhanden sind. Im konkreten Fall wurde der kleinstmögliche 16bit Wert gewählt.

Zum anderen besteht das Problem der Indexierung des Modells. In einem reinen Raster-Modell lassen sich die Indices trivial berechnen, da sie immer einem gleichmäßigem Muster folgen. Jetzt existieren natürlich Lücken in den Vertex-Daten und eine neues Polygon Netz muss gefunden werden. Ein Problem dabei ist, dass benachbarte Vertices sich mit den Rändern der Lücken verbinden müssen (siehe Abbildung x). Auch wenn die Ränder also redundante Daten aufweisen, müssen sie für die Erzeugung eines korrekt aussehenden Modells zugelassen werden. Anstatt also nur die Eckpunkte zu erhalten, müssen auch alle Ränder der Redundanzen erhalten bleiben, was den Prozentsatz der redundanten Vertices auf 1,87% drückt. Für die Berechnung der Indices wurde folgende Formel genutzt: TODO.

Wie beschrieben, ist die Unterteilung des Gesamtmodells in Abschnitte ein zentraler Aspekt der Anwendung um das *frustum-* und *occlusion culling* durchführen zu können. Die führt aber zu einer weiteren Einschränkung bei der Entfernung von Redundanzen, die über Abschnittsgrenzen hinaus gehen würden. Die müssen an den Rändern der Abschnitte definiert sein, um sich mit benachbarten Abschnitten verbinden zu können. Auch sollen per Design Abschnitte keinen Zugriff auf Vertex-Daten anderer Abschnitte erhalten (schon allein um Daten nicht unnötig im RAM zu halten). Dies führt dazu, dass der Algorithmus zusätzlich redundante

Teilraster an den Abschnittsgrenzen trennen muss. Dies verringert den Prozentsatz allerdings nur minimal auf 1,86%.

Nachdem nun Anforderungen an den Algorithmus definiert wurden, muss das Problem der hohen Komplexität angegangen werden. Die optimale Lösung ist durch die größtmögliche Entfernung von Vertices definiert, was auch durch eine möglichst große Summe des Flächeninhalts aller Teilraster beschrieben werden kann. Das Problem ist, dass während der Laufzeit nicht sichtbar ist, welche Konfiguration der Raster zur größtmöglichen Fläche führt. Es kann durchaus sein, dass in einem Teilschritt ein kleineres Raster gebildet werden muss, damit übrig gebliebene Vertices Teil eines deutlich größeren Rasters werden. Das diesem Problem am nächsten stehende bekannte Problem aus der Informatik ist das Problem des *rectangle packing*, einer 2D Variation des Rucksack-Problems, welches der Komplexitätsklasse NP-Vollständig zugerechnet wird. Die Überprüfung ob eine Lösung optimal ist, erfordert dabei das Überprüfen aller anderen Lösungen, was eine exponentielle Zeit benötigt. Allerdings sind im konkreten Fall einige Einschränkungen vorhanden, die die Komplexität verringern. Zum einen können die Rechtecke nicht an jede Stelle gepackt werden, sondern nur in den Flächen, die als Redundanzen in Frage kommen. Zum anderen ist die Mindestgröße eines Rechtecks 3x3, sodass sich die Anzahl der Möglichkeiten um den Faktor 9 verringert. Unter der Annahme, dass die aktuelle, nicht optimale Berechnung die Anzahl der Gesamtredundanzen mit ungefähr 2% widerspiegelt, ist die Komplexität:

$$O(n) = 2^n/9 * 2/100 = 2.75 * 10^{319.598.486}$$

.

Für

```
procedure findRedundancies(data)
    result, visited := boolean array with false values;
    for all x, z in data do
        if not visited[z][x] then
            rowsWidth, rowsHeight := checkRows(data, x, z);
            columnsWidth, columnsHeight := checkColumns(data, x, z);
            width, height := Minimum of both widths and heights;
            if width, height > 3 then
                for all xx, zz in width, height do
                    if not (0, 0 or 0, height or width, 0 or width, height) then
                        result[z + zz][x + xx] := true
                    end if
                    visited[z + zz][x + xx] := true
                end for
            end if
        end if
    end for
    return result
end procedure
```

3. Methodologie

4. Anforderungsanalyse

4.1. Use Case

Die Zielgruppe dieses Projekts lässt sich relativ schwer eingrenzen. Es betrifft vor allem Personen mit Interesse in Raumfahrt oder dem Weltraum allgemein. Wichtig ist, das diese Personen keinen Bezug zur Informatik aufweisen müssen und zum Beispiel die Bedienung des User Interface daher kein Spezialwissen erfordern darf. Auch die Visualisierung der Oberfläche muss ohne Vorkenntnisse verständlich sein und dabei trotzdem noch die eigentliche Oberfläche korrekt repräsentieren.

4.2. Ist-Zustand

4.3. Anforderungen

5. Konzeption

5.1. Programmiersprache

Die Wahl der richtigen Programmiersprache ist mehr als nur reine Präferenz, da jede Sprache projektrelevante Eigenheiten hat. Im folgenden werden daher drei Programmiersprachen auf ihre Projekttauglichkeit evaluiert. Für dieses Projekt irrelevante Aspekte wie zum Beispiel Sprachfeatures oder Paradigmen fließen in die Bewertung nicht mit ein.

Der wichtigste Faktor aus Projektsicht ist das Zielsystem, auf die Software schlussendlich laufen wird. Idealerweise sollte es die Sprache erlauben die Software, ohne große Veränderungen, auf eine andere Plattform zu portieren. Hier sind dann Betriebssystem- und Hardware-Unabhängigkeit von großer Bedeutung um dies zu erleichtern. Ein weiterer Aspekt ist die Geschwindigkeit, auch wenn sie einen deutlich kleineren Stellenwert einnimmt, da der Einfluss einer Sprache auf die Gesamtperformance der Anwendung äußerst gering ist und dort wichtigere Aspekte bevorzugt werden sollten[1]. Ein weiterer wichtiger Punkt ist das Vorhandensein von Tools, welche den Entwicklungsprozess unterstützen können. Hier sind zum Beispiel Debugger, Profiler oder Analyse-Tools zu nennen, mit denen die Performance der Anwendung evaluiert werden kann. Auch das Vorhandensein von Frameworks ist nicht zu unterschätzen, da sie zur Einhaltung des Projektzeitraumes benötigt werden und gerade im Bereich der Grafikprogrammierung nicht für jede Sprache existieren. Schlussendlich müssen auch persönliche Fähigkeiten und Erfahrungen genannt werden. Das Lernen einer neuen Sprache ist zeitintensiv und führt außerdem zu einer erhöhten Fehleranfälligkeit.

5.1.1. Java

Java ist eine objektorientierte Sprache mit sehr hohem Abstraktionsniveau. Sie wird in einer virtuellen Maschine namens Java Virtual Machine (JVM) ausgeführt und ist somit Betriebssystem- und Hardware unabhängig. Dadurch ist das Kriterium der guten Verfügbarkeit erfüllt und eine möglichst große Zielgruppe kann angesprochen werden. Insbesondere Smartphones sind hier als Zielplattform zu erwähnen, da hier die Entwicklung von Apps mit Java favorisiert wird. Auch das Motto “Write Once, Runs Everywhere”, welches sich unter anderem in der Rückwärtskompatibilität bis zur ersten Java Version von vor über 25 Jahren manifestiert, ist für die meisten Softwareprojekte, inklusive diesem, sehr vorteilhaft, da keine Änderungen am Code notwendig sind und die Software weiterhin auf neueren JVMs lauffähig bleibt.

Allerdings beeinträchtigt die Nutzung einer virtuellen Maschine die Nutzung von

[1] “premature optimization is the root of all evil” - Sir Tony Hoare

nativen, also Betriebssystem abhängigen Ressourcen außerhalb dieser isolierten Umgebung, insbesondere den OpenGL Implementationen. Unter anderem verwaltet die JVM jeglichen Speicher, was eine Übergabe von zum Beispiel Speicheradressen (Pointer) von Objekten im Java Heap an OpenGL unmöglich macht. Auch die verfügbaren Referenzen existieren natürlich nur im Kontext der virtuellen Maschine. Als Lösung bietet Java die Nutzung von sogenannten direct buffer an, welche außerhalb der Java Heaps liegen und nicht von der JVM verwaltet werden[2]. Hier ist dann ein ständiges Kopieren zwischen Heap und Buffer notwendig, was vor allem bei der Übergabe großer Vertex Daten problematisch ist. Hier ist viel manuelles Kopieren und das Serialisieren von komplexen Objekten in die Buffer notwendig. Auch die allgemeine Kommunikation mit dem OpenGL Treiber besitzt einen Performance Overhead, der nicht zu vernachlässigen ist.

Aufgrund des Alters der Sprache und seiner hohen Beliebtheit[3] existiert ein sehr große Community im Umfeld der Sprache, welcher auch oft als Java Ecosystem beschrieben wird. Dies zeigt sich in einem sehr guten Tooling-Support und einer Implementation der meisten größeren Frameworks in dieser Sprache. Abschließend kann auch die hohe persönliche Expertise mit dieser Sprache als Vorteil genannt werden.

5.1.2. C++

C++ ist eine objektorientierte Sprache mit niedrigerem Abstraktionsniveau als Java. Sie besitzt keine automatische Speicherverwaltung (keine Garbage Collection und explizite Speicher-Allokationen und -Referenzierung). Auch wird sie direkt zu Maschinencode kompiliert und direkt ausgeführt, sodass sie im Allgemeinen als schneller angesehen wird. Die heißt aber auch, dass für jedes Betriebsystem (bzw. sogar für jede CPU) ein eigenes Kompilat erzeugt werden muss. Auch das Web und Smarthphones können nicht ohne weiteres mit dieser Sprache angesprochen werden, was zu einer Verkleinerung der Zielgruppe führt.

Eine kurze Suche nach dem Begriff OpenGL in Repositories auf github.com zeigt, dass sie mit Abstand die meistgenutzte Sprache bei der Entwicklung von Grafikanwendungen mit OpenGL ist[4]. Sie besitzt mit über 27.800 von 60.600 Resultaten einen Anteil von über 45%. Java dagegen besitzt mit über 5000 Resultaten nur einen Anteil von 8%. Aufgrund der hohen Nutzung der Sprache in der Grafikprogrammierung existieren sehr viele Frameworks, die für dieses Projekt benötigt werden könnten. Auch erleichtert das

5.1.3. JavaScript/TypeScript

JavaScript ist eine schwach und dynamisch typisierte Skriptsprache, welche unter anderem im Browser ausgeführt wird. Da ein Browser auf nahezu jedem Computer oder Smartphone verfügbar ist, ist auch hier keine Einschränkung der Zielgruppe notwendig. Die Verbreitung kann sogar minimal größer als Java angesehen werden, da

[2] `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/ByteBuffer.html`

[3] seit 2001 in den Top 3 der beliebtesten Sprachen: `https://www.tiobe.com/tiobe-index/java/`

[4] `https://github.com/search?q=opengl&type=Repositories`

keine Installation einer JVM notwendig ist. Es existiert auch eine Sprache namens TypeScript, welche die Sprache um statische Typisierung erweitert. Die Sprache wird dabei in JavaScript transpiliert[5], sodass die Ausführung im Browser weiterhin funktioniert. Das heißt aber auch, dass die Typensicherheit nicht zur Laufzeit überprüft werden kann.

Der größte Vorteil bei der Nutzung eines Browsers ist die sehr einfache Gestaltung von Benutzeroberflächen (UI). Mit Hilfe von HTML und CSS ... Integration von JavaScript

HTML + CSS -¿ einfache UI

gleiche Nachteile wie Java mit der isolierten Umgebung -¿ allerdings WebGL

5.1.4. Fazit

Trotz der Nachteile wurde das Web als Zielplattform ausgewählt. Die Hauptursachen dafür sind die hohe Verfügbarkeit und die einfache UI Gestaltung. Als konkrete Sprache wurde TypeScript gewählt, da die eine starke Typisierung das Arbeiten mit einer unbekannten API stark erleichtert.

5.2. Bibliotheken

5.2.1. Webserver

Die Aufgabe des Webserver ist das Bereitstellen der statischen Webseite, die die Hauptanwendung mit der Visualisierung beinhaltet, und das Bereitstellen der Höhendaten zu bestimmten Koordinaten. Aus diesen Aufgaben ergibt sich ein sehr einfaches Anforderungsprofil an die Webserver Bibliothek und insbesondere Features, welche oftmals mit Webservern assoziiert werden, werden nicht benötigt. Dazu zählen zum Beispiel Benutzerauthentifizierung, Anfrageratenlimitierung, Firewalls, Proxies oder andere Sicherheitsaspekte. Weitere Aspekte, welche nicht benötigt werden, sind das Bereitstellen und Empfangen von Dateien, das Automatische Parsing von JSON in Java Objekte und anders herum, Routing von bestimmten Requests zu bestimmten Endpoints oder die Unterstützung von Websockets. Auch das Bereitstellen einer verschlüsselten Kommunikation über HTTPS ist im konkreten Anwendungsfall nicht nötig, da keine nutzerbezogene Daten vorhanden sind. Da viele Nutzer und Browser das Fehlen dieses Features als Sicherheitsrisiko sehen und die Anwendung dadurch potentiell meiden, wird das Vorhandensein dieses Features trotzdem als positiv bewertet.

Als wichtig erachtet wird als erstes eine geeignete Lizenz. Diese muss eine kostenlose Nutzung, potentiell auch in einem kommerziellem Umfeld, erlauben und darf einer Weiternutzung in OpenSource Projekten nicht im Wege stehen. Idealerweise sollte die Bibliothek selbst als OpenSource Projekt entwickelt werden, da so Probleme und Features transparenter sind, was die Wahl und die spätere Entwicklung vereinfacht. Ein weiterer positiver Aspekt ist die Unterstützung von HTTP/2 oder sogar

[5] im Gegensatz zum Kompilieren findet keine Verringerung des Abstraktionslevels statt

HTTP/3. Diese bringen durch verbesserte Features wie Header Kompression, Multiplexing über eine geteilte TCP Verbindung oder das Senden von Ressourcen ohne vorigen Request (HTTP/2 Server Push) schon allein durch die Nutzung einen Performancegewinn, ohne dass Änderungen an der Anwendung notwendig sind. Des Weiteren soll der Webserver als eigenständige Java Anwendung (JAR) laufen können und kein manuelles Deployment in ein Application Server wie Tomcat benötigen (siehe Java Servlet Technologie). Das die Anfragen in einer nicht blockierenden Weise abgearbeitet werden können, sollte natürlich selbstverständlich sein, um einen Mehrbenutzerbetrieb auf vernünftige Weise zu ermöglichen.

Mit diesen Anforderungen soll ein Bibliothek gefunden werden, welche einen möglichst minimalen Umfang besitzt und nur die hier beschriebenen Features unterstützt. Als erstes wurde die Möglichkeit evaluiert, überhaupt keine externen Abhängigkeiten zu nutzen und mit den Java internen Klassen die Anforderungen umzusetzen. Dazu zählt zum einen die direkte Verwendung von Sockets ... Des Weiteren ist in den meisten JDK Implementationen das package com.sun.net.httpserver enthalten, welches einen vollwertigen Webserver enthält. Dieser entspricht zwar, neben der Unterstützung von HTTP/2, fast allen Anforderungen, allerdings ist dieses package nicht Teil des Java Standards. Es ist zwar in den beiden großen Implementationen Oracle JDK und OpenJDK enthalten, allerdings kann sich das jederzeit ändern. Da die Anwendung auch für spätere Java Versionen oder sogar andere JDK Implementationen portierbar bleiben soll, wurde diese Möglichkeit ausgeschlossen.

5.2.2. Grafik/Mathematik

Ein ursprünglicher Gedanke hinter diesem Projekt war die direkte Nutzung von OpenGL, ohne weitere Abhängigkeiten. Insbesondere sollten keine vollständigen Grafikengines wie Unity oder Unreal verwendet werden, da sie das Projekt unnötig aufgebläht hätten und die Nutzung dieser das Arbeiten auf geringerem Abstraktionsniveau (prozedurales Erstellen von Modellen oder das gezielte Steuern des Ladevorgangs) erschwert hätten. Außerdem erschweren diese Engines das Deployment im Web, da teilweise eigene Laufzeitumgebungen (z.B. Unity Web Player) bereitgestellt werden müssen [CITE NEEDED]. Trotzdem wurde nach einigem Prototyping entschieden, zumindest eine minimale Unterstützung durch eine Bibliothek zu nutzen. Dies ist vor allem durch den relativ großen Arbeitsaufwand bei der Verwendung von reinem OpenGL begründet. Hier müssen selbst für kleine Programme mehrere hundert Zeilen Code geschrieben werden und [BOILERPLATE].

5.3. Architektur

Die Architektur der Anwendung gliedert sich in das Frontend und das Backend. Vor allem das Frontend ist stark durch die Einschränkung beeinflusst, dass jegliche OpenGL Kommandos von dem Thread ausgeführt werden müssen, welcher aktuell den OpenGL Kontext besitzt. Dies ist normalerweise der Thread, welcher ihn ursprünglich angelegt hatte. Es ist zwar möglich, den Kontext zwischen Threads zu wechseln, dies ist jedoch einer der teuersten Operationen in OpenGL und macht

meistens die Performance Verbesserungen durch die Nutzung von Threads zunichte. Gleichzeitig sollen natürlich Abschnitte, welche nichts mit OpenGL zu tun haben, so gut es geht parallelisiert werden. Daraus folgt, dass oftmals eine Synchronisation zwischen verschiedenen Threads notwendig ist, damit diese sich wieder in den Hauptthread vereinen können.

Ein weiterer Kernaspekt der Architektur ist die Unterteilung der Welt in Abschnitte, im folgenden auch oft als chunks bezeichnet. Dies ist bedingt durch die große Datenmenge, welches ein vollständiges Laden in den Arbeits-/Grafikspeicher nicht möglich macht (siehe TODO). Es wurde sich für die Nutzung von chunks in festen Größen, im konkreten TODO, entschieden, da es

In Abbildung 1 ist ein Aktivitätsdiagramm dargestellt, welches den geplanten Ablauf der Welterstellung auf Client-Seite grob darstellt. Die Aktivität wird durch jede Veränderung der Kamera durch den Benutzer gestartet. Dazu zählt die Translation durch die Benutzung der Pfeiltasten, die Rotation durch die Benutzung der Maus und die Skalierung oder Zoom durch die Benutzung des Mausrads. Als erstes wird eine Liste mit allen möglichen chunks berechnet, welche den Sichtbereich der Kamera (Frustum) auch nur ansatzweise schneiden. Diese Liste wird verglichen mit einer Liste aller aktuellen chunks, um neue und nicht mehr benötigte Abschnitte zu ermitteln. Die rohen Höhendaten des Abschnitt werden vom Server angefragt. Da dies noch keine Interaktion mit OpenGL benötigt und potentiell mehrere Abschnitte angefragt und transformiert werden müssen, habe ich hier eine erste Parallelisierung, durch die Nutzung von Web Worker, geplant. Als nächstes müssen die Höhendaten in dreidimensionale Koordinaten auf einer Kugel transformiert werden. Die Formel dafür lautet TODO. Abschnitte, welche aktuell vorhanden sind, aber nicht mehr sichtbar sind, müssen vollständig entladen werden. Hierbei ist darauf zu achten, auch die Ressourcen auf GPU Seite frei zu stellen, da dies natürlich nicht durch die automatische garbage collection auf JavaScript Seite abgedeckt ist.

Ein weiteres Problem, was durch die Benutzung von chunks auftritt, ist das TODO [inklusive Bild von nicht verbundenen chunks]

Chunks, welche aktuell vorhanden sind, aber nicht mehr in der Liste der sichtbaren chunks, müssen

5.4. User Interface

6. Implementierung

6.1. Build System

Der Build Prozess ist ein Prozess, welcher alle Schritte beinhaltet um aus dem vorhandenen Source Code eine ausführbare Software zu erstellen. Insbesondere werden dabei benötigte Abhängigkeiten zur Verfügung gestellt und in die Software integriert, der Source Code kompiliert und alle Software Teile in einem ausführbaren Format zusammengeführt. Dieser Prozess kann sehr komplex werden und ist daher häufig fehleranfällig. Unter anderem die Kombination von mehreren Build Systemen und unterschiedlichen Programmierumgebungen wird dabei als ein Kernproblem angesehen.

Da sich dieses Projekt in ein Frontend und Backend mit unterschiedlichen Programmiersprachen gliedert, ist es allerdings notwendig unterschiedliche Build Systeme zu integrieren. Dies ist erforderlich, da die existierenden Systeme sich eher auf eine Umgebung spezialisieren und zum Beispiel nur Abhängigkeiten einer Programmiersprache in einem zentralen Repository anbieten (z.B. npm registry für JavaScript oder maven central für Java Anwendungen). Auch existieren bestimmte Plugins, zum Beispiel für das Transpilieren von TypeScript, nicht für alle Build Systeme. Alles in allem erhöht sich zwar die Build Komplexität und Dauer, allerdings hat es auch Vorteile. Zum einen führt es dazu, dass das Frontend vom Backend vollständig isoliert ist und die reine Frontendentwicklung ohne unnötige Abhängigkeiten und Build Prozesse ablaufen kann. Zum anderen erweitert sich dadurch auch die Auswahl an vorhanden Tools und Plugins, was die Entwicklung deutlich vereinfachen kann.

Für dieses Projekt habe ich mich für Maven als System für das Backend und npm als System für das Frontend entschieden. Die Entscheidung wurde auf Grund der hohen Beliebtheit und der persönlichen Expertise mit diesen Tools getroffen. Aus reiner Systemsicht ist das Frontend ein Teil des Backends, da es für die Auslieferung des Frontends an den Nutzer verantwortlich ist. Der Build beginnt daher auch in Maven. Als erstes muss sichergestellt werden, dass npm und Node.js auf dem Rechner vorhanden sind. Für die Kommunikation mit diesen nutze ich das frontend-maven-plugin[1], welches sie in einem ersten Schritt in einen lokalen Ordner installiert, solange sie noch nicht vorhanden sind. Anschließend wird “npm install” aufgerufen, um alle deklarierten Abhängigkeiten aus dem Frontend zu installieren. Dazu zählt zum Beispiel THREE.js oder auch der TypeScript Transpiler. Anschließend wird der Build an das Frontend übergeben, indem ein fest definiertes npm Script aufgerufen wird. Dieses führt als erstes den TypeScript Transpiler aus um die benötigten JavaScript Dateien anzulegen.

[1] `https://github.com/eirslett/frontend-maven-plugin`

6.2. Visualisierung

Für die Visualisierung wurde eine schematische Darstellung gewählt, bei der keine weiteren Datenquellen notwendig sind. Bei dieser topographische Darstellung, bekannt aus Atlanten, ist der Farbwert abhängig vom Höhenwert und wird aus vordefinierten Farbbereichen interpoliert. Der Hauptgrund für diese Wahl ist vor allem der begrenzte Projektzeitraum. Ein Einbinden von weiteren Quellen zur Bestimmung der Oberflächenbeschaffenheit (z.B. Gesteinsart oder Eis) hätte die Komplexität noch weiter erhöht und wurde in diesem Projekt als nicht durchführbar eingestuft.

Für die eigentliche Visualisierung kommen Shader zum Einsatz. Dafür muss dem Shader natürlich der Höhenwert bekannt sein. Bei einer flachen Projektion lässt sich dieser aus dem Vertex ablesen (entsprechend skalierte y-Koordinate), dies ändert sich natürlich, sobald eine sphärische Projektion zum Einsatz kommt. Die einfachste Möglichkeit ist es, dem Shader den Höhenwert in Metern neben dem Vertex als weiteres Attribut zu übergeben. Da dies die theoretische maximale Speichernutzung allerdings um weitere 4 GB erhöhen würde, was einer Erhöhung um 1/3 der Gesamtmenge entspricht, wurde dieser Ansatz verworfen. Stattdessen ist effizienter, den Höhenwert aus dem Vertex zu berechnen. Da der Höhenwert immer der Abweichung von einem vordefinierten Radius entspricht (ähnlich dem Meeresspiegel auf der Erde), muss dieser Radius einfach von der Länge des Vertex abgezogen werden. Dies funktioniert allerdings nur, wenn der Mittelpunkt des Modells auch mit dem Koordinaten-Nullpunkt übereinstimmt, da nur dann die Länge des Vertex der Distanz zum Punkt auf der Oberfläche entspricht. Um dies im Vertexshader zu implementieren, müssen ihm die verwendete Projektion, der Radius und die Skalierung von GL Einheiten zu Metern bekannt sein. Diese werden als Uniform Werte übergeben, sind also nicht abhängig von der Vertex-Anzahl und spielen daher für die maximale Speichernutzung keine Rolle. Der Vertexshader berechnet also wie beschrieben die Höhe in Metern und gibt sie an den Fragmentshader weiter. Des Weiteren transformiert er wie üblich den Vertex mit der Model-Matrix (enthält die Transformationen des Modells), der View-Matrix (enthält die inversen Transformationen der Kamera) und der Projektions-Matrix (enthält die perspektivische Transformation der Kamera) um die endgültige Position des Vertex zu bestimmen.

Der Fragmentshader hat nun die Aufgabe, aus diesem Höhenwert einen Farbwert zu generieren. Eine Möglichkeit wäre es, einfach verschiedene Grenzen zu definieren und diesen Grenzen feste Farbwerte zuzuweisen. Dann kann geprüft werden, welchem Bereich der Höhenwert entspricht und der endgültige Farbwert entspricht dann einer Variation des Farbwerts des Bereichs. Da dies allerdings mehrere Verzweigungen (conditionals) zur Prüfung der Grenzen erfordert und dies in der Shaderentwicklung vermieden werden sollte[2], wurde eine bessere Lösung gesucht. Insbesondere, da es sich um das sogenannte dynamic branching handelt, da da die Bedingung abhängig vom Höhenwert ist, welcher natürlich pro Vertex anders ist. Des Weiteren wurde auf Grund der Datenmenge die kritischste Stelle der Performance (bottleneck) eher auf GPU Seite angesehen, sodass hier dringender auf der Performance geachtet werden sollte. Schlussendlich wurde der Fakt genutzt, das der Hue-Wert im HSV-Farbraum eine relativ lineare Verteilung verschiedener Farben enthält und so gut als Farbskala

[2] siehe [1], Kapitel 14, Abschnitt Avoid Dynamic Branching, S. 273

genutzt werden kann. Da die Ausgabe allerdings im RGBA-Format erfolgen muss, ist hier eine Umwandlung des HSV Werten in diesen Farbraum erforderlich. Es wird also der Prozentwert des aktuellen Höhenwerts abhängig von vom Nutzer definierten Grenzen berechnet und diesem Prozentwert ein Hue Wert zugeordnet. Dabei wurde der Farbraum vorher noch verkleinert und invertiert, sodass die Farben dann von einem Blau-Ton (niedrigster Wert) zu einem Rot-Ton (höchster Wert) reichten.

6.3. Kamerabewegung

Die Steuerung der Kamera zur Betrachtung von verschiedenen Stellen ist ein wichtiger Aspekt im Design des Prototypen. Hierbei muss darauf geachtet werden, dass die Bewegung sowohl aus der Ferne flüssig ist, also auch bei hoher Zoomstufe noch präzise bestimmte Orte betrachtet werden können. Auch muss die Steuerung intuitiv sein und darf keine textuellen Erklärung benötigen.

Die erste Implementierung war eine frei im Raum bewegbare Kamera, welche man mit der Tastatur steuern konnte. Als konkrete Tasten wurden zum einen die Pfeiltasten als auch die übliche Alternative WASD genutzt. Diese sind allgemein als Steuerungstasten bekannt und sollten daher keiner Erklärung bedürfen. Die Kamera bewegte sich dabei entlang des lokalen Koordinatensystems der Kamera. Dieses kann dann mit der Maus entlang der x-Achse (pitch) und y-Achse (yaw) gedreht werden. Eine Drehung um die z-Achse (roll) verkompliziert die Steuerung und wurde daher nicht implementiert. Eine Bewegung der Maus auf der x-Achse führt dabei zu einer Rotation der Kamera entlang der y-Achse und eine Bewegung auf der y-Achse zu einer Rotation entlang der x-Achse. Die Blickrichtung der Kamera folgt also effektiv der Bewegung der Maus. Dabei wurde das Drehen nur beim Gedrückthalten der Maustaste (dragging) durchgeführt, da die Kamera sich sonst natürlich bei der normalen Navigation auf der Seite bewegen würde. Das Gedrückthalten ist dabei schon weniger intuitiv, allerdings entspricht es der physischen Bewegung des Ziehens an einer Seite des Globus in der realen Welt. Hier ist allerdings eine genauere Evaluation der Steuerung notwendig um eine gute User-Experience zu gewährleisten.

Wichtig bei der Implementierung ist, dass die Geschwindigkeit der Bewegung nicht von der Geschwindigkeit des Browser abhängen darf. Daher müssen alle Vektoren, welche eine Bewegung darstellen, mit der Zeit multipliziert werden, die seit dem letzten Aufruf der Bewegung vergangen ist. Nur so wird in einem bestimmten Zeitabschnitt immer die gleiche Länge zurückgelegt. Ein weiterer Aspekt ist, dass das Gedrückthalten einer Taste natürlich zu einer kontinuierlichen Bewegung führen soll. Um dies zu erkennen kann zum einen das keydown-Event des Browser genutzt werden, welches auch gesendet wird, solange die Taste gedrückt bleibt. Allerdings ist dabei eine spürbare Verzögerung zwischen erstem und nachfolgenden Events vorhanden, sodass die Bewegung initial ziemlich ruckartig erfolgt. Auch ist die Geschwindigkeit, mit der die Events gesendet werden, nicht definiert und kann so zu sehr ruckartigen Bewegungen führen, sollte die Rate weit unter der Bildwiederholrate des Monitors liegen.

7. Evaluation

8. Fazit

A. Appendix

A.1. Redundanzberechnung

```
procedure findRedundancies(data)
    result, visited := boolean array with false values;
    for all x, z in data do
        if not visited[z][x] then
            rowsWidth, rowsHeight := checkRows(data, x, z);
            columnsWidth, columnsHeight := checkColumns(data, x, z);
            width, height := Minimum of both widths and heights;
            if width, height > 3 then
                for all xx, zz in width, height do
                    if not (0, 0 or 0, height or width, 0 or width, height) then
                        result[z + zz][x + xx] := true
                    end if
                    visited[z + zz][x + xx] := true
                end for
            end if
        end if
    end for
    return result
end procedure
```

Der beschriebene Algorithmus iteriert über alle Werte in den übergebenen Daten und versucht an jeder Stelle das größtmögliche Raster zu finden, bei dem alle Spalten und Reihen eine lineare Abhängigkeit aufweisen. Die Anzahl und Länge der Reihen, sowie die Anzahl und Länge der Spalten, bei den dies hintereinander zutrifft wird also ermittelt und die kleinste Überschneidung in jeweils Breite und Höhe entsprechen den Dimensionen des umfassenden Rasters. Solange dieses Raster die Mindestbreite und -höhe von 3 erreicht hat, können alle Werte außer den Eckpunkten als redundant markiert werden. Zusätzlich können alle Punkte dieses Rasters als besucht markiert werden, damit zukünftige Iterationen dies nicht erneut berechnen müssen.

```
procedure checkRows(data, currentX, currentZ)
    width, height := 0
    for z := currentZ to data.height do
        difference := data[currentZ][currentX] - data[currentZ][currentX + 1];
        rowSize := 2;
        for x := currentX + 1 to data.width - 1 do
            if data[z][x] - data[z][x + 1] == difference then
                if width not 0 and width <= rowSize then
                    break;
                end if
                rowSize := rowSize + 1;
            else if rowSize < 3 then
                return width, height;
            else
                break;
            end if
        end for
        width := rowSize;
        height := height + 1;
    end for
    return width, height;
end procedure
```

Um die horizontalen linearen Abhängigkeiten zu finden muss vom aktuellen Punkt über alle verbliebenen Reihen iteriert werden. Die lineare Abhängigkeit kann geprüft werden, indem nacheinander die Differenzen der verbliebenen Werte in der Reihe verglichen werden. Solange diese gleich sind wird die Länge aktualisiert. Sollte bereits eine kürzere Länge gefunden worden sein, kann mit der nächsten Reihe weitergemacht werden. Sobald dies nicht mehr zutrifft und die aktuelle Länge kleiner als die Mindestlänge 3 ist, wird die aktuelle Reihe und die kürzeste Länge zurückgeliefert, ansonsten wird mit der nächsten Reihe weitergemacht. Der Algorithmus zum Bestimmen der Spalten ist equivalent und daher nicht aufgeführt.

Literatur

[1] Kyle Halladay. *Practical Shader Development*. 2019. DOI: 10.1007/978-1-4842-4457-9.

[2] Randi Rost John Kessenich Dave Baldwin. *The OpenGL® Shading Language, Version 4.60.7*. 2019. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[3] MOLA Team. *Mars MGS MOLA DEM 463m v2*. 2020. URL: https://astrogeology.usgs.gov/search/details/Mars/GlobalSurveyor/MOLA/Mars_MGS_MOLA_DEM_mosaic_global_463m.

[4] Raymond J. Dezzani Yue-Hong Chou Pin-Shuo Liu. „Terrain complexity and reduction of topographic data“. In: *Journal of Geographical Systems* (1999). DOI: 10.1007/s101090050011.